БАЙЕСОВСКАЯ ОПТИМИЗАЦИЯ

С ПРИМЕРАМИ ИЗ БИБЛИОТЕК PYTHON



Куан Нгуен

Предисловие Луиса Серрано
и Дэвида Свита

MANNING

alist

Bayesian Optimization in Action

QUAN NGUYEN
FOREWORDS BY LUIS SERRANO AND DAVID SWEET



MANNING
SHELTER ISLAND

Куан Нгуен

БАЙЕСОВСКАЯ ОПТИМИЗАЦИЯ

С ПРИМЕРАМИ ИЗ БИБЛИОТЕК PYTHON

Астана
«АЛИСТ»
2024

УДК 004.4
ББК 32.973.26-018.1
Н37

Нгуен К.

Н37 Байесовская оптимизация с примерами из библиотек Python: Пер. с англ. — Астана: АЛИСТ, 2024. — 416 с.: ил.

ISBN 978-601-08-4437-7

Книга рассказывает об оптимизации моделей машинного обучения на основе байесовского метода и теории вероятностей. Примеры составлены на языке Python с использованием библиотек PyTorch, GPyTorch и BoTorch. Описаны приемы настройки гиперпараметров, гауссовское распределение, использование политик машинного обучения, в частности на основе задачи о многоруких бандитах, повышение производительности вычислений, многовариантная оптимизация с учетом полезности и затрат, а также применение байесовского метода в специализированных сценариях.

Для специалистов по машинному обучению и математическим алгоритмам

УДК 004.4
ББК 32.973.26-018.1

© 2024 ALIST LLP

Authorized translation of the English edition © 2023 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

Авторизованный перевод английской редакции книги © 2023 Manning Publications.
Перевод опубликован и продается с разрешения компании-правообладателя Manning Publications.
Все права защищены.

ISBN 978-1-63343-907-8 (англ.)
ISBN 978-601-08-4437-7 (каз.)

© Manning Publications, 2023
© Издание на русском языке. ТОО "АЛИСТ", 2024

Оглавление

Введение 11

Пролог 13

Благодарности 15

О книге 17
Кому следует прочитать эту книгу? 17
Как организована эта книга: дорожная карта 17
О коде 20
Дискуссионный форум liveBook 21
Архив с иллюстрациями к книге 21

Об авторе 22
О техническом редакторе 22

Об изображении на обложке 23

Глава 1. Введение в байесовскую оптимизацию 24
1.1. Нахождение оптимума дорогостоящей функции «черного ящика» 25
1.1.1. Настройка гиперпараметров как пример дорогостоящей задачи оптимизации «черного ящика» 25
1.1.2. Проблема дорогостоящей оптимизации «черного ящика» 27
1.1.3. Другие реальные примеры дорогостоящих задач оптимизации «черного ящика» 28
1.2. Введение в байесовскую оптимизацию 30
1.2.1. Моделирование с помощью гауссовского процесса 31
1.2.2. Принятие решений с помощью политики БО 34
1.2.3. Объединение ГП и политики БО для формирования цикла оптимизации 35
1.2.4. Байесовская оптимизация в действии 37
1.3. Что вы узнаете в этой книге? 43
Заключение 44

ЧАСТЬ I. МОДЕЛИРОВАНИЕ С ПОМОЩЬЮ ГАУССОВСКИХ ПРОЦЕССОВ 45

Глава 2. Гауссовский процесс как распределение по функциям 47
2.1. Как продать свой дом по методу Байеса 49
2.2. Моделирование корреляций с помощью многомерных гауссовских распределений и байесовских обновлений 51
2.2.1. Использование многомерных распределений Гаусса для совместного моделирования нескольких переменных 51

2.2.2. Обновление нормальных распределений ... 55
2.2.3. Моделирование множества переменных с помощью многомерных гауссовских распределений ... 58
2.3. Переход от конечной гауссианы к бесконечной ... 60
2.4. Реализация ГП в Python ... 65
2.4.1. Настройка обучающих данных ... 65
2.4.2. Реализация класса ГП ... 67
2.4.3. Создание прогнозов с помощью ГП ... 69
2.4.4. Визуализация прогнозов ГП ... 70
2.4.5. Выход за рамки одномерных целевых функций ... 73
2.5. Упражнение ... 75
Заключение ... 77

Глава 3. Настройка ГП с помощью функций среднего значения и ковариации ... 78
3.1. Важность априорных знаний в байесовских моделях ... 79
3.2. Включение предварительных знаний в ГП ... 82
3.3. Определение поведения целевой функции с помощью функции среднего значения ... 83
3.3.1. Использование функции нулевого среднего в качестве базовой стратегии ... 84
3.3.2. Использование постоянной функции с градиентным спуском ... 86
3.3.3. Использование линейной функции с градиентным спуском ... 90
3.3.4. Использование квадратичной функции с помощью реализации пользовательской функции среднего ... 92
3.4. Определение изменчивости и гладкости с помощью ковариационной функции ... 95
3.4.1. Настройка масштабов функции ковариации ... 95
3.4.2. Управление гладкостью с помощью различных ковариационных функций ... 99
3.4.3. Моделирование различных уровней изменчивости с помощью множественных масштабов длины ... 101
3.5. Упражнение ... 104
Заключение ... 106

ЧАСТЬ II. ПРИНЯТИЕ РЕШЕНИЙ С ПОМОЩЬЮ БАЙЕСОВСКОЙ ОПТИМИЗАЦИИ ... 107

Глава 4. Достижение наилучшего результата с помощью политик, основанных на улучшениях ... 109
4.1. Навигация по пространству поиска в БО ... 110
4.1.1. Цикл БО и политики ... 110
4.1.2. Баланс между исследованием и эксплуатацией ... 120
4.2. Поиск улучшений в БО ... 122
4.2.1. Измерение улучшения с помощью ГП ... 123
4.2.2. Вычисление вероятности улучшения ... 127
4.2.3. Запуск политики PoI ... 131
4.3. Оптимизация ожидаемого значения улучшения ... 135
4.4. Упражнения ... 139
4.4.1. Упражнение 1: поощрение исследования с помощью PoI ... 139
4.4.2. Упражнение 2: БО для настройки гиперпараметров ... 140
Заключение ... 142

Глава 5. Исследование пространства поиска с помощью «многорукого бандита» ... 143
5.1. Введение в задачу многорукого бандита ... 144
5.1.1. Как найти наилучший игровой автомат в казино ... 144
5.1.2. От MAB к БО ... 147

5.2. Как быть оптимистом в условиях неопределенности с помощью политики верхней доверительной границы ... 149
5.2.1. Оптимизм в условиях неопределенности ... 149
5.2.2. Баланс между исследованием и эксплуатацией ... 152
5.2.3. Реализация с помощью BoTorch ... 154
5.3. Интеллектуальная выборка с помощью политики выборки Томпсона ... 157
5.3.1. Одна выборка для представления неизвестного ... 157
5.3.2. Реализация с помощью BoTorch ... 160
5.4. Упражнения ... 166
5.4.1. Упражнение 1: Настройка плана эксплуатации для UCB ... 167
5.4.2. Упражнение 2: БО для настройки гиперпараметров ... 167
Заключение ... 168

Глава 6. Использование теории информации с политиками на основе энтропии ... 170
6.1. Измерение знаний с помощью теории информации ... 171
6.1.1. Измерение неопределенности с помощью энтропии ... 171
6.1.2. Поиск пульта дистанционного управления с помощью энтропии ... 174
6.1.3. Бинарный поиск с применением энтропии ... 177
6.2. Энтропийный поиск в БО ... 183
6.2.1. Поиск оптимума с помощью теории информации ... 183
6.2.2. Реализация энтропийного поиска с помощью BoTorch ... 187
6.3. Упражнения ... 190
6.3.1. Упражнение 1: Включение предварительных (априорных) знаний в энтропийный поиск ... 190
6.3.2. Упражнение 2: Байесовская оптимизация для настройки гиперпараметров ... 192
Заключение ... 193

ЧАСТЬ III. РАСШИРЕНИЕ БАЙЕСОВСКОЙ ОПТИМИЗАЦИИ НА СПЕЦИАЛИЗИРОВАННЫЕ СЦЕНАРИИ ... 195

Глава 7. Максимизация производительности за счет оптимизации пакетной обработки ... 197
7.1. Одновременное выполнение нескольких оценок функций ... 198
7.1.1. Параллельное использование всех доступных ресурсов ... 198
7.1.2. Почему нельзя использовать обычные политики БО в пакетном режиме? ... 201
7.2. Вычисление улучшения и верхней доверительной границы для набора точек ... 203
7.2.1. Расширение эвристик оптимизации для пакетного режима ... 203
7.2.2. Реализация пакетных политик PoI, EI и UCB ... 210
7.3. Упражнение 1: Расширение TS до пакетной оптимизации с помощью повторной выборки ... 216
7.4. Вычисление ценности пакета точек с помощью теории информации ... 217
7.4.1. Нахождение наиболее информативного пакета точек с помощью цикличного уточнения ... 217
7.4.2. Реализация пакетного энтропийного поиска с помощью BoTorch ... 220
7.5. Упражнение 2: Оптимизация конструкции самолета ... 222
Заключение ... 225

Глава 8. Удовлетворение особых ограничений с помощью ограниченной оптимизации ... 227
8.1. Учет ограничений в задаче ограниченной оптимизации ... 228
8.1.1. Ограничения могут изменить решение задачи оптимизации ... 228
8.1.2. Структура БО, учитывающая ограничения ... 231

8.2. Принятие решений с учетом ограничений в БО ... 232
8.3. Упражнение 1: Ручное вычисление ограниченной EI ... 238
8.4. Реализация ограниченной EI с помощью BoTorch ... 239
8.5. Упражнение 2: Ограниченная оптимизация при конструировании самолета ... 243
Заключение ... 245

Глава 9. Балансирование между полезностью и затратами с помощью многовариантной оптимизации ... 246
9.1. Использование приближений с низкой точностью для изучения дорогостоящих явлений ... 248
9.2. Разноточное моделирование с помощью ГП ... 251
9.2.1. Форматирование разноточного набора данных ... 251
9.2.2. Обучение разноточного ГП ... 255
9.3. Баланс между информацией и затратами в разноточной оптимизации ... 260
9.3.1. Моделирование затрат на запросы различной точности ... 260
9.3.2. Оптимизация количества информации для управления оптимизацией ... 262
9.4. Измерение производительности в разноточной оптимизации ... 267
9.5. Упражнение 1: Визуализация средней производительности в разноточной оптимизации ... 272
9.6. Упражнение 2: Разноточная оптимизация с использованием нескольких низкоточных приближений ... 273
Заключение ... 275

Глава 10. Обучение на парных сравнениях с помощью оптимизации предпочтений ... 277
10.1. Оптимизация «черного ящика» с помощью парных сравнений ... 279
10.2. Формулировка задачи оптимизации предпочтений и форматирование данных парного сравнения ... 282
10.3. Обучение ГП, основанного на предпочтениях ... 286
10.4. Оптимизация предпочтений с помощью игры «Царь горы» ... 290
Заключение ... 294

Глава 11. Оптимизация нескольких целевых функций одновременно ... 296
11.1. Балансирование нескольких целей оптимизации с помощью БО ... 297
11.2. Нахождение границы наиболее оптимальных точек данных ... 299
11.3. Стремление к улучшению оптимальной границы данных ... 307
11.4. Упражнение: Многоцелевая оптимизация для конструирования самолета ... 313
Заключение ... 314

ЧАСТЬ IV. СПЕЦИАЛЬНЫЕ МОДЕЛИ ГАУССОВСКОГО ПРОЦЕССА ... 315

Глава 12. Масштабирование гауссовского процесса для больших наборов данных ... 317
12.1. Обучение ГП на большом наборе данных ... 319
12.1.1. Постановка задачи обучения ... 319
12.1.2. Обучение обычного ГП ... 322
12.1.3. Проблемы с обучением обычного ГП ... 324
12.2. Автоматический выбор репрезентативных точек из большого набора данных ... 327
12.2.1. Минимизация различий между двумя ГП ... 327
12.2.2. Обучение модели небольшими пакетами ... 329
12.2.3. Реализация приближенной модели ... 332

12.3. Достижение лучшей оптимизации с учетом геометрии поверхности потерь ... 338
12.4. Упражнение ... 344
Заключение ... 346

Глава 13. Комбинирование гауссовских процессов с нейросетями ... 347
13.1. Данные, имеющие структуру ... 348
13.2. Нахождение сходства в структурированных данных ... 351
13.2.1. Использование ядра с GPyTorch ... 352
13.2.2. Работа с изображениями в PyTorch ... 352
13.2.3. Вычисление ковариации двух изображений ... 354
13.2.4. Обучение ГП на изображениях ... 355
13.3. Использование нейросетей для обработки сложных структурированных данных ... 359
13.3.1. Зачем использовать нейросети для моделирования? ... 359
13.3.2. Реализация комбинированной модели в GPyTorch ... 361
Заключение ... 368

ПРИЛОЖЕНИЕ. Решения для упражнений ... 369
П.1. Глава 2: Гауссовский процесс как распределение по функциям ... 369
П.2. Глава 3: Объединение предварительных знаний с функциями среднего и ковариации ... 372
П.3. Глава 4: Достижение наилучшего результата с помощью политик, основанных на улучшениях ... 374
П.3.1. Упражнение 1: Поощрение исследований с помощью вероятности улучшения ... 375
П.3.2. Упражнение 2: БО для настройки гиперпараметров ... 376
П.4. Глава 5: Исследование пространства поиска с помощью политик «многорукого бандита» ... 382
П.4.1. Упражнение 1: Настройка плана исследования для верхней доверительной границы ... 383
П.4.2. Упражнение 2: БО для настройки гиперпараметров ... 384
П.5. Глава 6: Использование теории информации с политиками, основанными на энтропии ... 387
П.5.1. Упражнение 1: Включение предварительных знаний в энтропийный поиск ... 388
П.5.2. Упражнение 2: БО для настройки гиперпараметров ... 391
П.6. Глава 7: Увеличение пропускной способности с помощью пакетной оптимизации ... 392
П.6.1. Упражнение 1: Расширение TS для пакетного режима с помощью повторной выборки ... 392
П.6.2. Упражнение 2: Оптимизация конструкции самолета ... 393
П.7. Глава 8: Удовлетворение дополнительных ограничений с помощью ограниченной оптимизации ... 397
П.7.1. Упражнение 1: Ручное вычисление ограниченной политики EI ... 397
П.7.2. Упражнение 2: Ограниченная оптимизация конструкции самолета ... 400
П.8. Глава 9: Баланс между полезностью и затратами с помощью разноточной оптимизации ... 402
П.8.1. Упражнение 1: Визуализация средней производительности при разноточной оптимизации ... 403
П.8.2. Упражнение 2: Разноточная оптимизация с использованием нескольких низкоточных приближений ... 405
П.9. Глава 11: Оптимизация нескольких целевых функций одновременно ... 407
П.10. Глава 12: Масштабирование гауссовских процессов на большие наборы данных ... 409

Предметный указатель ... 414

Моему лучшему другу, Ньюнгу

Введение

Поскольку сложность задач, с которыми мы сталкиваемся в *машинном обучении* (**Machine Learning**, **ML**) и смежных областях, продолжает расти, все более важным становится умение оптимизировать использование ресурсов и способность эффективно принимать обоснованные решения. *Байесовская оптимизация* — мощный метод, помогающий находить максимумы и минимумы целевых функций, оценка которых требует больших затрат. Это очень полезное решение данной проблемы, поскольку любую функцию можно рассматривать как «черный ящик», что позволяет исследователям и практикам решать очень сложные функции, используя *байесовский вывод* в качестве основного метода оптимизации.

Из-за своей сложности байесовская оптимизация является менее доступной для начинающих специалистов в машинном обучении, чем другие методы. Однако такой инструмент должен присутствовать в наборе каждого, кто хочет достичь наилучших результатов. Для освоения этой темы нужно иметь очень хорошее представление о математических вычислениях и теории вероятностей.

Здесь данная книга и приходит на помощь. В ней Кван красиво и успешно демистифицирует эти сложные концепции. Практический подход, четкие диаграммы, примеры из реальной жизни и полезные листинги кода помогают автору приоткрыть завесу над этой темой, как с теоретической, так и с практической точки зрения.

Автор использует свой обширный опыт специалиста по данным (Data Science) и преподавателя с целью дать читателю очень четкое представление об этих методах и о том, как их можно применять для решения реальных проблем. Начиная с принципов байесовского вывода, в книге постепенно развиваются концепции байесовской оптимизации и модели *гауссовского процесса*. Кван рассказывает о современных библиотеках, таких как GPyTorch и BoTorch, исследуя их использование сразу в нескольких областях.

Эта книга является обязательной к прочтению для любого специалиста по data science или ML, который хочет использовать истинную мощь байесовской оптимизации для решения реальных задач. Я настоятельно рекомендую ее всем, кто хочет овладеть искусством оптимизации с помощью байесовского вывода.

– Луис Серрано,
доктор философии,
ученый и популяризатор искусственного интеллекта,
автор книги *«Grokking Machine Learning»*

Инженеры и ученые сталкиваются с общим вызовом, который важен для закрепления ценности их исследований и творчества. Им нужна оптимизация. Инженеры по машинному обучению работают с *гиперпараметрами*, которые делают модели генерализованными (обобщенными). Физики настраивают лазеры на свободных электронах на максимальную энергию импульса. Программисты настраивают сборщики мусора JVM (Java Virtual Machine) с целью максимизировать пропускную способность сервера. Материаловеды выбирают морфологию микроструктуры, которая максимально увеличивает светопоглощение солнечной батареи. В каждом примере есть проектные решения, которые невозможно сделать на основе лишь базовых принципов. Они выполнены с помощью экспериментальных измерений.

Чтобы оценить что-то экспериментально, нужно создать программное обеспечение, запустить оборудование или построить новый объект, одновременно измеряя его производительность. Чтобы найти хороший дизайн, нужно провести его оценку. Такие вычисления требуют времени, стоят денег и могут быть сопряжены с риском. Поэтому крайне важно, чтобы их количество, необходимое для поиска оптимальной конструкции, было минимальным. В этом и есть вся суть байесовской оптимизации.

В своей работе на протяжении последних 20 лет я использовал данный подход и связанные с ним методы-предшественники. За это время академические исследования и отчеты о промышленном применении улучшили производительность и расширили возможности применения байесовской оптимизации. В настоящее время существуют высококачественные программные инструменты и методы, которые используются при создании оптимизаторов для конкретных проектов.

Текущее состояние можно рассматривать как аналог прогнозирования с помощью линейных моделей. Инженеры, желающие построить линейную модель, могут обнаружить, что такие программные инструменты, как sklearn, позволяют им разрабатывать модели для разных типов (например, непрерывных или категориальных) и различного количества входных и выходных переменных, выполнять автоматический их выбор и измерять качество генерализации. Точно так же инженеры, желающие создать байесовский оптимизатор, могут убедиться, что BoTorch (построенный на GPyTorch, Pyro и PyTorch) предоставляет инструменты, которые способны оптимизировать различные типы переменных, максимизировать несколько целей, обрабатывать ограничения и многое другое.

В этой книге изучение байесовской оптимизации начинается с ее самых основных компонентов (*регрессия гауссовского процесса* и *численная оптимизация функции сбора оценки*) и заканчивается новейшими методами по обработке большого количества оценок (наблюдений) и экзотических пространств проектирования. Попутно она охватывает все специализации, которые могут вам понадобиться в определенном проекте: обработка ограничений, множественные цели, параллельная оценка, а также оценка через парные сравнения. Вы найдете много технической информации, чтобы освоить инструменты и методы, а также увидите много примеров реального кода, которые можно использовать в работе.

Несмотря на весь успех байесовской оптимизации, существует мало литературы, ориентированной на новичков. Эта книга прекрасно заполняет данную нишу.

– Дэвид Свит,
адъюнкт-профессор университета Йешива (Yeshiva),
автор книги *«Experimentation for Engineers»*

Пролог

Осенью 2019 года я был аспирантом на первом курсе и не знал, над какой проблемой работать в своем исследовании. Я знал, что хочу сосредоточиться на искусственном интеллекте (ИИ). Было что-то привлекательное в автоматизации мыслительных процессов с помощью компьютеров. Но это огромная область, и мне было трудно сузить свою работу до конкретной темы.

Вся неопределённость исчезла, когда я прошёл курс *«Байесовские методы машинного обучения»*. К этому моменту я уже несколько раз встречался с теоремой Байеса, когда учился на бакалавриате, но именно на первых лекциях этого курса все стало получаться! Теорема Байеса может предложить интуитивный способ поразмышлять о вероятностях, и, как по мне, это элегантная модель человеческих убеждений. У каждого из нас есть *априорное убеждение* (изначальное мнение о чем-либо), с которого мы начинаем. Оно обновляется по мере того, как мы встречаем доказательства за или против него. Результатом является *апостериорное убеждение*, отражающее как априорное предположение (изначальное), так и данные. Тот факт, что теорема Байеса дает элегантный способ поддерживать убеждения в ИИ и находит применение во многих задачах, стал для меня мощным сигналом, что байесовское машинное обучение — это тема, которой стоит заниматься.

К тому времени, когда мы добрались до лекции по байесовской оптимизации (БО), решение было уже принято: теория интуитивно понятна, приложений много, возможностей для построения еще больше. Опять же, что-то врожденное привлекло (и продолжает привлекать) меня к автоматизации мышления, а если точнее, к принятию решений. БО стала идеальным аттракционом. Я попал в исследовательскую лабораторию Романа Гарнетта (Roman Garnett), профессора, ведущего курс. И мое путешествие по БО началось!

Перенесемся в 2021 год. Я потратил некоторое время на исследование и внедрение решений этого метода, и мое понимание БО только выросло. Я бы порекомендовал его друзьям и коллегам при решении сложных задач оптимизации с полной уверенностью, что он будет работать хорошо. Была только одна проблема: отсутствовал хороший ресурс, на который я мог бы указать. Исследовательские работы изобиловали математикой, онлайн-уроки были слишком короткими, чтобы дать содержательную информацию, а учебные пособия по программному обеспечению БО были разрозненными и не содержали хорошего описания.

Тогда мне в голову пришла идея в виде цитаты Тони Моррисона (Toni Morrison): *«Если есть книга, которую вы хотите прочитать, но она еще не написана, тогда*

вы должны ее написать». Как верно сказано! Эта перспектива взволновала меня по двум причинам. Во-первых, я мог бы написать книгу о чем-то близком и дорогом моему сердцу. Во-вторых, писательство, несомненно, помогло бы мне достичь еще более глубокого понимания. Я подготовил предложение и связался с Manning (издателем моих любимых книг) в том стиле, который я себе представлял.

В ноябре 2021 года мой редактор по сбору данных Энди Уолдрон (Andy Waldron) прислал мне электронное письмо, ознаменовав этим самое первое сообщение от Manning. В декабре 2021 года я подписал контракт и начал писать, что, как позже выяснилось, потребовало больше времени, чем я изначально предполагал (я уверен, что это касается любой книги). В апреле 2023 года я написал это предисловие как один из последних этапов перед публикацией!

Благодарности

Чтобы воспитать ребенка и написать книгу, нужна целая деревня. Ниже представлена лишь небольшая часть ее жителей, которые мне очень помогли в процессе работы.

Моя первая и главная благодарность моим родителям Бангу (Bang) и Лан (Lan), чья постоянная поддержка позволила мне бесстрашно исследовать неизведанное: отучиться за границей, получить докторскую степень и, конечно же, написать книгу. И я хотел бы искренне поблагодарить мою сестру и доверенное лицо – Ню (Nhu), которая всегда находилась рядом, чтобы помочь мне в самые трудные времена.

Байесовская оптимизация составляет значительную часть моих докторских исследований, и я хотел бы поблагодарить участников программы, которые действительно сделали мой опыт бесценным. Особая благодарность моему консультанту, Роману Гарнетту, который без особых усилий убедил меня продолжить исследования в области байесовского машинного обучения. Ты тот, кто все это начал. Я также благодарю своих друзей из лаборатории активного обучения: Еу Чен (Yehu Chen), Шаян Монаджеми (Shayan Monadjemi) и профессора Алвитта Оттли (Alvitta Ottley). Награда за докторскую степень не очень велика, и работа с вами составляет большую ее часть.

Также я выражаю благодарность замечательной команде издательства Manning. Я благодарю моего редактора по разработке Марину Майклс (Marina Michaels), которая руководила этим проектом с самого первого дня, проявляя высочайший уровень профессионализма, заботы, поддержки и терпения. Мне очень повезло, что я встретил вас в нашем проекте. Спасибо моему редактору по сбору данных Энди Уолдрону за веру в эту идею, несмотря на то, что над подобной темой уже работал другой автор (который лучше меня), а также Ивану Мартиновичу (Ivan Martinovic) за помощь с вопросами по AsciiDoc и терпеливое исправление моего кода.

Я хотел бы также поблагодарить следующих рецензентов: Аллан Макура (Allan Makura), Андрей Палейес (Andrei Paleyes), Карлос Айя-Морено (Carlos Aya-Moreno), Клаудиу Шиллер (Claudiu Schiller), Козимо Аттанаси (Cosimo Attanasi), Денис Лапчев (Denis Lapchev), Гэри Бэйк (Gary Bake), Джордж Онофрей (George Onofrei), Говард Бэнди (Howard Bandy), Иоаннис Атсониос (Ioannis Atsonios), Хесус Антонино Хуарес Геррего (Jesús Antonino Juárez Guerrero), Джош МакАдамс (Josh McAdams), Квеку Реджинальд Уэйд (Kweku Reginald Wade), Кайл Питерсон (Kyle Peterson), Локеш Кумар (Lokesh Kumar), Люциан Мирча Сасу (Lucian Mircea Sasu), Марк-Энтони Тейлор (Marc-Anthony Taylor), Марсио Николау (Marcio

Nicolau), Макс Део (Max Dehaut), Максим Волгин (Maxim Volgin), Мишель Ди Педе (Michele Di Pede), Мирерфан Геиби (Mirerfan Gheibi), Ник Декроос (Nick Decroos), Ник Васкес (Nick Vazquez), Ор Голан (Or Golan), Питер Хенсток (Peter Henstock), Филип Вайсс (Philip Weiss), Рави Киран Бамиди (Ravi Kiran Bamidi), Ричард Тобиас (Richard Tobias), Рохит Госвами (Rohit Goswami), Серджио Говони (Sergio Govoni), Шаби Икбал (Shabie Iqbal), Шриша Джагадиш (Shreesha Jagadeesh), Симон Сгуацца (Simone Sguazza), Шрирам Мачарла (Sriram Macharla), Шимон Харабаз (Szymon Harabasz), Томас Форис (Thomas Forys) и Влад Навицкий (Vlad Navitski).

При написании книги неизбежно возникают «слепые пятна». Именно рецензенты помогают их заполнить и сосредоточить внимание автора на том, что действительно важно. Выражаю особую благодарность Керри Койчу (Kerry Koitzsch) за его содержательные отзывы и Джеймсу Байлеки (James Byleckie) за его отличные предложения, как по коду, так и по его написанию.

Наконец, я благодарю команды, создавшие невероятные инструменты GPyTorch и BoTorch. Я пробовал и другие различные библиотеки для гауссовских процессов (ГП) и БО, но всегда возвращался к ним. Я надеюсь, что эта книга сможет создать и без того замечательное сообщество вокруг этих инструментов.

О книге

Раньше узнать о байесовской оптимизации (БО) можно было из онлайн-статей и туториалов к определенной библиотеке. Они были разрознены и по своей природе не углублялись в специфику. И их поиск отнимал время и силы. Вы также могли обратиться к техническим учебникам, но они, как правило, были слишком объемными и содержали много математики. Это представляло проблему, если вы практикующий специалист, который хотел бы сразу же приступить к делу.

Эта книга заполняет данный пробел, предлагая сочетание практик, ссылок на более глубокие материалы для заинтересованного читателя и готовых к использованию примеров кода. Она работает, сначала создавая интуитивно понятное представление о компонентах БО, а затем реализуя их на Python с использованием современного программного обеспечения.

Цель книги — предоставить доступное введение в байесовскую оптимизацию, основанное на глубоком понимании математики и теории вероятностей. Кроме того, заинтересованный читатель может найти дополнительные технические тексты, на которые в книге имеются ссылки, для более глубокого погружения в интересующую тему.

Кому следует прочитать эту книгу?

Эта книга принесет пользу специалистам по data science и ML, которые интересуются настройкой гиперпараметров, A/B-тестированием или экспериментируют с принятием решений (в общем смысле).

Она также будет полезна исследователям в таких научных областях, как химия, материаловедение и физика, которые сталкиваются со сложными задачами по оптимизации. Несмотря на то что в книге будет охвачена большая часть базовых знаний, необходимых для понимания содержания, аудитория должна быть знакома с такими общими концепциями машинного обучения, как *обучающие данные*, *прогностические модели*, *многомерные нормальные распределения* и прочие.

Как организована эта книга: дорожная карта

Книга разделена на четыре части. Каждая из них содержит несколько глав, охватывающих соответствующую тему:

- В главе 1 представлена байесовская оптимизация с использованием реальных сценариев применения. Она также включает (не вдаваясь в технические детали) наглядный пример, как БО может ускорить поиск глобального оптимума дорогостоящей функции.

Первая часть рассматривает гауссовские процессы как прогностическую модель функции, которую необходимо оптимизировать. Центральный тезис заключается в том, что гауссовские процессы предлагают калиброванную количественную оценку неопределенности, что важно в рамках нашей байесовской оптимизации. Эта часть состоит из двух глав:

- Глава 2 показывает, что гауссовские процессы являются естественным решением задачи обучения регрессионной модели на основе некоторых наблюдаемых данных. Такой процесс определяет распределение по функциям и может быть обновлен, чтобы отразить наше убеждение о значении функции, учитывая некоторые наблюдаемые данные.
- Глава 3 содержит два основных способа включить априорную информацию в гауссовский процесс: *функция нахождения среднего значения* и *функция ковариации*. Первая определяет общую тенденцию, а вторая показывает гладкость функции.

Во второй части перечислены политики байесовской оптимизации, которые представляют собой процедуры принятия решений, как следует выполнять оценки функций, чтобы глобальный оптимум можно было определить наиболее эффективно. Несмотря на то что разные политики мотивированы разными задачами, все они направлены на баланс между *исследованием* и *эксплуатацией*. Эта часть состоит из трех глав:

- В главе 4 обсуждается естественный способ принятия решений, какую оценку функции наиболее выгодно выполнить, учитывая улучшение, которое можно было бы получить от текущего наилучшего значения функции. Благодаря предположению об этой функции, основанному на гауссовском процессе, мы можем вычислить связанные с улучшением величины в закрытой форме и с минимальными затратами, что позволяет использовать две конкретные политики БО: *вероятность улучшения* (**Probability of Improvement**, **PoI**) и *ожидаемое улучшение* (**Expected Improvement**, **EI**).
- В главе 5 исследуется связь между байесовской оптимизацией и другим распространенным классом задач, которое называется *«Многорукий бандит»* (**Multi-Armed Bandit**, **MAB**). Мы научимся переносить политики многорукого бандита в настройку байесовской оптимизации и получим соответствующие стратегии: *верхнюю доверительную границу* (**Upper Confidence Bound**, **UCB**) и *выборку Томпсона* (**Thompson Sampling**, **TS**).
- В главе 6 рассматривается стратегия, которая уменьшает наибольшую неопределенность в наших убеждениях относительно глобального оптимума функции. Она представляет собой политику, основанную на энтропии, которая использует область математики, называемую *теорией информации*.

В третьей части представлены некоторые наиболее распространенные варианты использования, которые не совсем вписываются в рабочий процесс, разобранный к этому моменту в книге. А также показано, как можно модифицировать БО для решения различных задач оптимизации:

- В главе 7 представлена пакетная оптимизация, при которой для увеличения пропускной способности мы позволяем экспериментам выполняться параллельно. Например, можно обучать несколько экземпляров большой нейросети одновременно на кластере графических процессоров. Для этого политика оптимизации должна возвращать более одной рекомендации одновременно.
- В главе 8 обсуждаются критически важные для безопасности случаи использования, когда мы не можем свободно исследовать пространство поиска, поскольку некоторые оценки функций могут иметь пагубные последствия. Это приводит к сценарию, в котором существуют ограничения на то, как должна вести себя рассматриваемая функция, а также к потребности учитывать данные ограничения при разработке политики оптимизации.
- Глава 9 показывает, что при наличии доступа к множеству способов наблюдать за значениями функции на разных уровнях стоимости и точности (широко известному как *разноточность байесовской оптимизации*), учет переменных затрат может привести к повышению производительности оптимизации.
- В главе 10 рассматриваются парные сравнения, которые более точно отражают различные предпочтения человека, чем числовые оценки или рейтинги, поскольку они являются более простыми для понимания и создают меньшую когнитивную нагрузку на маркировщика. Эта глава расширяет представление о БО сначала с помощью специальной модели гауссовского процесса, а затем через использование существующих политик таким образом, чтобы они вписывались в рабочий процесс парного сравнения.
- В главе 11 рассматривается проблема многоцелевой оптимизации, где мы стремимся оптимизировать сразу несколько целевых функций, а также как БО может быть распространена на подобные задачи.

Четвертая часть посвящена специальным вариантам моделей гауссовских процессов, демонстрируя их гибкость и эффективность при моделировании и обеспечении прогнозов с калибровкой по неопределенности, даже вне контекста БО:

- В главе 12 мы узнаем, что в некоторых случаях получить решения обученного гауссовского процесса в замкнутой форме невозможно. Тем не менее высокоточные приближения все же могут быть получены с использованием сложных стратегий аппроксимации.
- Глава 13 демонстрирует, что благодаря экосистеме Torch объединение нейросетей PyTorch с гауссовскими процессами GPyTorch является простым процессом. Это позволяет нашим моделям стать более гибкими и выразительными.

Первые шесть глав принесут новичку большую пользу. Опытные специалисты, желающие внедрить байесовскую оптимизацию в свой вариант использования, могут найти ценность в главах с 7 по 11, которые можно читать независимо друг от друга

и в любом порядке. Давним пользователям гауссовских процессов, скорее всего, будут интересны последние две главы, в которых мы разрабатываем специализированные гауссовские процессы.

О коде

Исполняемые фрагменты кода находятся в онлайн-версии этой книги в liveBook по адресу: `https://livebook.manning.com/book/bayesian-optimization-in-action/`. Код доступен для загрузки с веб-сайта издательства (`https://www.manning.com/books/bayesian-optimization-in-action/`) и GitHub (`https://github.com/KrisNguyen135/bayesian-optimization-in-action/`). Последний будет включать ошибки и *пул-реквесты* (**Pull request**).

Для запуска кода вам понадобится **Jupyter Notebook**. Он предлагает удобный способ динамической работы с кодом, позволяя исследовать, как каждый объект ведет себя и взаимодействует с другими объектами. Дополнительную информацию, как начать работу с Jupyter Notebook, можно найти на их официальном сайте. Возможность динамически исследовать объекты особенно полезна в нашем случае, поскольку многие компоненты в рабочем процессе БО реализованы в виде объектов Python с помощью GPyTorch и BoTorch (основные библиотеки, которые мы будем использовать).

GPyTorch и BoTorch — лучшие инструменты для гауссовского моделирования процессов и байесовской оптимизации в Python. Существуют и другие варианты, такие как расширение scikit-optimize для библиотеки scikit-Learn или GPflow и GPflowOpt для фреймворка TensorFlow. Однако комбинация GPyTorch и BoTorch образует наиболее полную и гибкую кодовую базу, включающую множество современных алгоритмов, которые получены в результате исследований. На собственном опыте использования ПО для байесовской оптимизации я обнаружил, что GPyTorch и BoTorch обеспечивают хороший баланс между удобством для новичков и предоставлением самых современных методов.

Необходимо отметить еще одну вещь. Поскольку эти библиотеки имеют активную поддержку, API, показанные в книге, могут быть немного изменены в более новых версиях. Поэтому важно установить именно версии библиотек, указанные в файле `require.txt`, чтобы код запускался без ошибок. Дополнительные инструкции, как создавать среды Python, используя файл `require.txt`, можно найти, например, в официальной документации языка. При этом для работы с более новыми версиями вам, скорее всего, потребуется лишь внести незначительные изменения в код.

Читая книгу, вы также можете заметить, что текст имеет тенденцию сосредоточиваться только на ключевых компонентах кода, упуская из виду многие детали, такие как импорт библиотек и бухгалтерский учет. Разумеется, при первом появлении фрагмента они будут представлены в тексте корректно. Но именно краткость наших рассуждений поможет сосредоточиться на том, что нового привносится в каждой главе, и избежать необходимости повторяться. С другой стороны, код в Jupyter Notebook является автономным, и каждый фрагмент можно запускать отдельно, без каких-либо изменений.

Дискуссионный форум liveBook

Покупка книги включает бесплатный доступ к liveBook — онлайн-платформе для чтения от издательства Manning. Используя эксклюзивные функции, вы можете добавлять комментарии ко всей книге или к определенным разделам и абзацам. Здесь также легко можно оставлять для себя заметки, задавать технические вопросы и отвечать на них, а также получать помощь от автора и других пользователей. Вы также можете узнать больше о форумах Manning и правилах поведения на их странице обсуждения.

Manning предоставляет читателям место для обсуждения между отдельными читателями, а также между читателями и автором. Это совсем не обязывает к какому-то определенному объему участия со стороны последнего, чей вклад в форум остается добровольным (и неоплачиваемым). Предлагаем вам самим попробовать задать автору несколько интересных вопросов, чтобы его интерес не угас! Форум и архив предыдущих обсуждений будут доступны на сайте издательства до тех пор, пока книга находится в печати.

Архив с иллюстрациями к книге

Для полноценного изучения материала иллюстрации этой книги наиболее наглядны в цветном, а не черно-белом исполнении. На страницах данного печатного издания иллюстрации монохромные, а архив с их цветными версиями выложен на сайте издательства. Его можно скачать по ссылке

https://files.alistbooks.kz/3015_Nguyen_Python.zip

Об авторе



Кван Нгуен — Python-программист и энтузиаст машинного обучения. Он интересуется задачами о принятии решений, связанных с неопределенностью. Кван является автором нескольких книг по программированию на Python и научным вычислениям. В настоящее время он работает над степенью доктора компьютерных наук в Вашингтонском университете в Сент-Луисе, где занимается исследованием байесовских методов машинного обучения.

О техническом редакторе

Технический редактор этой книги — Керри Койч. Он является автором и архитектором ПО с более чем тридцатилетним опытом по внедрению корпоративных приложений и решений информационной архитектуры. Керри — автор книги по распределенной обработке, а также множества более коротких технических публикаций и обладатель патента на инновационную технологию *оптического распознавания символов* (**Optical Character Recognition**, **ORC**). Он также является лауреатом медали Армии США «За достижения».

Об изображении на обложке

Рисунок на обложке книги подписан Polonnois (*лат.*) или «Поляк». Он взят из сборника Жака Грассе де Сен-Совера, опубликованного в 1797 году. Каждая иллюстрация прекрасно нарисована и раскрашена вручную.

В те времена просто по одежде было легко определить, где живут люди и какое они имеют занятие или общественное положение. Manning отмечает изобретательность и инициативность компьютерного бизнеса, создавая обложки книг, основанные на богатом разнообразии региональной культуры много веков назад, возвращая их к жизни с помощью фотографий из подобных коллекций.

1 Введение в байесовскую оптимизацию

Темы главы:

- Что мотивирует использовать байесовскую оптимизацию и как она работает?
- Реальные примеры задач с применением байесовской оптимизации.
- «Игрушечный» пример байесовской оптимизации в действии.

В глобальном смысле байесовская оптимизация (БО) — это метод оптимизации, применяемый к функции (или любому процессу, генерирующему выходные данные), которая представляет собой «черный ящик», и ее оценка требует больших затрат с точки зрения времени, денег или других ресурсов. Данный подход содержит множество важных задач, включая настройку гиперпараметров, которые будут затронуты чуть позже. Использование БО может ускорить процедуру поиска и помочь найти оптимум функции как можно быстрее.

Несмотря на то что БО вызывает интерес со стороны ML-сообщества, на практике она используется не так часто, как другие методы, связанные с машинным обучением. Но почему так происходит? Некоторые находят ответ в том, что байесовская оптимизация сложна в изучении: нужно разбираться в вычислениях и теории вероятностей, а также иметь некий опыт в ML, позволяющий использовать метод в приложениях. Цель книги — развеять представление, что БО сложно использовать, а также показать, что она более интуитивна и доступна, чем можно было бы подумать.

На протяжении всей книги вы будете встречать множество иллюстраций, сюжетов и кода, которые призваны сделать эту тему более простой и конкретной. Вы узнаете, как работает каждый компонент байесовской оптимизации на высоком уровне и как реализовать их с помощью современных библиотек Python. Кроме того, примеры помогут вам приступить к реализации собственных проектов, поскольку основа подхода является универсальной и работает по принципу «подключи и играй». В этом отношении также полезны упражнения.

В общем, я надеюсь, что книга окажется полезной для ваших потребностей в машинном обучении и в целом будет интересна для чтения. Прежде чем углубиться в содержание, уделим некоторое время описанию проблемы, которую БО призвана решить.

1.1. Нахождение оптимума дорогостоящей функции «черного ящика»

Настройка гиперпараметров в машинном обучении является одним из наиболее распространенных применений байесовской оптимизации. В этом разделе будет исследована данная задача (а также несколько других) как пример общей задачи оптимизации «черного ящика». Это поможет понять, зачем нужна БО.

1.1.1. Настройка гиперпараметров как пример дорогостоящей задачи оптимизации «черного ящика»

Допустим, вы хотите обучить нейросеть на большом наборе данных, но не уверены, сколько слоев она должна иметь. Ее архитектура является решающим фактором при *глубоком обучении* (**Deep Learning**, **DL**), поэтому необходимо провести некоторое первоначальное тестирование (результаты представлены в табл. 1.1).

Таблица 1.1. Пример задачи по настройке гиперпараметров

Число слоев	Точность на тестовом наборе
5	0.72
10	0.81
20	0.75

Наша задача — определить, сколько слоев должна иметь нейросеть, чтобы быть наиболее точной. Есть некоторая сложность при выборе числа слоев, которое следует попробовать. Наилучшая обнаруженная точность согласно таблице — 81%. Это хороший результат, но есть возможность добиться большего с другим количеством. Предположим, руководство установило крайний срок, когда нужно завершить внедрение модели. Поскольку обучение нейросети занимает несколько дней, в нашем распоряжении есть некоторое конечное число испытаний, прежде чем нужно будет решить, сколько слоев должна иметь сеть. Имея это в виду, нам необходимо знать, какие еще значения следует попробовать следующими, чтобы найти количество слоев, обеспечивающее максимально возможную точность.

Подобная задача, где нужно найти наилучшие настройки (значения гиперпараметров) модели с целью оптимизировать показатели производительности (например, точность прогнозирования), обычно называется *настройкой гиперпараметров* (**Hyperparameter tuning**). В данном примере гиперпараметром нейросети является ее *глубина* (количество слоев). Если мы работаем с деревом решений, общими

гиперпараметрами будут максимальная глубина, минимальное количество точек на узел и критерий разделения. Используя *машину опорных векторов* (**Support Vector Machine**, **SVM**), можно было бы настроить *регуляризацию* и *ядро*. Поскольку производительность модели во многом зависит от ее гиперпараметров, их настройка является важным компонентом любого ML-конвейера.

Если бы это был типичный набор реальных данных, процесс мог бы занять много времени и ресурсов. Рис. 1.1 от OpenAI показывает, что по мере того, как нейросети становятся больше и глубже, объем необходимых вычислений (измеряется в петафлопс-днях) увеличивается экспоненциально.

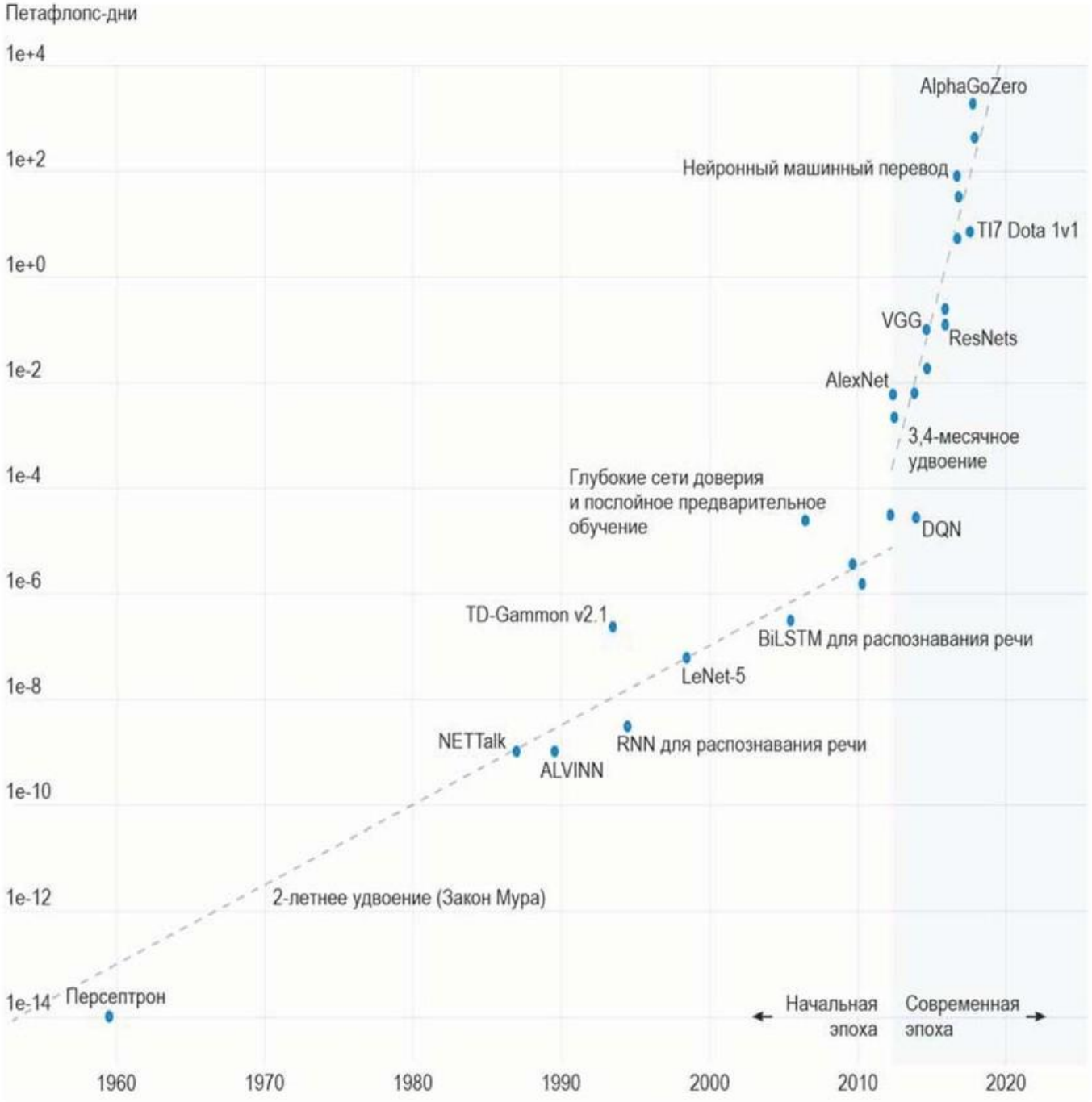


Рис. 1.1. График, демонстрирующий затруднения при настройке гиперпараметров из-за неуклонного роста вычислительных затрат на обучение больших нейросетей

Это означает, что обучение модели на большом наборе данных является весьма сложным и требует больших усилий. Кроме того, необходимо определить значения гиперпараметров, которые дают наибольшую точность. Поэтому обучение придется проводить множество раз. Основной вопрос при настройке гиперпараметров: как следует выбирать значения, которые стоит использовать для параметризации нашей модели, чтобы можно было как можно быстрее найти наилучшую комбинацию?

Вернемся к примеру. Какое количество слоев следует попробовать дальше, чтобы достигнуть точности выше 81%? Некоторое значение между 10 и 20 кажется многообещающим, поскольку в обоих случаях производительность выше, чем при 5 слоях. Но какое именно значение стоит проверить следующим? Ответ пока не очевиден, т. к. точность результата для числа слоев в диапазоне от 10 до 20 может быть очень изменчивой. Когда мы говорим об *изменчивости* (variability), мы неявно имеем в виду нашу *неопределенность* относительно того, каким образом точность теста для модели ведет себя в зависимости от количества слоев. Другими словами, ее поведение можно описать как некую функцию. Несмотря на то что 10 слоев приводят к 81%, а 20 — к 75%, нельзя с уверенностью сказать, какое значение дадут, например, 15 слоев. Это означает, что при рассмотрении значений в диапазоне от 10 до 20 необходимо учитывать уровень неопределенности.

Что если максимально возможную точность может дать какое-то число слоев больше 20? Так обстоит дело со многими большими наборами данных, где нейросеть должна быть достаточно глубокой, чтобы дать хоть какой-то полезный результат. Или (маловероятно, но все же) нам действительно нужно небольшое количество слоев (меньше 5)?

Каким способом следует изучить эти различные варианты, чтобы к моменту отчета перед начальством мы могли быть достаточно уверены, что достигли наилучшего количества слоев для нашей модели? Этот вопрос является примером дорогостоящей задачи оптимизации «черного ящика», которая будет обсуждаться далее.

1.1.2. Проблема дорогостоящей оптимизации «черного ящика»

В этом разделе мы познакомимся с данной проблемой, на решение которой и направлена байесовская оптимизация. Знание, почему она является настолько сложной, поможет понять, почему БО является более предпочтительной, чем простые и тривиальные подходы, вроде поиска по сетке (когда пространство поиска делится на равные сегменты) или случайного поиска (когда используются случайно сгенерированные точки для поиска).

В этой задаче доступа к функции нет, она представлена «черным ящиком» (Black box). Это некий механизм ввода-вывода. Цель — найти вход, который максимизирует выход. Такую функцию часто называют целевой, поскольку ее оптимизация является нашей целью. А именно необходимо найти ее оптимум — входные данные, которые дают наибольшее значение функции.

Характеристики целевой функции

Термин «черный ящик» означает, что нам неизвестно, как устроен объект (функция, программа, формула и пр.) изнутри. Все, к чему есть доступ, — это выходные данные после того, как объект каким-либо образом обработал информацию со входа. В нашем примере с нейросетью неизвестно, как изменится точность модели, если постепенно увеличить количество слоев (в противном случае мы бы просто выбрали наилучший).

Проблема является затратной, потому что во многих случаях наблюдение за целевым объектом (функцией) является непомерно дорогостоящим, что делает простые подходы, вроде исчерпывающего поиска, трудно реализуемыми. Как уже упоминалось, в машинном обучении (особенно в DL) время является основным ограничением.

Настройка гиперпараметров также относится к подобному классу дорогостоящих задач, но она не единственная! Любая процедура, в которой нужно найти некоторые параметры для оптимизации процесса, не зная, как различные настройки влияют и контролируют результат, квалифицируется как *задача оптимизации «черного ящика»*. Кроме того, опробование определенной настройки и наблюдение за ее результатом в целевом процессе (целевой функции, целевом объекте) требует много времени (или иных затрат в каком-либо другом смысле).

Определение

Процесс проверки определенной настройки (оценки значения целевой функции на каком-то входе) называется *выполнением запроса* или *запросом* целевой функции. Вся процедура представлена на рис. 1.2.



Рис. 1.2. Структура задачи оптимизации «черного ящика», где мы неоднократно *запрашиваем* значения функции в разных местах, чтобы найти глобальный оптимум

1.1.3. Другие реальные примеры дорогостоящих задач оптимизации «черного ящика»

Рассмотрим несколько примеров из реальной жизни, подпадающих под данную категорию задач. Они достаточно распространены в этой области. Допустим, вы столкнулись с функцией, которую надо оптимизировать, но запросы к ней можно сделать лишь небольшое количество раз. В таких случаях хорошо было бы найти способ разумно выбирать, в каком месте оценивать функцию.

Первый пример — открытие медицинских лекарственных средств. Это процесс, в ходе которого ученые идентифицируют соединения с желаемыми химическими свойствами для синтеза из них лекарств. Как вы понимаете, экспериментальный процесс весьма сложен и стоит больших денег. Еще одним фактором, усложняющим их открытие, является тенденция (активно наблюдается в последние годы), что продуктивность исследований и разработок в этой области снижается. Такое явление известно, как Eroom's Law (Закон Эрума) — перевернутый вариант Moore's Law (закон Мура). Он гласит, что количество новых лекарств, одобренных на каждый миллиард долларов США, сокращается вдвое в течение фиксированного периода времени. Этот закон представлен на рис. 1 статьи «Diagnosing the Decline in Parmaceutical R&D Efficiency» («Диагностика снижения эффективности фармацевтических исследований и разработок») в журнале Nature, авторами которой являются Джек В. Сканнелл, Алекс Блэнкли, Хелен Болдон и Брайан Уоррингтон.

Закон Эрума показывает, что в результате инвестиций очередного миллиарда долларов в медицинские исследования объем разработки лекарственных средств по логарифмической шкале уменьшается линейно с течением времени. Другими словами, в последние годы при фиксированной сумме инвестиций снижение объемов разработки лекарств носит экспоненциальный характер. Несмотря на то что в течение многих лет данная тенденция имеет взлеты и падения, экспоненциальный спад очевиден с 1950 по 2020 год.

Фактически такая проблема есть в любом научном исследовании, где ученые ищут новые химические вещества, материалы или разработки, являющиеся редкими, новыми и полезными с точки зрения некоторых показателей. Такие эксперименты требуют оборудования высочайшего класса и могут занимать дни или недели. Поэтому ученые пытаются оптимизировать определенные целевые функции, оценки которых чрезвычайно дороги.

Для наглядности в табл. 1.2 приведены несколько точек данных из реального набора, полученного для подобной задачи. Цель — найти состав сплава (из четырех исходных элементов) с самой низкой температурой смешивания. Это задача из разряда оптимизации «черного ящика». Здесь материаловеды работают с составами сплавов свинца (Pb), олова (Sn), германия (Ge) и марганца (Mn). Каждая комбинация процентного содержания веществ соответствует потенциальному сплаву, который можно синтезировать в лаборатории.

Таблица 1.2. Данные из задачи обнаружения материалов

% Pb	% Sn	% Ge	% Mn	Температура смешивания (°F)
0,50	0,50	0,00	0,00	192,08
0,33	0,33	0,33	0,00	258,30
0,00	0,50	0,50	0,00	187,24
0,00	0,33	0,33	0,33	188,54

Поскольку низкая температура смешивания указывает на стабильную и ценную структуру сплава, задача — найти составы с наиболее низкими температурами смешивания. Но есть одно «узкое место»: определение этого параметра для конкретного сплава обычно занимает несколько дней. Вопрос, который предстоит решить алгоритмически, аналогичен. Учитывая текущий набор данных, с каким следующим составом стоит экспериментировать (сколько каждого материала должно присутствовать в сплаве), чтобы найти минимальную температуру смешивания?

Другой пример — горнодобывающая промышленность и бурение нефтяных скважин. Если брать более конкретно, то речь идет о поиске на большой территории региона, который имеет самый высокий уровень ценных полезных ископаемых или нефти. Это требует обширного планирования, инвестиций и рабочей силы (опять же дорогостоящее мероприятие). Поскольку земляные работы оказывают существенное негативное воздействие на окружающую среду, есть правила, позволяющие сократить добычу полезных ископаемых и установить ограничения на количество оценок целевого объекта (функции), которые могут быть выполнены.

Главный вопрос в дорогостоящей оптимизации «черного ящика» заключается в следующем: каким образом лучше всего выбрать, в каком месте оценивать целевую функцию, чтобы в конце поиска можно было найти ее оптимум? Как показывает следующий пример, простые эвристики (вроде случайного поиска или поиска по сетке, реализованных в популярных пакетах Python, таких как scikitlearn) могут привести к избыточным тратам на вычисление целевой функции и, следовательно, к общей плохой оптимизационной производительности. Здесь в игру вступает байесовская оптимизация.

1.2. Введение в байесовскую оптимизацию

Рассмотрим БО как решение описанной выше проблемы. Это даст общее представление, что она собой представляет и как использует вероятностное машинное обучение для оптимизации дорогостоящих функций «черного ящика».

> **Определение**
>
> *Байесовская оптимизация* (**Bayesian optimization**) — это метод машинного обучения, который одновременно поддерживает прогностическую модель для изучения целевой функции и принимает решения, как получить новые данные, чтобы уточнить уже имеющиеся знания о ней, используя байесовскую теорию принятия решений.

Под *данными* подразумеваются пары «ввод-вывод», в каждой из которых значение на входе сопоставляется со значением на выходе. Они отличаются (при каждой конкретной настройке гиперпараметров) от данных для обучения ML-модели, которую мы настраиваем.

В *процедуре БО* решения принимаются на основе советов *алгоритма БО*. После того как действие (рекомендуемое БО) выполнено, модель БО обновляется на основе этого результата и начинает рекомендовать следующее действие. Процесс повторяется, пока мы не будем уверены, что выбрано оптимальное действие.

Есть два основных компонента рабочего процесса:

- *Модель машинного обучения* (или *ML-модель*), которая учится на основе наблюдений и делает прогнозы относительно значений целевых функций на неочевидных точках данных.
- *Политика оптимизации*, которая принимает решения, в каком месте функции провести следующее наблюдение, оценивая целевой объект для определения оптимального значения.

Рассмотрим каждый из этих компонентов подробнее.

1.2.1. Моделирование с помощью гауссовского процесса

БО работает, сначала подгоняя прогностическую ML-модель к целевой функции, которую мы пытаемся оптимизировать. Иногда ее называют *суррогатной моделью*, поскольку она действует как суррогат между тем, что, по нашему мнению, представляет собой функция на основе наблюдений, и самой функцией. Роль прогностической модели очень важна, поскольку ее предсказания влияют на решения, принимаемые алгоритмом БО, и, следовательно, напрямую сказываются на производительности оптимизации.

Почти всегда в роли прогностической модели используется *гауссовский процесс* (**Gaussian Process**, **GP**). На высоком уровне он, как и любая другая ML-модель, действует по принципу, когда схожие точки данных дают схожие прогнозы. ГП (гауссовский процесс), возможно, не самый популярный класс моделей по сравнению, например, с *гребневой регрессией* (**Grid Regression**), *деревьями решений* (**Decision Tree**), машинами опорных векторов или нейросетями. Однако, как можно наблюдать в этой книге, ГП обладает уникальной и важной особенностью — он не дает точечных прогнозов, как другие упомянутые модели. Вместо этого его предсказания имеют форму вероятностных распределений (или вероятностных прогнозов). Они являются ключевыми в БО, позволяя количественно оценивать неопределенность в прогнозах, что, в свою очередь, улучшает компромисс между риском и вознаграждением при принятии решений.

Рассмотрим сначала, как выглядит ГП при обучении на наборе данных. В качестве примера предположим, что мы заинтересованы обучить модель на наборе данных из табл. 1.3 (на рис. 1.3 обозначены черными крестиками).

Таблица 1.3. Пример регрессионного набора данных, соответствующий рис. 1.3

Точка обучающих данных	Метка
1.1470	1.8423
-4.0712	0.7354
0.9627	0.9627
1.2471	1.9859

Здесь модель гребневой регрессии подгоняется к имеющемуся набору данных и делает прогнозы в диапазоне от –5 до 5 (показаны в верхней части рис. 1.3). Модель гребневой регрессии — это модифицированная версия модели линейной регрессии, в которой регуляризованы веса, поэтому маленькие значения более предпочтительны во избежание переобучения. Каждый ее прогноз в конкретной контрольной точке представляет собой однозначное число, что совсем не отражает уровень неопределенности относительно того, как ведет себя исследуемая функция. Например, в контрольной точке, равной 2, модель просто предсказывает значение 2,2.

Вдаваться в подробности внутренней работы этой модели не нужно. Дело в том, что гребневый регрессор дает точечные оценки без измерения неопределенности, что также применимо ко многим ML-моделям, таким как машины опорных векторов, деревья решений и нейронные сети.

Как же тогда ГП делает свои прогнозы? Как показано на нижней части рис. 1.3, его предсказания имеют форму распределения вероятностей (в частности, нормального

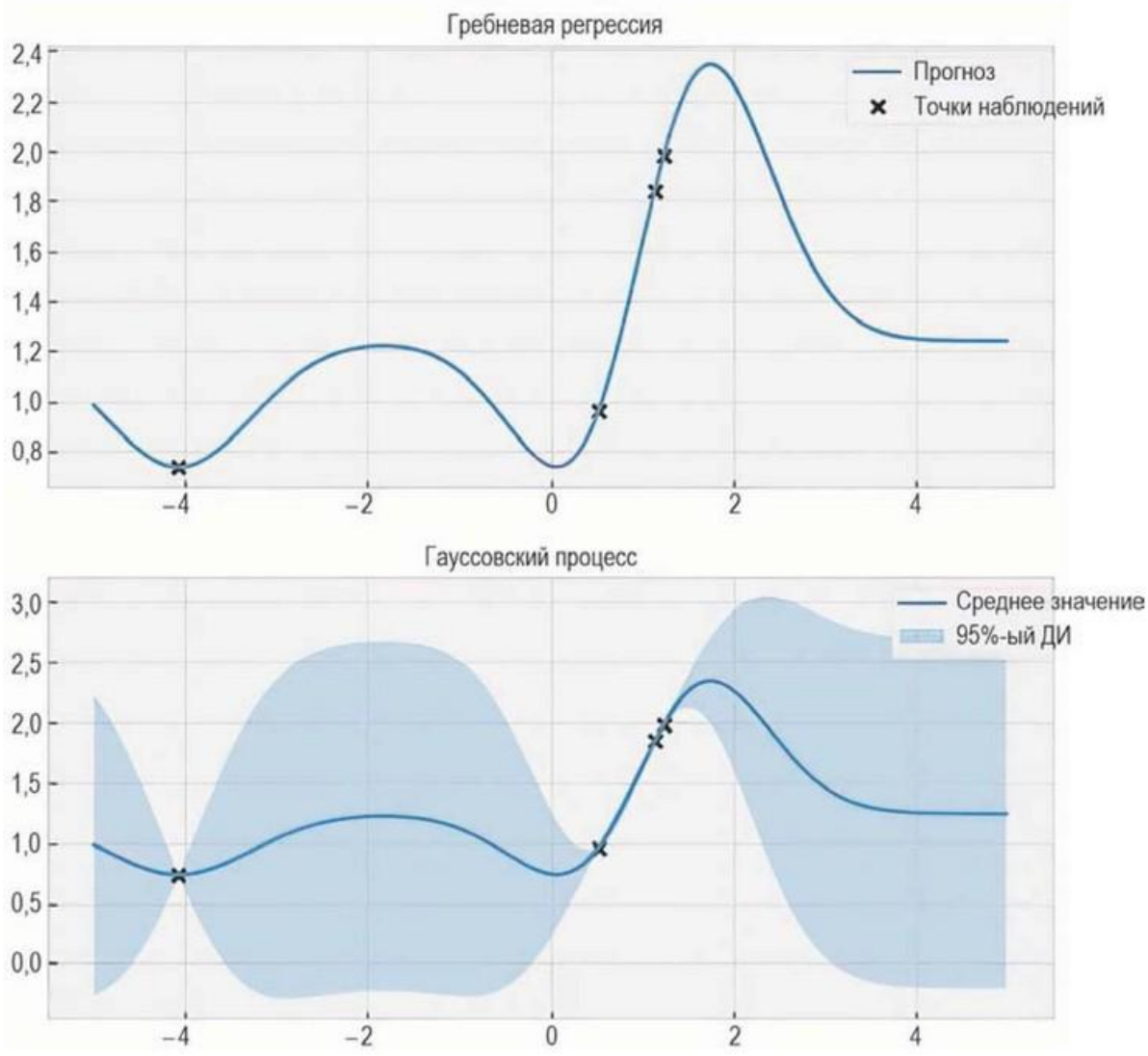


Рис. 1.3. Не-байесовские модели, такие как гребневые регрессоры, дают точечные оценки, в то время как гауссовские процессы создают распределения вероятностей в качестве прогнозов: таким образом, гауссовские процессы предлагают калиброванную количественную оценку неопределенности, которая является важным фактором при принятии решений с высоким риском

распределения). Это означает, что в каждой контрольной точке есть средний прогноз (сплошная линия), а также так называемый 95%-ный *достоверный интервал* (**credible interval**), который показан заштрихованной областью.

Обратите внимание, аббревиатура ДИ часто используется для сокращения *доверительного интервала* (**Confidence interval**) в *частотной статистике*. В этой книге данное сокращение используется исключительно для обозначения достоверного (вероятностного) интервала. Можно говорить о технических различиях между этими двумя концепциями, но на высоком уровне ДИ по-прежнему можно представлять как диапазон, в котором вполне вероятно, что интересующая величина (в данном случае истинное значение прогнозируемой функции) снижается.

Гауссовский процесс против гребневой регрессии

Интересно, что при использовании одной и той же *ковариационной функции* (также известной как ядро) ГП и модель гребневой регрессии дают одни и те же прогнозы (прогнозы среднего значения для ГП), как показано на рис. 1.3 (более подробно ковариационные функции будут обсуждаться в главе 3). Это означает, что ГП получает все преимущества, которыми обладает модель гребневой регрессии, предлагая при этом дополнительные прогнозы ДИ.

Несмотря на то что среднее значение является наиболее вероятным, 95%-ный достоверный интервал показывает диапазон вероятных значений в том смысле, что он составляет 95% вероятностной массы распределения.

По сути, ДИ измеряет уровень нашей неопределенности относительно значения в каждом месте наблюдения функции. Если имеет место большой прогностический ДИ (например, –2 или 4, как на рис. 1.3), то для него возможен более широкий диапазон значений. Другими словами, есть большая неопределенность относительно этого значения. Если имеет место «узкий» ДИ (0 или 2, как на рис. 1.3), тогда уверенность в значении этого местоположения выше. Приятной особенностью ГП является то, что для каждой точки обучающих данных прогностический ДИ близок к 0, что указывает на отсутствие какой-либо неопределенности относительно его значения. Это имеет смысл. В конце концов, мы уже знаем, что это за значение из обучающего набора.

Оценка шумных функций

Несмотря на то что на рис. 1.3 все в порядке, вполне возможно, что во многих реальных ситуациях процесс наблюдения за данными может быть искажен шумом (точки данных в наборе содержат шум). В этих случаях можно дополнительно указать его уровень с помощью ГП, тогда ДИ в наблюдаемых точках не снизится до 0, а вместо этого достигнет заданного уровня. Это демонстрирует гибкость моделирования с использованием гауссовских процессов.

Возможность присваивать числовое значение уровню неопределенности, которое носит название *количественной оценки неопределенности*, весьма полезна в любой процедуре принятия решений с высоким уровнем риска (наподобие БО). Вспомните сценарий из раздела 1.1 с настройкой количества слоев в нейросети, где есть ограничения по времени и возможность опробовать дополнительно лишь еще одну модель. Допустим, после обучения на точках данных ГП даст прогноз, что 25 слоев

дадут среднюю точность 0,85, а соответствующий 95%-ный ДИ составит от 0,81 до 0,89. С другой стороны, при 15 слоях ГП предположительно даст прогноз, что точность также будет в среднем 0,85, но 95%-ный ДИ составит от 0,84 до 0,86. Здесь вполне разумно отдать предпочтение 15 слоям, хотя оба варианта имеют одинаковое ожидаемое значение. Это потому, что мы более уверены в том, что именно 15 слоев принесут хороший результат.

Внесем ясность. Гауссовский процесс не принимает за нас никаких решений, но он предлагает средства сделать это с помощью своих вероятностных предсказаний. Принятие решений выполняет вторая часть структуры БО — *политика*.

1.2.2. Принятие решений с помощью политики БО

Помимо ГП, как прогнозной модели, нам также нужна процедура принятия решений. Это второй компонент БО, который учитывает прогнозы, сделанные моделью ГП, и определяет, как лучше всего оценить целевую функцию для эффективного нахождения оптимума.

Как упоминалось ранее, прогноз с 95%-ным ДИ от 0,84 до 0,86 считается лучшим, чем прогноз с ДИ от 0,81 до 0,89, особенно если есть только одна попытка. Это объясняется тем, что первый вариант более надежен и гарантированно принесет хороший результат. Но как следует принимать решение в более общем случае, когда две точки могут иметь разные прогностические средства и уровни неопределенности?

Именно это и помогает делать политика БО — количественно оценивать полезность точки с учетом ее прогнозируемого распределения вероятностей. Задача политики — взять модель ГП, которая отражает представление о целевой функции, и присвоить каждой точке данных оценку, указывающую, насколько та полезна при определении глобального оптимума. Такую оценку называют *оценкой приобретения* (**Acquisition score**). Наша задача состоит в выборе точки, которая максимизирует этот показатель, и в оценке целевой функции в этой точке.

На рис. 1.4 изображен тот же ГП, что и на 1.3. На нижней части показан график, как политика под названием «Ожидаемое улучшение» оценивает каждую точку на оси X в диапазоне между –5 и 5 (это наше пространство поиска). Что это означает и как политика оценивает точки данных, мы узнаем в главе 4. А пока просто запомним, что если точка имеет большую оценку приобретения, то она имеет высокую ценность для определения глобального оптимума.

На рис. 1.4 наилучшая точка находится возле значения 1,8. Это имеет смысл, поскольку согласно нашему ГП (верхняя часть рисунка) именно здесь будет достигнуто самое высокое *прогностическое среднее значение*. Это означает, что следующей мы выберем именно данную точку (1,8), чтобы оценить нашу цель, надеясь на улучшение по сравнению с самым высоким значением, которое мы собрали.

Следует отметить, что это не разовая процедура, а целый *обучающий цикл*. На каждой итерации ГП обучается на данных, которые получены из целевой функции. Затем запускается политика для этого ГП с целью получить рекомендацию, которая

должна помочь определить глобальный оптимум. После проводится наблюдение в рекомендованном месте, и новая точка добавляется к обучающим данным. Вся процедура повторяется, пока не будет достигнуто некоторое условие для завершения. Возможно, все выглядит немного запутанным, поэтому сделаем шаг назад и взглянем на БО более глобально.

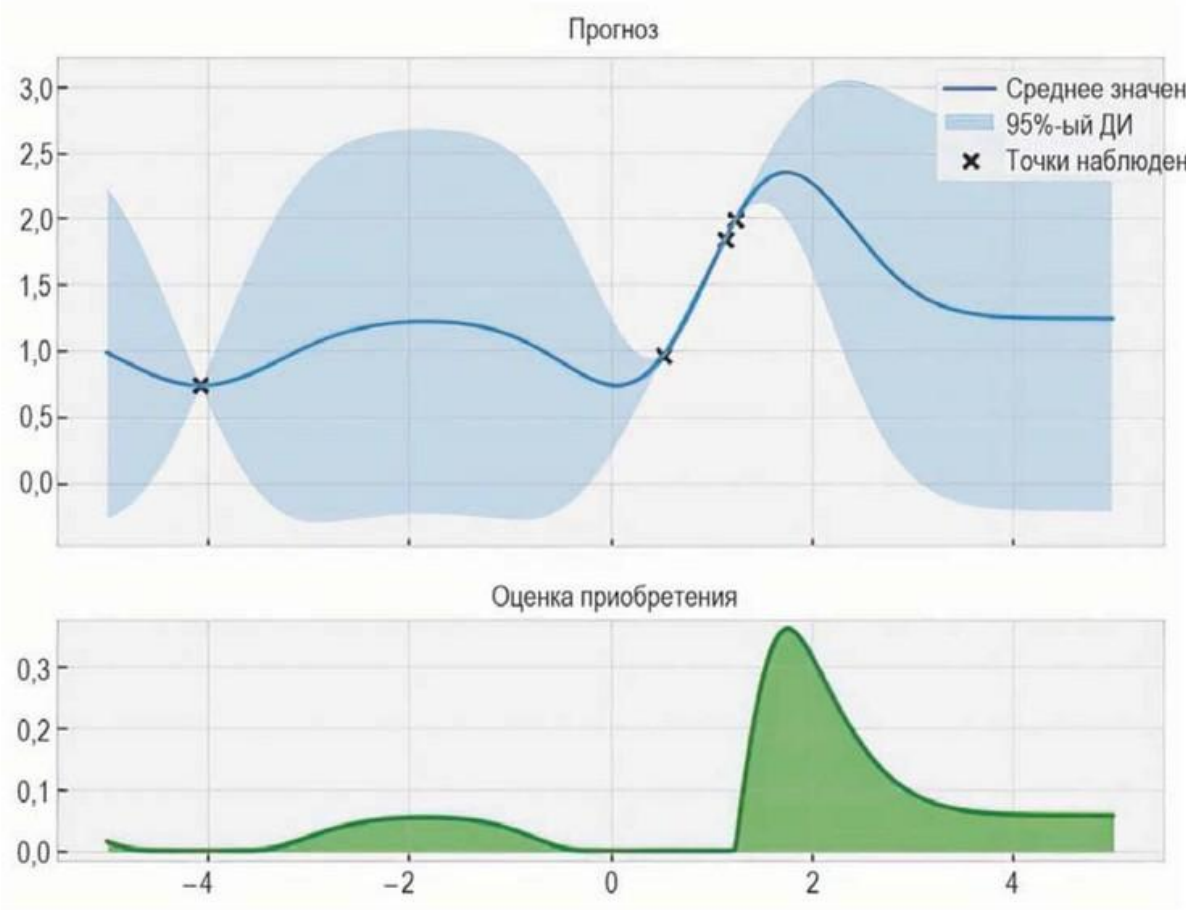


Рис. 1.4. Политика оценивает каждую отдельную точку данных по ее полезности для определения глобального оптимума: предпочтение отдается высокой прогностической ценности (где выигрыш более вероятен), а также высокой неопределенности (когда выигрыш потенциально может быть большим)

Связь с планированием экспериментов

Описание БО на этом этапе может напомнить вам концепцию *планирования экспериментов* (**Design of Experiments**, **DoE**) в статистике, задачей которой является решение той же проблемы оптимизации целевой функции путем настройки управляемых параметров. Между этими двумя методами существует много общего, но БО можно рассматривать как более общий подход, чем DoE.

1.2.3. Объединение ГП и политики БО для формирования цикла оптимизации

В этом разделе мы свяжем воедино все, что описывали до этого момента, и внесем в процедуру больше конкретики. Также рассмотрим рабочий процесс БО в целом, чтобы лучше понять, как различные компоненты работают друг с другом.

Начнем с исходного набора данных, похожий на те, что в табл. 1.1–1.3. Затем представим рабочий процесс БО на рис. 1.5, который вкратце можно обобщить следующим образом:

1. Модель ГП обучается на наборе. Это дает представление, как целевая функция выглядит в целом на основе того, что наблюдается из данных обучения. Это убеждение представлено сплошной кривой и заштрихованной областью (рис. 1.3 и 1.4).
2. Затем политика БО учитывает этот ГП и оценивает каждую точку в области в плане того, насколько она является ценной при поиске глобального оптимума. На это указывает нижняя кривая на рис. 1.4.
3. Точка, максимизирующая оценку, — это точка, которая будет выбрана следующей для запроса, а затем добавлена в набор обучающих данных.
4. Процесс повторяется, пока есть возможность продолжать оценивать цель.

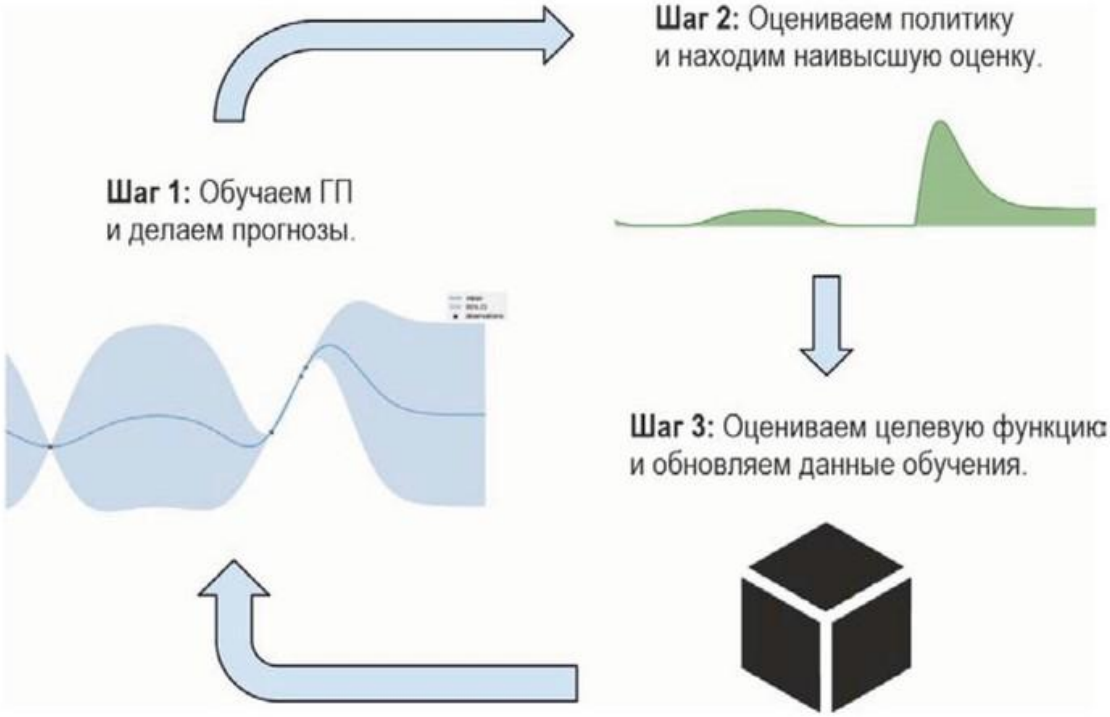


Рис. 1.5. Цикл БО, сочетающий в себе ГП для моделирования и политику принятия решений: этот полный рабочий процесс теперь можно использовать для оптимизации функций «черного ящика»

При обучении с учителем прогностическая модель просто подгоняется к набору обучающих данных и прогнозы выполняются по тестовому набору (который включает только шаги 1 и 2 на рис. 1.5). В отличие от такого подхода, рабочий процесс БО относится к активному обучению. Это подобласть машинного обучения, в которой можно решать, на каких точках данных учится модель. Такой процесс принятия решений, в свою очередь, зависит от самой модели.

Как уже обсуждалось, ГП и политика являются двумя основными компонентами процедуры БО. С одной стороны, если первый компонент не моделирует цель должным образом, то политику не получится снабдить качественными данными. А с другой, если второй компонент не позволяет назначать высокую оценку «хо-

рошим» точкам, а низкую — «плохим» (где «хорошо» означает помощь в поиске глобального оптимума), тогда последующие решения будут ошибочными и, скорее всего, приведут к плохим результатам.

Другими словами, без хорошей прогностической модели (в данном случае ГП) нельзя делать хорошие прогнозы с калиброванной неопределенностью. А без политики, хотя и с хорошими прогнозами, нельзя принимать правильные решения.

Пример, который неоднократно будет рассматриваться в книге, — это прогноз погоды. Представьте себе сценарий, в котором вы пытаетесь решить, брать ли с собой зонт перед выходом из дома, и просматриваете приложение с прогнозом погоды на своем смартфоне.

Не нужно говорить, что предсказания должны быть точными и надежными, чтобы можно было с уверенностью принимать на их основе решения. Приложение, которое всегда предсказывает только солнечную погоду, просто не имеет смысла. Нужен разумный способ принимать решения на основе прогнозов. Никогда не брать с собой зонтик, независимо от вероятности дождливой погоды, — это плохая политика принятия решений. Могут возникнут проблемы, когда дождь все же пойдет. С другой стороны, постоянно брать с собой зонтик, даже при 100%-ной вероятности солнечной погоды, тоже не самый разумный выбор. Оптимальное решение — адаптивно выбирать, взять с собой зонт или нет, основываясь на прогнозе погоды.

Вся суть БО заключается именно в адаптивном принятии решений. Прогностическая модель и политика не работают по отдельности. Необходимо уделять внимание взаимодействию обоих компонентов. Вот почему две основные части книги, следующие за этой главой, посвящены моделированию с помощью ГП и принятию решений с использованием политик БО соответственно.

1.2.4. Байесовская оптимизация в действии

На этом этапе вы можете задаться вопросом, действительно ли весь этот тяжелый механизм работает? На самом ли деле он работает лучше, чем какая-то простая стратегия, например случайная выборка? Это можно выяснить, взглянув на демопример с простой функцией, что также будет хорошим способом перейти от абстрактного к конкретному и выяснить, что мы сможем сделать в будущих главах.

Допустим, целевая функция, которую необходимо оптимизировать (в данном случае максимизировать), представляет собой одномерную функцию, как на рис. 1.6, определенную на интервале от –5 до 5. Стоит оговориться, что данный случай предназначен только для справки. При оптимизации «черного ящика» форма целевого объекта (функции) фактически неизвестна. Видно только, что он имеет несколько локальных максимумов в районе –5 (примерно –2,4 и 1,5), но глобальный максимум находится справа и имеет значение примерно 4,3. Предположим также, что оценивать целевую функцию разрешено максимум 10 раз.

Прежде, чем рассмотреть, как БО решает данную проблему оптимизации, нужно сначала взглянуть на две базовые стратегии. Первая — это *случайный поиск*, при котором равномерно выбираются значения в диапазоне от –5 до 5. Какие бы точки

ни получились, именно в этих местах будет оцениваться целевой объект. Рис. 1.6 представляет собой результат одной из таких схем. Наибольшее значение находится примерно в точке $x = 4$ при значении $f(x) = 3{,}38$.

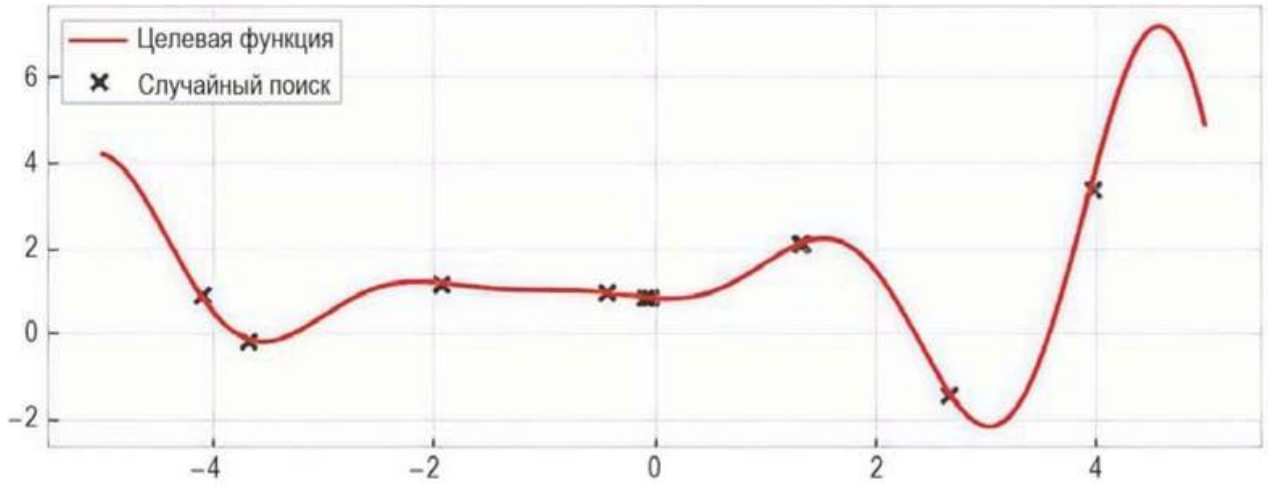


Рис. 1.6. Целевая функция, которую необходимо максимизировать, где случайный поиск тратит ресурсы на бесперспективные регионы

Как работает случайный поиск

Он предполагает равномерный и случайный выбор точек в пределах области целевой функции. Вероятность оказаться в какой-либо точке области равна вероятности оказаться в любой другой точке. Вместо равномерной выборки можно извлечь случайные выборки из неравномерного распределения, если предполагать, что в пространстве поиска есть важные области, которым следует уделить больше внимания. Однако подобная неравномерная стратегия требует знания, какие регионы важны, прежде чем начинать поиск.

Вас может смутить, что многие из точек попадают в область, близкую к 0, но это лишь случайность. Некоторые выборки действительно группируются именно вокруг 0, и при другом поиске они сгруппируются в другой области. Однако остается вероятность, что можно было бы потратить ценные ресурсы на проверку небольшой области, применяя множество оценок именно к ней. Вполне логично распределить их так, чтобы больше узнать о целевой функции.

Идея распределения оценок приводит ко второй базовой стратегии: поиску по сетке. Здесь пространство поиска делится на равномерно расположенные сегменты, и оцениваются конечные точки этих сегментов, как показано на рис. 1.7.

Лучшая точка для поиска — самая последняя точка справа (равная 5) с оценкой примерно 4,86. Этот способ лучше, чем случайный поиск, но нам по-прежнему нужен реальный глобальный оптимум.

Теперь можно посмотреть на БО в действии! Она начинается, как и при случайном поиске, со случайной выборки точки (рис. 1.8).

В верхней части рисунка представлен гауссовский процесс, обученный на оцениваемой точке, а в нижней — оценка, рассчитанная с помощью политики ожидаемого улучшения, которая говорит, насколько сильно следует оценивать каждое место

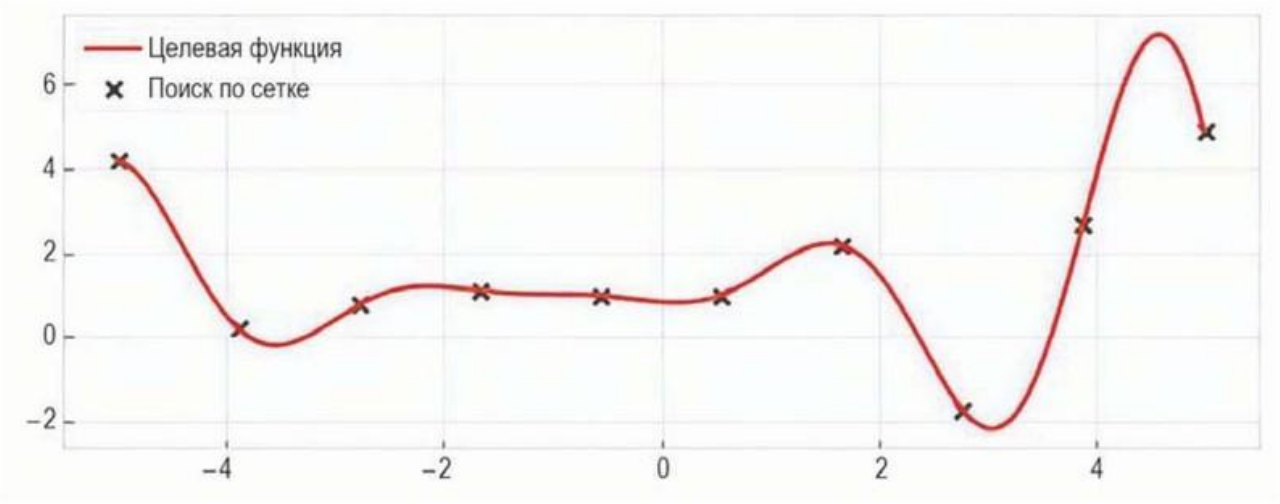


Рис. 1.7. Поиск по сетке по-прежнему неэффективен для сужения подходящего региона

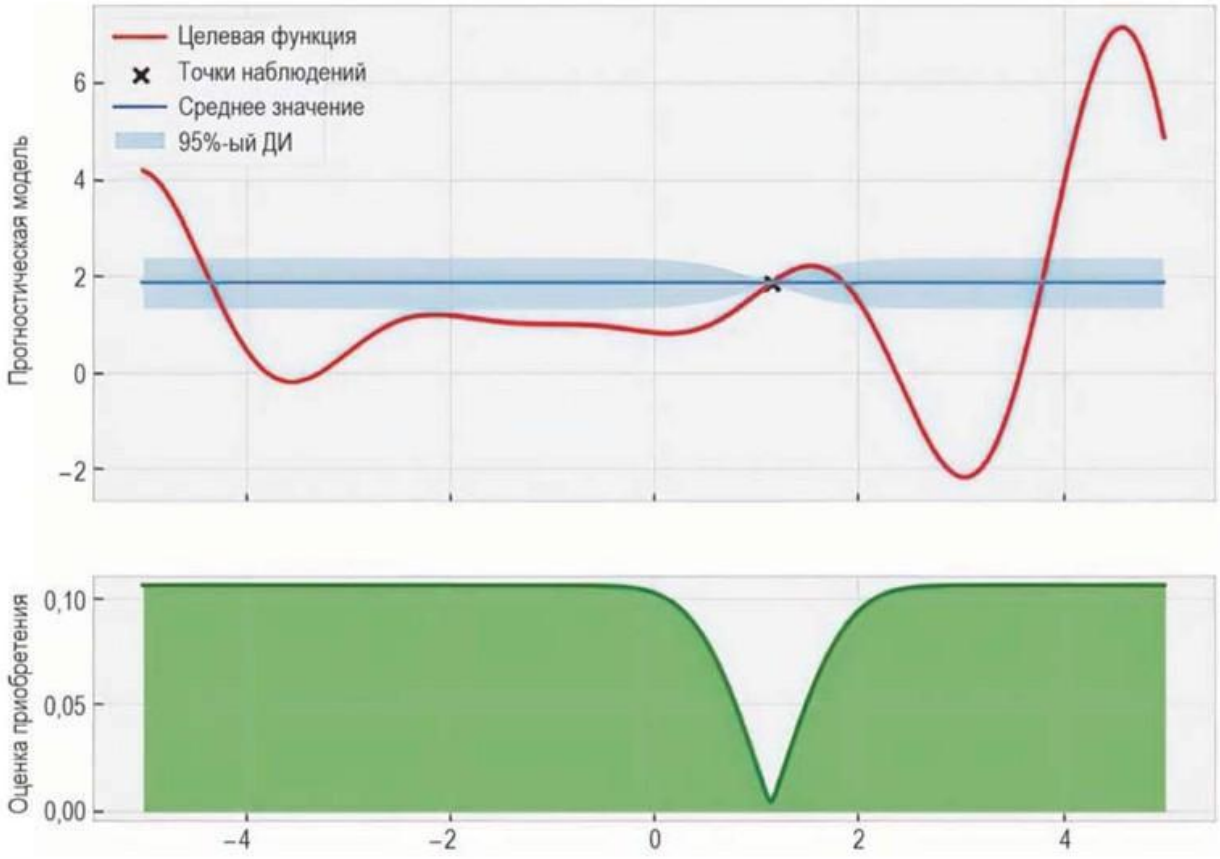


Рис. 1.8. Начало БО, аналогичное началу случайного поиска

в пространстве поиска. Выбирать следует то место, которое дает наивысшее значение при оценке следующей точки. Интересно следующее: политика на этом этапе говорит, что почти весь диапазон между –5 и 5, в котором выполняется поиск, является многообещающим (за исключением области возле 1, где сделан запрос). Это должно быть интуитивно понятным, поскольку мы видели только одну точку данных и еще не знаем, как остальная часть целевой функции выглядит в других областях. Политика говорит, что стоит исследовать больше! Далее посмотрим на состояние нашей модели (запросы с первого по четвертый на рис. 1.9).

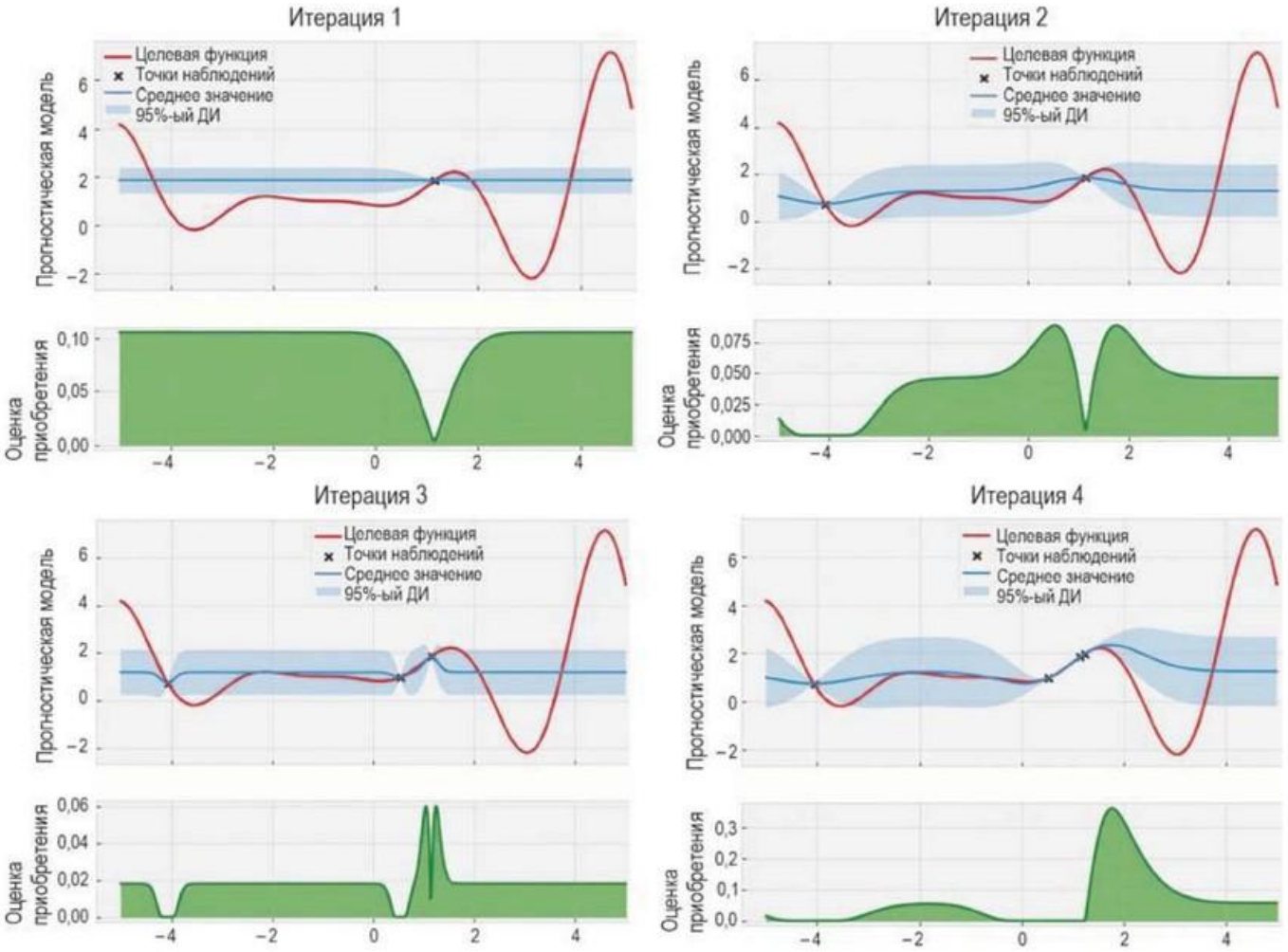


Рис. 1.9. После четырех запросов определен второй лучший оптимум

Три запроса из четырех сконцентрированы вокруг точки со значением 1, где существует локальный оптимум. Также видно, как политика предполагает, что следующая точка будет запрошена в этой же области. На данном этапе могут возникнуть опасения, что мы застрянем в этой локально оптимальной области и не сможем вырваться наружу, чтобы найти истинный (глобальный) оптимум. Но дальше вы убедитесь, что это не так. Перенесемся к следующим двум итерациям на рис. 1.10.

После пяти запросов на определение локально оптимального региона политика решила, что есть и другие, более перспективные регионы для изучения. А именно: тот, что слева около точки −2, и тот, что справа около точки 4. Это очень обнадеживает, поскольку становится ясно, что как только регион достаточно изучен, БО на нем не зацикливается. Теперь посмотрим, что происходит после восьми запросов (рис. 1.11).

Здесь можно заметить еще две точки справа, которые обновляют информацию как о модели ГП, так и о политике. Глядя на функцию среднего (сплошная линия, представляющая наиболее вероятный прогноз), видно, что она почти соответствует истинной целевой функции на диапазоне от 4 до 5. Кроме того, политика (нижняя кривая) сейчас очень близка только к глобальному оптимуму и больше ни к какой другой области. Это интересно, потому что область слева исследована не досконально (есть только одно наблюдение слева от 0). Но модель считает, что незави-

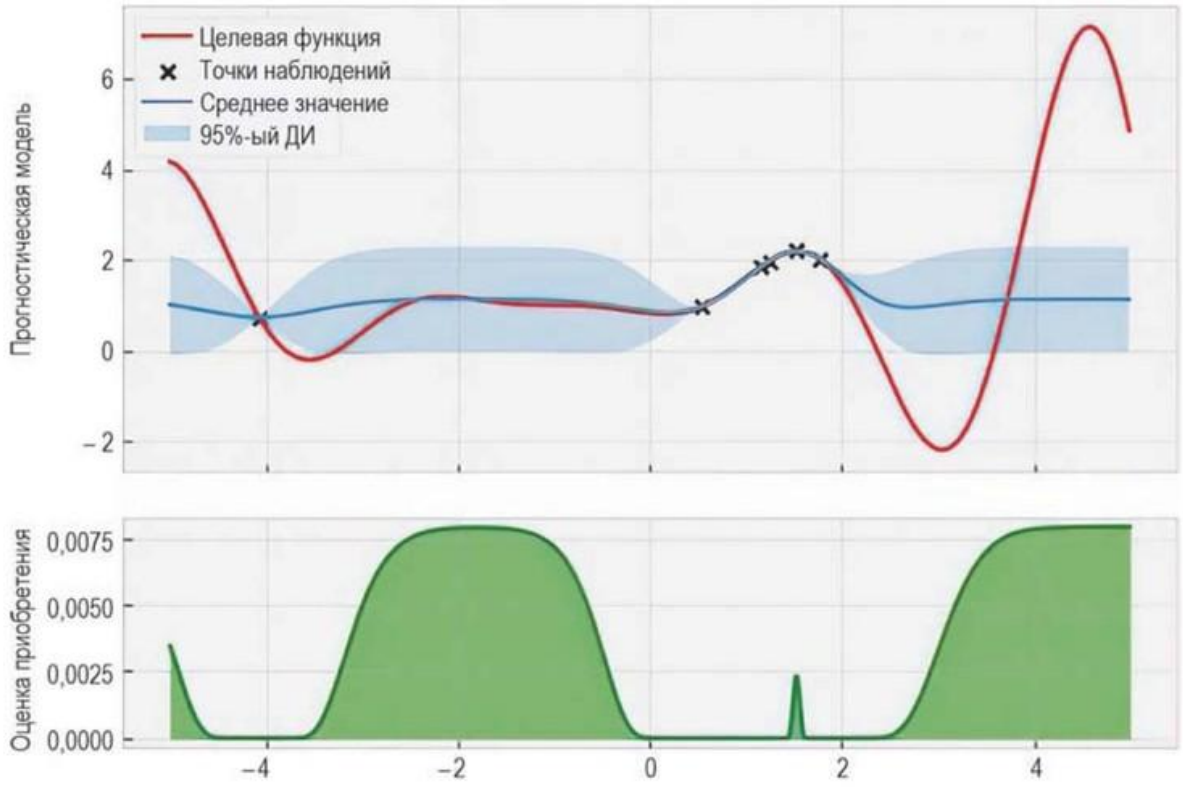


Рис. 1.10. После достаточного изучения локального оптимума политика предлагает рассмотреть другие области

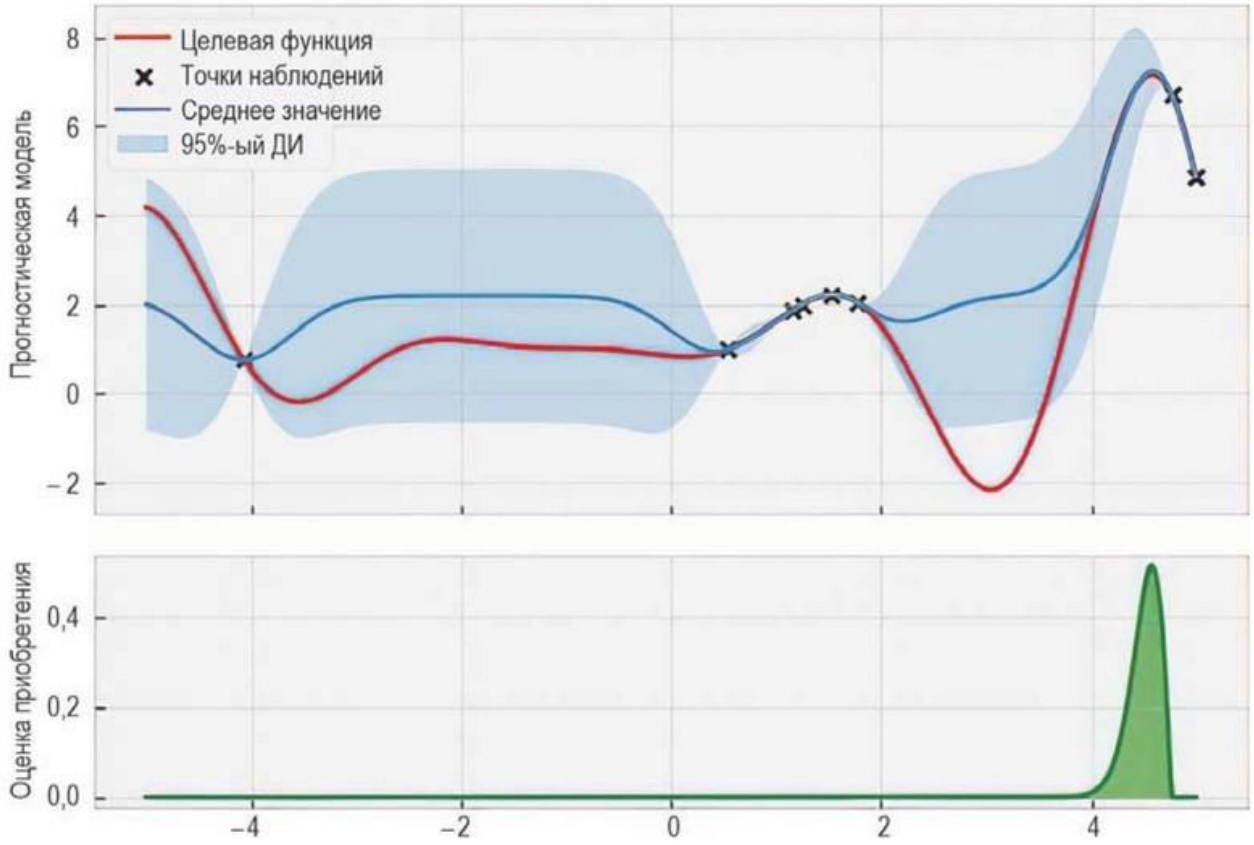


Рис. 1.11. БО успешно игнорирует большую область слева

симо от того, как функция выглядит в этой области, ее не стоит исследовать, в отличие от текущего региона. Это действительно так в данном случае.

В конце поиска по 10 запросам рабочий процесс показан на рис. 1.12. Сейчас нет никаких сомнений, что глобальный оптимум определен возле точки 4,3.

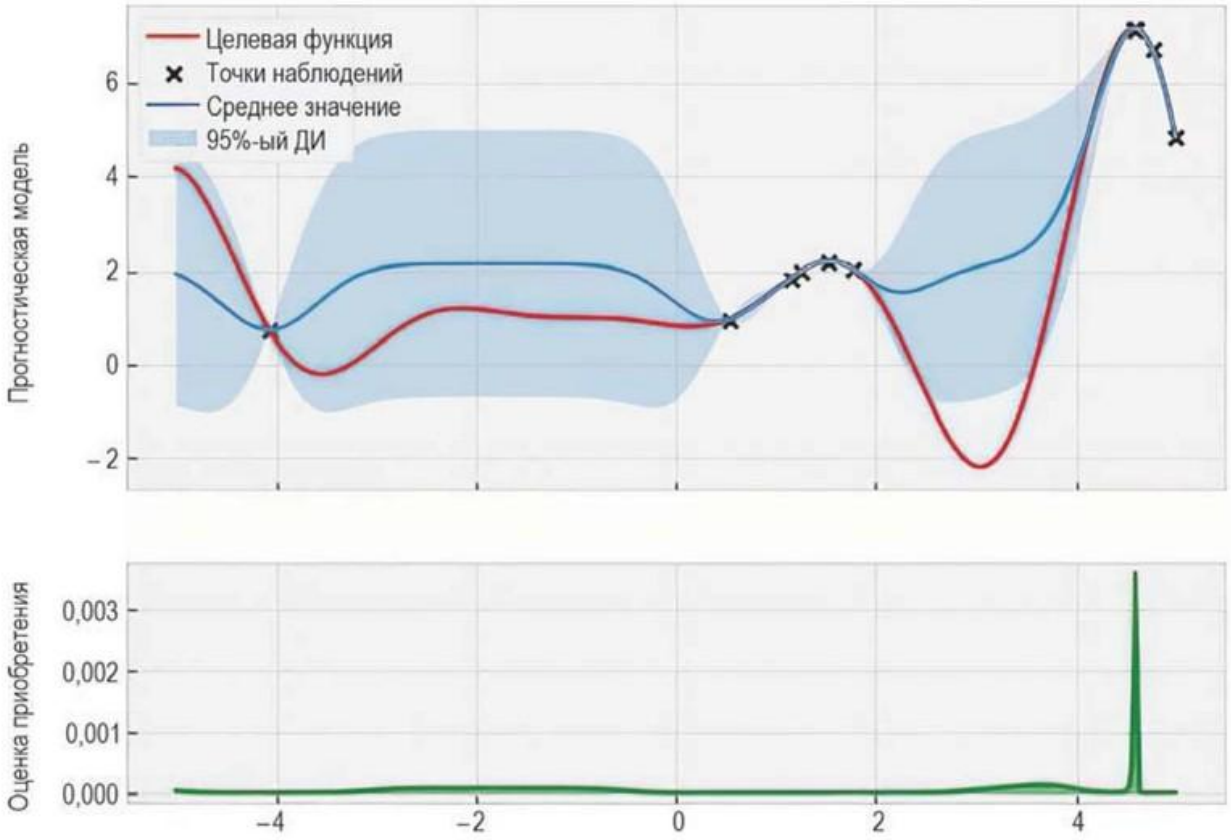


Рис. 1.12. В конце поиска БО нашла глобальный оптимум

Данный пример ясно показывает, что БО может работать намного лучше, чем случайный поиск и поиск по сетке. Это очень обнадеживает, учитывая, что последние две стратегии используются многими ML-специалистами, когда те сталкиваются с проблемой настройки гиперпараметров.

Библиотека scikit-learn является одной из самых популярных для машинного обучения в Python. Она предлагает модуль `model_selection` для различных задач при выборе модели. Однако случайный поиск и поиск по сетке — единственные методы настройки гиперпараметров, реализованные в нем. Другими словами, если гиперпараметры настраиваются с помощью случайного или сеточного поиска, у нас есть много возможностей добиться большего.

В целом использование БО может привести к значительному увеличению производительности оптимизации. Кратко рассмотрим несколько реальных примеров:

- В исследовательской работе 2020 года под названием «Bayesian Optimization is Superior to Random Search for Machine Learning Hyperparameter Tuning» («Байесовская оптимизация превосходит случайный поиск для настройки гиперпараметров машинного обучения»), ставшей результатом совместного исследования Facebook, Twitter, Intel и других компаний, было обнаружено, что БО оказалась чрезвычайно успешной во многих задачах настройки гиперпараметров.

- Фрэнсис Арнольд, лауреат Нобелевской премии 2018 года и профессор Калифорнийского технологического института, использует БО в своих исследованиях для поиска ферментов, эффективно катализирующих желаемые химические реакции.
- В исследовании под названием «Design of Efficient Molecular Organic Light-Emitting Diodes by a High-Throughput Virtual Screening and Experimental Approach» («Разработка эффективных молекулярных органических светоизлучающих диодов с помощью высокопроизводительного виртуального скрининга и экспериментального подхода»), опубликованном в журнале Nature, метод БО был применен к проблеме скрининга молекулярных органических светоизлучающих диодов (важного типа молекул), и обнаружено значительное повышение эффективности.

И таких примеров очень много.

Когда не стоит использовать БО

БО полезна, когда ограниченные ресурсы не позволяют многократно оценивать целевую функцию. Если исследования обходятся дешево, нет никаких причин подходить к ней экономно.

Если целевую функцию можно тщательно изучить с помощью поиска по сетке, этого будет достаточно для обнаружения глобального оптимума. В противном случае можно использовать другие стратегии (алгоритм DIRECT или эволюционные алгоритмы), которые часто показывают отличный результат в оптимизации, когда затраты на оценку довольно малы. Кроме того, если доступна информация о градиенте целевой функции, лучше подойдут алгоритмы на его основе.

Я надеюсь, что эта глава смогла разжечь интерес и вдохновить вас на то, что будет дальше. Мы будем суммировать ключевые навыки, которыми вы овладеете по ходу чтения всей книги.

1.3. Что вы узнаете в этой книге?

Эта книга даст глубокое понимание модели ГП и задачи БО. Вы узнаете, как реализовать конвейер БО на Python, используя самые современные инструменты и библиотеки. При решении данного типа задач вы также познакомитесь с широким спектром стратегий по моделированию и оптимизации. А к концу книги вы научитесь делать следующее:

- Реализовывать высокопроизводительные модели ГП, используя GPyTorch (лучший инструмент моделирования ГП на Python). Визуализировать и оценивать свои прогнозы. Выбирать подходящие параметры для моделей и внедрять такие расширения, как вариационные ГП и байесовские нейросети, для масштабирования решений до больших объемов данных.
- Внедрять широкий спектр политик БО, используя современную библиотеку BoTorch, которая прекрасно интегрируется с GPyTorch. А также проверять и понимать стратегии принятия решений.

- Применять различные специализированные настройки, такие как пакетная, ограниченная и многоцелевая оптимизации, используя платформы БО.
- Применять БО к реальным задачам, например к настройке гиперпараметров ML-модели.

Кроме того, в упражнениях будут использованы примеры и данные из реальной жизни с целью закрепить материал в каждой главе. На протяжении всей книги мы будем запускать алгоритмы на одном и том же наборе данных в самых разных условиях, чтобы иметь возможность сравнивать и анализировать различные использованные подходы.

Заключение

- Многие случаи в реальном мире можно рассматривать как дорогостоящие задачи оптимизации «черного ящика». В них можно увидеть только значения функции без какой-либо дополнительной информации. Кроме того, наблюдение за одним значением функции обходится дорого, что делает непригодными для использования многие алгоритмы оптимизации с учетом затрат.
- БО — это метод машинного обучения, который решает проблему оптимизации «черного ящика», используя интеллектуальные оценки целевой функции так, что оптимум может быть найден как можно быстрее.
- Внутри БО гауссовский процесс действует как прогностическая модель. Он предсказывает значение целевой функции в заданном месте и дает не только средний прогноз, но и 95%-ный ДИ (достоверный интервал), представляющий неопределенность через нормальное распределение.
- Для оптимизации функции «черного ящика» политика БО итеративно принимает решения, в каких точках следует оценивать целевую функцию. Политика делает это с помощью количественной оценки того, насколько полезна каждая точка данных для оптимизации.
- В БО оба компонента (ГП и политика) идут рука об руку. Первый делает хорошие прогнозы, а второй принимает правильные решения.
- Принимая решения адаптивным образом, БО справляется с оптимизацией лучше, чем случайный поиск или поиск по сетке, которые часто используются в качестве распространенных стратегий в задачах оптимизации «черного ящика».
- БО добилась значительных успехов в настройке гиперпараметров машинного обучения и других научных приложениях, например при разработке и открытии новых лекарств.

Часть I

Моделирование с помощью гауссовских процессов

Прогностическая модель играет решающую роль в байесовской оптимизации. Она направляет процесс принятия решений с помощью точных прогнозов. Как уже обсуждалось, ГП предлагает калиброванную количественную оценку неопределенности, которая является ключевым компонентом в любой задаче принятия решений. Она также может быть представлена функцией, которой нет во многих ML-моделях.

В главе 2 мы рассмотрим ГП как распределение по функциям в целом, а также как обобщение многомерного нормального распределения в бесконечных измерениях в частности. Мы исследуем, как с помощью теоремы Байеса можно обновить ГП, чтобы отразить представление о значении функции в свете новых данных.

В главе 3 мы убедимся в математической гибкости ГП, которая позволяет пользователям включать априорную информацию в прогнозы с помощью общей тенденции и вариативности гауссовского процесса. Комбинируя разные компоненты ГП, можно получить возможность математически моделировать широкий спектр функций.

Рассуждения в этой части сопровождаются примерами кода, которые реализованы с использованием современной Python-библиотеки — GPyTorch. С ее помощью вы получите практический опыт проектирования и обучения моделей.

2 Гауссовский процесс как распределение по функциям

Темы главы:

- Ускоренный курс по многомерным гауссовым распределениям и их свойствам.
- Понимание ГП как многомерного распределения Гаусса в бесконечных измерениях.
- Реализация ГП на Python.

Рабочий процесс БО состоит из двух основных частей: гауссовский процесс (как прогностическая модель) и политика принятия решений. С помощью первого мы получаем не только точечные оценки в качестве прогнозов для тестовых данных, но также вместо этого имеем полное распределение вероятностей, отражающее нашу уверенность в прогнозе.

ГП делает схожие предсказания на основе схожих точек данных. Например, прогнозируя погоду, когда оценивается сегодняшняя температура, он будет учитывать климатические данные либо дней, похожих на текущий, либо последних нескольких дней, либо именно этого дня год назад. Дни в другом сезоне не повлияют на ГП при составлении данного прогноза. Точно так же, определяя цену дома, ГП понимает, что похожие дома в том же районе более информативны, чем дома в другом штате.

Насколько одна точка данных похожа на другую, кодируется с помощью ковариационной функции, которая, помимо этого, моделирует неопределенность в предсказаниях. В главе 1 мы уже сравнивали модели гребневой регрессии и ГП (снова представлены на рис. 2.1). Несмотря на то что гребневый регрессор дает только однозначные прогнозы, ГП выводит нормальное распределение в каждой контрольной точке. Именно количественная оценка неопределенности отличает ГП от других ML-моделей, особенно в контексте принятия решений в условиях неопределенности.

Далее рассмотрим, как корреляционное моделирование и количественная оценка неопределенности математически реализуются с помощью гауссовских распреде-

лений, а также научимся использовать ГП в GPyTorch. Возможность моделировать функцию с помощью гауссовского процесса — первый шаг к байесовской оптимизации, который будет предпринят в этой главе.

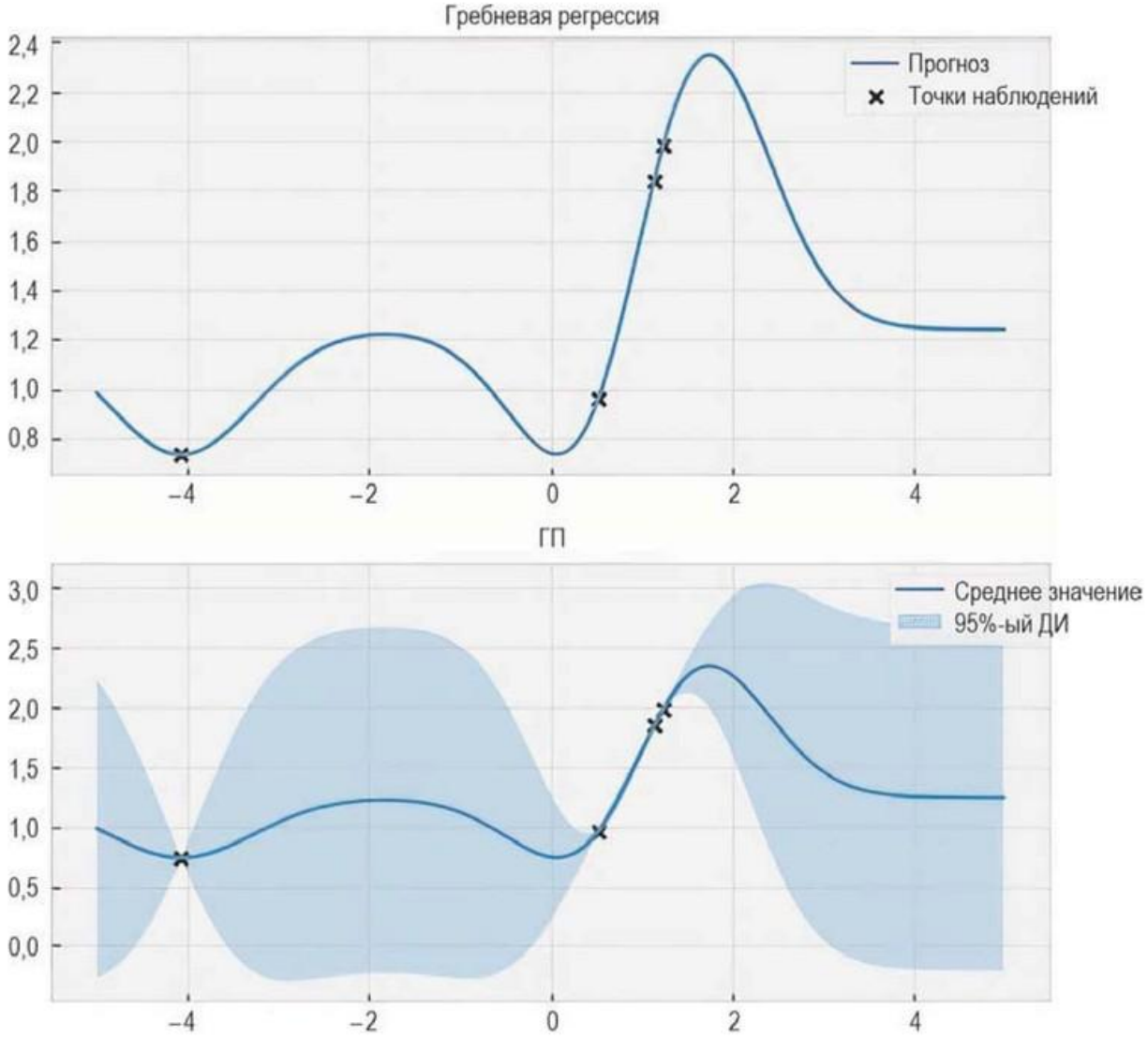


Рис. 2.1. Сравнение прогнозов гребневой регрессии и ГП: несмотря на то что среднее значение прогноза ГП аналогично прогнозу гребневой регрессии, он также предлагает ДИ, который указывает на уровень неопределенности (серая область)

Почему GPyTorch?

В Python есть и другие библиотеки моделирования ГП, такие как GPy или GPflow, но для этой книги мы выбрали именно GPyTorch. Он создан на основе PyTorch, имеет активную поддержку и предлагает оптимизированный рабочий процесс: от манипуляций с массивами до ГП-моделирования. В конечном итоге он позволит дойти до БО с использованием BoTorch, который мы начнем использовать в главе 4.

В этой библиотеке также реализовано множество современных методов. Например, глава 12 рассказывает об использовании GPyTorch для масштабирования ГП на большие наборы данных, а глава 13 научит интегрировать нейросеть в модель.

2.1. Как продать свой дом по методу Байеса

Прежде чем перейти сразу к ГП, рассмотрим пример в области моделирования цен на жилье, а именно, как определяется стоимость определенного дома по отношению к другим домам. Этот наглядный сценарий покажет, как работает корреляция в многомерном распределении Гаусса, которое является центральной частью ГП.

Допустим, вы владеете домом в Миссури и хотите его продать. Вы пытаетесь установить подходящую цену и обсуждаете с другом, как это сделать:

Вы: Я не знаю, как быть. Дело в том, что я не представляю, за сколько я могу продать свой дом.

Друг: Может, у тебя есть хоть какая-нибудь приблизительная цифра?

Вы: Думаю, где-то между 150 и 300 тысячами.

Друг: Это довольно большой диапазон.

Вы: Да, хотелось бы спросить людей, которые уже продавали свои дома. Нужны какие-нибудь примеры, от которых можно отталкиваться.

Друг: Я слышал, что Алис продала свой дом за 250 тысяч.

Вы: Ты имеешь в виду Аликс, которая в Калифорнии? Это прекрасно, но я не думаю, что ее дом поможет мне правильно оценить мой собственный.

Друг: Нет, я говорю про Алис, живет рядом с тобой.

Вы: О, понятно. Это очень полезная информация, ее дом действительно очень похож на мой! Тогда, я думаю, что он будет стоить где-то между 230 и 270 тысячами. Время поговорить с моим риелтором!

Друг: Рад, что смог помочь.

Использование дома Алис в качестве ориентира — хорошая стратегия для оценки стоимости вашего дома, поскольку они схожи по характеристикам и физически находятся рядом. Поэтому вполне ожидаемо, что они будут продаваться за примерно одинаковую цену. И наоборот, дом в Калифорнии совершенно не имеет отношения к вашему. Поэтому, даже если его цена известна, это не даст никакой полезной информации.

Проведенный только что расчет представляет собой *байесовское обновление* представлений о стоимости дома. Возможно, вы уже знакомы с теоремой Байеса (рис. 2.2). Хорошее введение в нее, а также в байесовское обучение можно найти в главе 8 книги Луиса Серрано *«Grokking Machine Learning»* (*«Машинное обучение Гроккинга»*, издательство Manning, 2021).

Теорема Байеса дает возможность обновить *представление* (предположение, убежденность) об интересующей нас величине (стоимость дома в данном случае). Применяя ее, мы переходим от нашего *априорного* (изначального) убеждения, которое является первым предположением, к *апостериорному* (итоговому, результирующему) убеждению о рассматриваемой величине, которое вытекает из первого после получения новой полезной информации. Это апостериорное убеждение сочетает



Рис. 2.2. Теорема Байеса, которая предоставляет метод для обновления убеждений об интересующей величине, представленной как вероятностное распределение случайной переменной: прежде чем наблюдать какие-либо данные, имеется априорное убеждение относительно величины X; после обновления данных мы получаем апостериорное убеждение об этой величине

в себе как предыдущее представление, так и вероятность любых наблюдаемых нами данных.

В нашем примере мы начинаем с априорного предположения, что цена находится в диапазоне от 150 до 300 тысяч, который довольно велик, поэтому в первоначальном убеждении содержится не так уж много информации. Далее мы обновляем диапазон и получаем апостериорное убеждение с учетом новой информации о цене дома Алис.

Во-первых, после предположения, что дом в Калифорнии может оцениваться в 250 тысяч, наше апостериорное мнение о собственном доме остается неизменным (по-прежнему от 150 до 300 тысяч). Все потому, что дом Аликс не имеет отношения к нашему, и его цена не дает полезной информации об интересующей нас величине.

Во-вторых, когда мы получаем новую информацию, что дом Алис находится рядом с нашим и оценивается в 250 тысяч, наше апостериорное убеждение значительно меняется по сравнению с предыдущим (диапазон стал 230–270 тысяч). Используя этот дом в качестве эталона, наше убеждение обновилось (250 тысяч), при этом диапазон предположения сузился (от разницы в 150 тысяч до разницы в 40 тысяч). Это имеет смысл, поскольку цена дома Алис очень информативна относительно цены нашего дома (рис. 2.3).

Обратите внимание, что числа в примере не являются достоверными. Они нужны только для интуитивного понимания ситуации. Используя многомерное распределение Гаусса для моделирования убеждений, можно реализовать данную процедуру обновления количественным способом. Кроме того, с помощью него также можно определить, достаточно ли одна величина (чей-то дом) похожа на ту, что интересует нас (собственный дом), а также как она влияет на апостериорное убеждение и в какой степени.

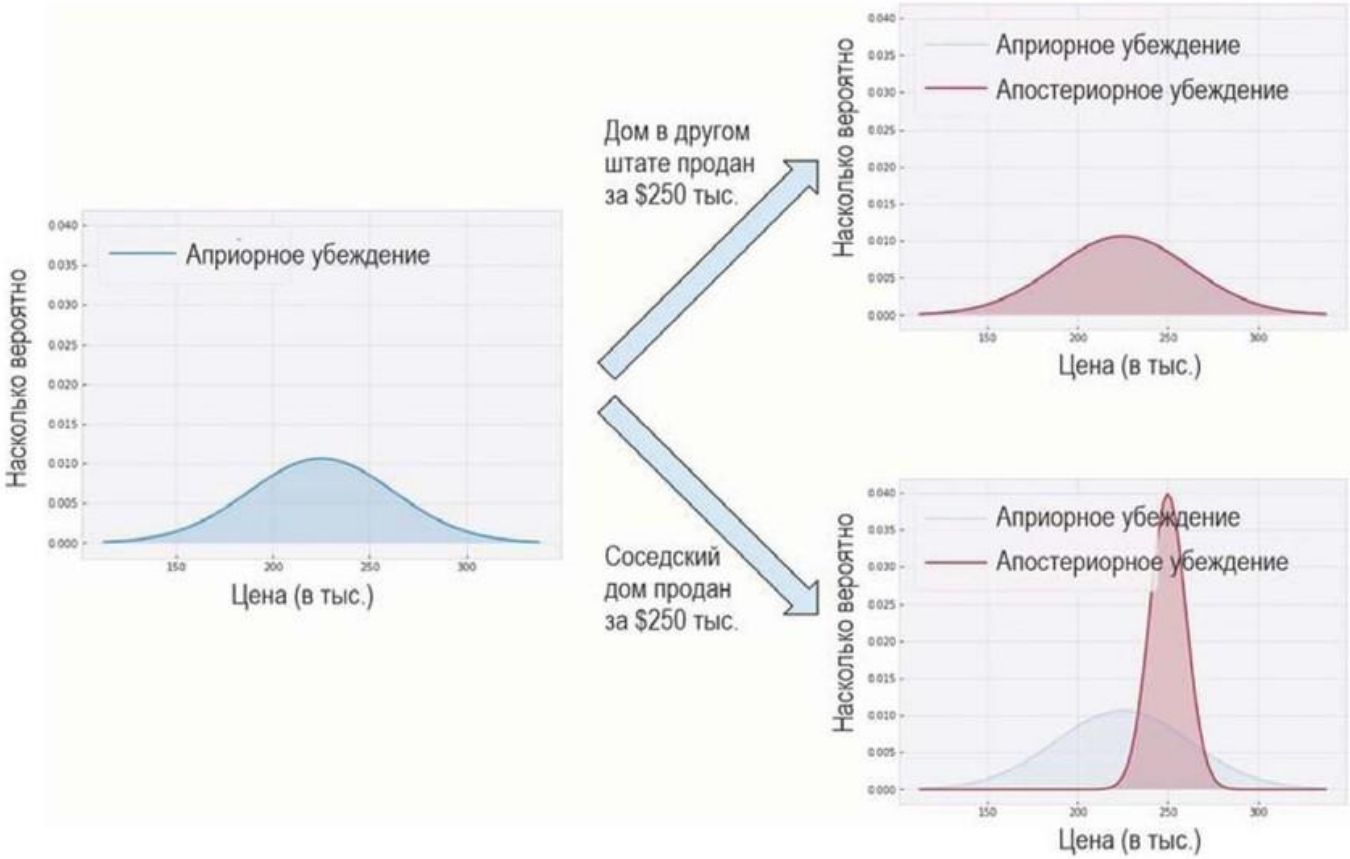


Рис. 2.3. Обновление убеждения о цене нашего дома по методу Байеса: в зависимости от того, насколько наблюдаемый дом похож на наш, апостериорное убеждение либо остается прежним, либо радикально меняется

2.2. Моделирование корреляций с помощью многомерных гауссовских распределений и байесовских обновлений

В этом разделе речь пойдет о *многомерных гауссовых распределениях* (коротко — *гауссианах*), также известных, как *многомерные нормальные распределения* (**Multivariate Normal Distribution**, **MVN**), и о том, как они облегчают выполнение обновления, которое было рассмотрено ранее. Это послужит основой для последующего обсуждения вопросов, связанных с ГП.

2.2.1. Использование многомерных распределений Гаусса для совместного моделирования нескольких переменных

Для начала рассмотрим, что такое многомерные гауссианы и что можно моделировать с их помощью. Также изучим, что, используя ковариационную матрицу, многомерная гауссиана может описать не только поведение отдельных случайных величин, но и их корреляцию.

Во-первых, стоит начать с понимания, что такое нормальное распределение (MVN), или *колоколообразная кривая* (**Bell curve**). Они широко распространены в реальном

мире и используются для моделирования большого диапазона величин, например рост, IQ, доход, вес при рождении и пр.

MVN стоит использовать, когда необходимо смоделировать более одной переменной (величины). Для этого их нужно объединить в *вектор случайных величин*. В этом случае говорят, что такой вектор соответствует нормальному распределению (рис. 2.4).

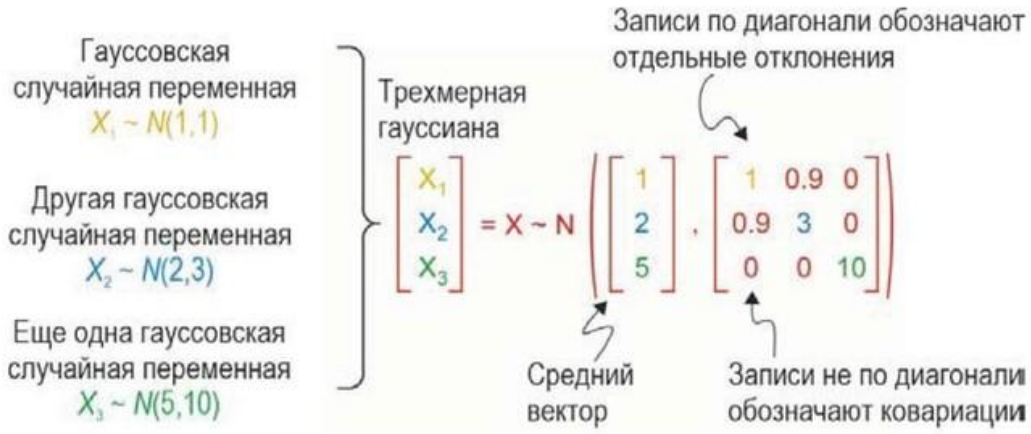


Рис. 2.4. MVN объединяет вместе несколько нормально распределенных случайных величин: пока его средний вектор объединяет средние значения, ковариационная матрица моделирует корреляции между отдельными переменными

Определение

Рассмотрим случайный вектор $X = [X_1 X_2 \ldots X_n]$, который подчиняется распределению Гаусса, обозначаемому $N(\mu, \Sigma)$, где μ — вектор длины n, а Σ — матрица размером $n \times n$. Здесь μ является средним вектором, отдельные элементы которого обозначают ожидаемые значения соответствующих случайных величин в X, а Σ — ковариационная матрица, которая описывает дисперсию отдельных переменных, а также корреляции между ними.

Важно уделить немного времени и разобрать определение нормального распределения:

- Благодаря удобным свойствам каждая случайная величина в векторе X подчиняется нормальному распределению. В частности, i-я переменная X_i имеет среднее значение μ_i, которое является i-м элементом среднего вектора μ в нормальном распределеним.
- Дисперсия X_i — это i-й диагональный элемент ковариационной матрицы Σ.
- Если есть вектор случайных величин, соответствующих нормальному распределению, то каждая отдельная переменная в нем также ему соответствует.

Если диагональные элементы ковариационной матрицы Σ являются дисперсиями отдельных переменных, чем тогда являются недиагональные элементы? Запись в i-й строке и j-м столбце матрицы обозначает ковариацию между X_i и X_j, которая связана с корреляцией между двумя случайными величинами. Если предположить, что корреляция положительная, то верно следующее:

- Если она высокая, то говорят, что две случайные величины X_i и X_j коррелируют. Это означает, когда значение одного увеличивается, значение другого также

имеет тенденцию к увеличению, а когда значение одного уменьшается, значение другого также будет уменьшаться. Цены на оба дома (ваш и вашей соседки Алис) являются примерами коррелирующих величин.

- С другой стороны, если она низкая и близка к нулю, то независимо от значения X_i значение X_j, скорее всего, сильно не изменится. Это связано с тем, что между двумя переменными нет корреляции. Дом Аликс в Калифорнии и ваш попадают в эту категорию.

Отрицательные корреляции

Корреляция также может быть отрицательной, указывая на то, что переменные движутся в противоположных направлениях. Если одна переменная увеличивается, другая — уменьшается, и наоборот. Именно положительные корреляции иллюстрируют важные концепции, которые обсуждаются в этой книге, поэтому случаи с отрицательной корреляцией нам не важны.

Внесем больше конкретики в обсуждение и определим MVN-распределение, которое совместно моделирует три случайные переменные:

- цену нашего дома (A);
- цену дома нашей соседки Алис (B);
- и цену дома Аликс в Калифорнии (C).

Эта трехмерная гауссиана также имеет ковариационную матрицу (рис. 2.4).

> **Примечание**
>
> Для удобства принято предполагать, что средний вектор гауссианы нормирован до *нулевого вектора*. На практике такая нормализация обычно выполняется для упрощения многих математических деталей.

Диагональные ячейки говорят о дисперсиях (отклонениях) отдельных случайных величин. Отклонение B (3) немного больше, чем A (1). Это означает, что значение B имеет большую неопределенность, чем A, поскольку нет достаточной информации о соседнем доме. Поэтому более точную оценку сделать невозможно. С другой стороны, C имеет наибольшую дисперсию. Это указывает на тот факт, что дома в Калифорнии в целом имеют более широкий ценовой диапазон.

> **Примечание**
>
> Используемые здесь значения (1, 3, 10) являются примерами. Они демонстрируют, что чем больше дисперсия случайной переменной, тем больше неопределенность в отношении ее значения (прежде чем мы его узнаем).

Кроме того, ковариация между нашим домом (A) и соседним (B) равна 0,9, что указывает на существенную корреляцию цен этих величин. Это имеет смысл, поскольку, зная цену соседнего дома, можно получить более точную оценку нашего. Также обратите внимание, что ни A, ни B не имеют никакой корреляции с ценой дома в Калифорнии, поскольку с точки зрения местоположения величина C не имеет с ними ничего общего. Другими словами, даже если известно, сколько стоит дом в Калифорнии, о цене собственного дома мы ничего не узнаем. Визуализируем нашу трехмерную гауссиану, используя график параллельных координат (рис. 2.5).

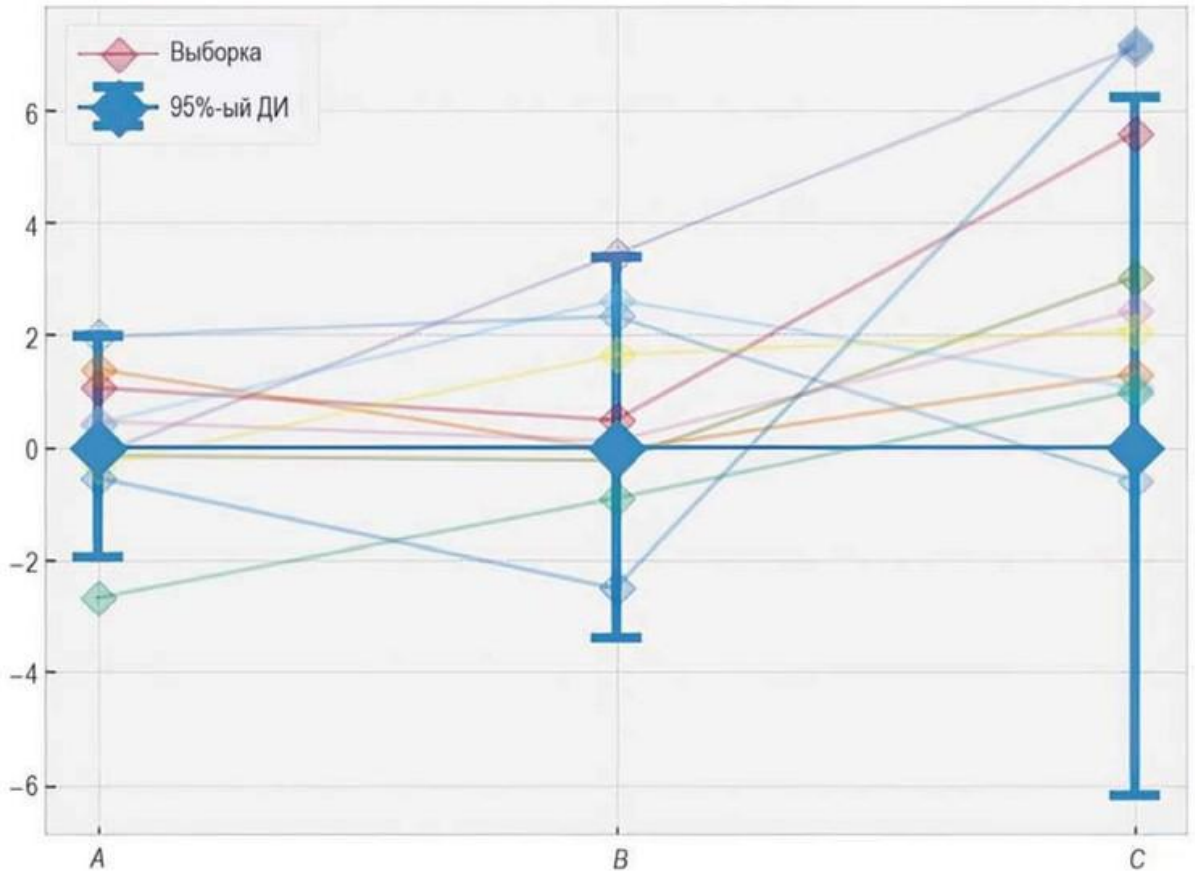


Рис. 2.5. График параллельных координат, который визуализирует MVN-распределение, нормализованное к среднему значению, для примера цен на жилье: полосы ошибок (погрешностей) обозначают 95%-ный ДИ соответствующих нормальных распределений, а бледные линии показывают выборки, взятые из многомерной гауссианы

Обратите внимание на жирные ромбики и соответствующие полосы ошибок (погрешностей) на рисунке:

- Жирные ромбики обозначают средний вектор гауссианы, который является просто нулевым вектором.
- Столбики погрешностей обозначают 95%-ные достоверные интервалы (ДИ) трех отдельных переменных. Переходя от A к B и затем к C, можно наблюдать все большие и большие ДИ, которые отображают возрастающие значения дисперсий.

Достоверный интервал (ДИ)

ДИ $(1 - \alpha)$ нормального распределения случайной величины X — это диапазон, в котором вероятность, что X в него попадает, равна $1 - \alpha$. Ученые-статистики обычно используют 95%-ный ДИ для нормального распределения. В нем нет ничего особенного, за исключением того факта, что он является порогом, который используется во многих статистических процедурах с целью определить, имеет ли что-либо значение или нет. Например, t-тест обычно использует уровень достоверности $1 - \alpha = 0{,}95$. Это соответствует тому, что значение p (меньше $\alpha = 0{,}05$) указывает на значимые результаты. Удобный факт касательно нормальных распределений заключается в том, что $\mu \pm 1{,}96\sigma$ представляет собой 95%-ный ДИ (некоторые даже используют $\mu \pm 2\sigma$), где μ и σ — соответственно среднее и стандартное отклонение переменной X, которая является легко вычислимой величиной.

Рис. 2.5 отражает наше *априорное убеждение* о нормализованных ценах трех домов. Исходя из этого, можно предположить, что все они имеют нормализованную цену, а мы имеем разные уровни неопределенности в отношении наших предположений. Далее можно было бы извлечь выборки из этого нормального распределения, которые показаны как соединенные между собой «блеклые» ромбы.

2.2.2. Обновление нормальных распределений

В этом разделе мы узнаем, как можно обновить нормальное распределение, учитывая некоторые дополнительные данные, которые будут рассмотрены далее. В частности, нам необходимо получить некоторое апостериорное убеждение о цене на дом с учетом наблюдения за величинами B или C. Это важная задача, поскольку именно так MVN (как и ГП) учится на данных.

> **Определение**
>
> Такой процесс обновления иногда называют *обусловливанием* (**Conditioning**) — получением *условного распределения* (**Conditional distribution**) переменной с учетом, что известно значение некоторой другой переменной. Другими словами, некоторое убеждение (представленное совместной трехмерной гауссианой) обусловливается значением B или C, и получается совместное апостериорное распределение для этих трех переменных.

Используя теорему Байеса (рис. 2.2), можно вывести необходимое апостериорное распределение в закрытой форме. Однако это сложно сделать в математических расчетах, поэтому нет необходимости вдаваться в подробности. Достаточно знать, что есть формула и в нее можно подставить значение B или C, которое необходимо обусловить. Тогда она покажет, каковы апостериорные распределения A, B и C. Удивительно, но апостериорное распределение гауссианы обусловлено данными, которые также являются гауссовскими, и можно получить точные апостериорные средние значения и отклонения (дисперсии), которые определяют апостериорные гауссианы. Далее в этой главе мы увидим, что при реализации ГП в Python библиотека GPyTorch выполняет это трудоемкое математическое обновление за нас.

> **Примечание**
>
> Эту формулу и ее вывод можно найти в разделе 2 главы 2 в книге Карла Эдварда Расмуссена и Кристофера К. И. Уильямса *«Gaussian Processes for Machine Learning»* (*«Гауссовы процессы для машинного обучения»*, издательство MIT Press, 2006), которую часто считают библией для ГП.

> **Замечание**
>
> Гауссиана имеет удобное свойство апостериорного распределения при условии, что данные являются другой гауссианой. Это одна из причин, почему с ними так легко работать.

Далее обновим график параллельных координат, приняв $B = 4$ в качестве примерного значения. Результат показан на рис. 2.6.

После наблюдения за величиной B в нашем апостериорном убеждении поменялось несколько вещей:

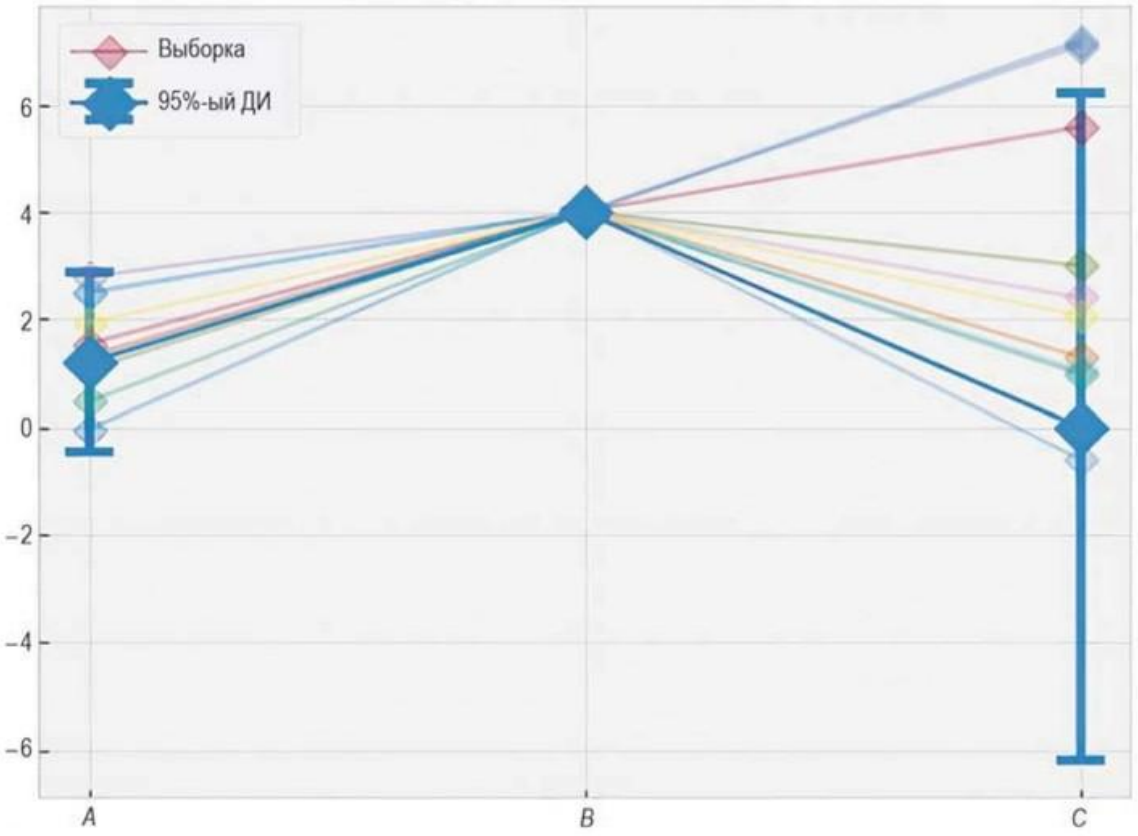


Рис. 2.6. Визуализация MVN-распределения при условии B = 4: распределение A обновлено; все нарисованные выборки интерполируют B = 4

- Распределение A изменилось и приняло немного большее среднее значение из-за положительной корреляции между A и B. Кроме того, его полосы ошибок теперь имеют меньший диапазон.
- Апостериорное распределение B стало просто частным случаем нормального распределения с нулевой дисперсией, поскольку точно известно, каково его апостериорное значение (4). Другими словами, больше нет неопределенности относительно величины B.
- Распределение C после обновления остается прежним, поскольку оно не имеет корреляции с B.

Все это имеет смысл и соответствует предположению из примера с ценами на жилье. Когда становится известно о соседнем доме, убеждение о собственном доме обновляется и становится похожим на наблюдаемую цену, соответственно неопределенность уменьшается.

Что произойдет, когда будет обусловлено значение C? Как можно догадаться, по той же причине апостериорные распределения A и B останутся неизменными. На рис. 2.7 представлен пример для C = 4.

Теперь стало известно, что был продан дом в Калифорнии. Поскольку он не имеет ничего общего с нашим домом в Миссури, предположение о цене осталось прежним.

Есть еще одна интересная вещь относительно рисунков 2.6 и 2.7. Обратите внимание, когда выставлено условие B = 4 (рис. 2.6), все примеры, нарисованные для

апостериорного MVN, проходят через точку (B, 4). Это связано с тем, что больше нет никакой неопределенности относительно значения B, и любая выборка, полученная из апостериорного распределения, должна удовлетворять ограничениям из этого условия. То же самое касается точки (C, 4) на рис. 2.7.

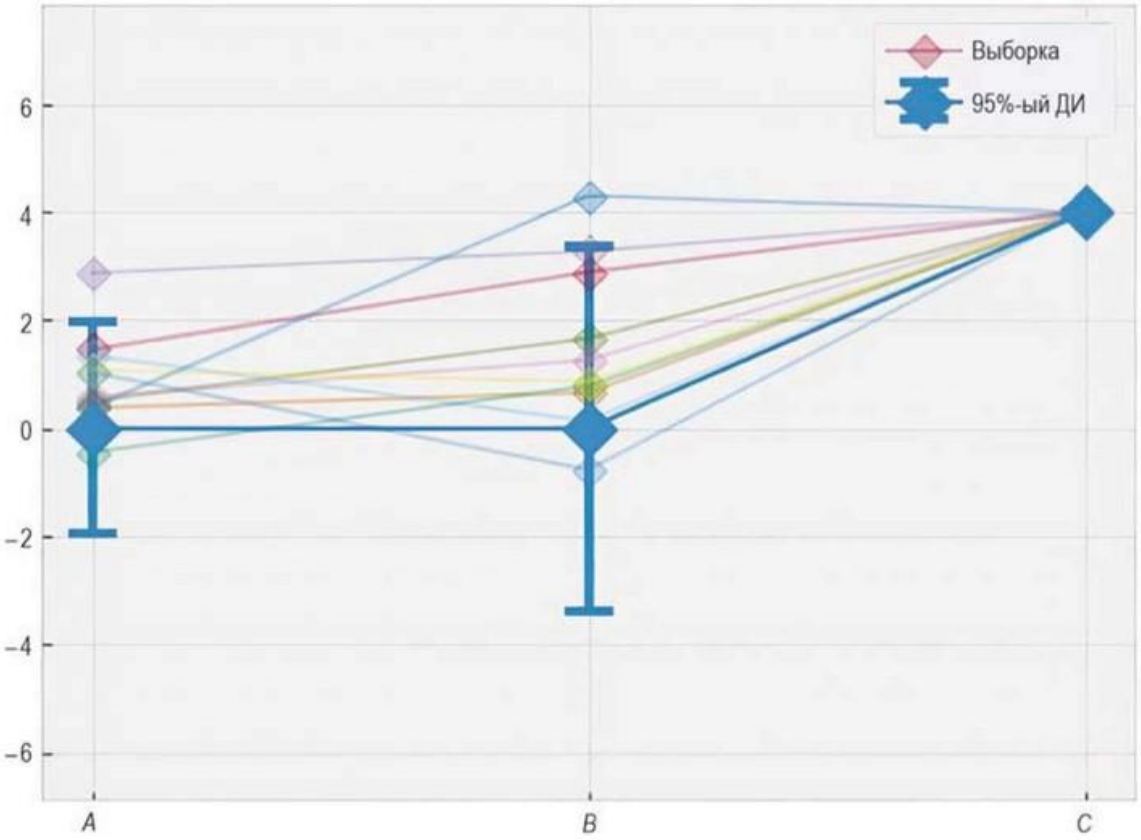


Рис. 2.7. График, визуализирующий MVN-распределение, при условии C = 4: никакое другое предельное распределение не меняется; все нарисованные выборки интерполируют C = 4

Визуально это можно представить как «узел», связывающий выборки из предыдущего (априорного) распределения (рис. 2.5) в определенной точке, которая соответствует обусловленной переменой (рис. 2.8).

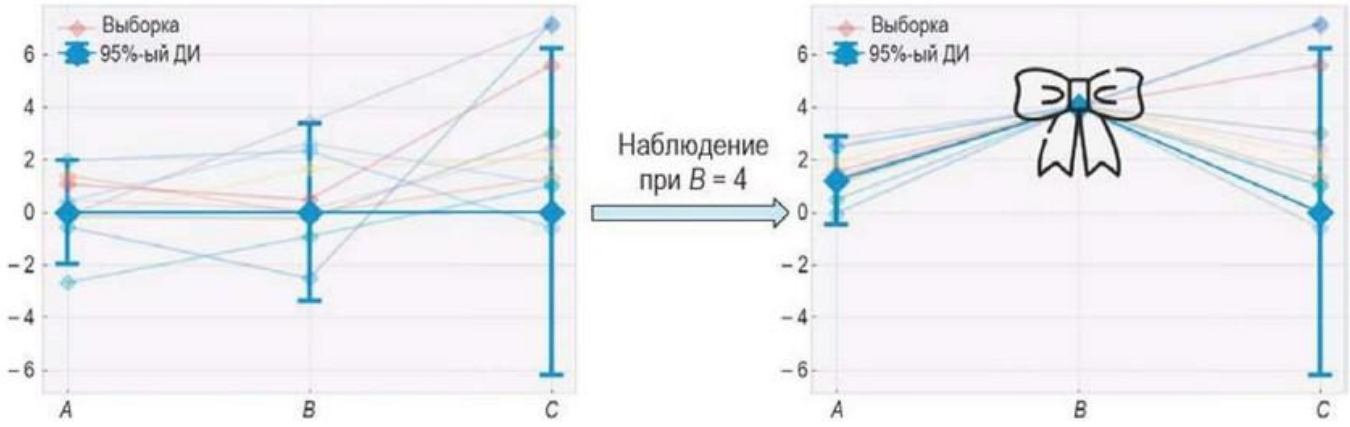


Рис. 2.8. Обусловливание гауссовой зависимости от наблюдения аналогично «завязыванию узла» вокруг этого наблюдения: все выборки из апостериорного распределения должны пройти через «узел»; в наблюдаемой точке нет никакой неопределенности

Наконец, мы теперь можем изобразить только что пройденную процедуру байесовского обусловливания (рис. 2.9.) с помощью диаграммы (аналогичной рис. 2.3).

Опять же, если обусловливать гауссиану C, апостериорные распределения некоррелированных переменных останутся неизменными. Однако если определить условие B, то коррелирующая с ним переменная A обновится.

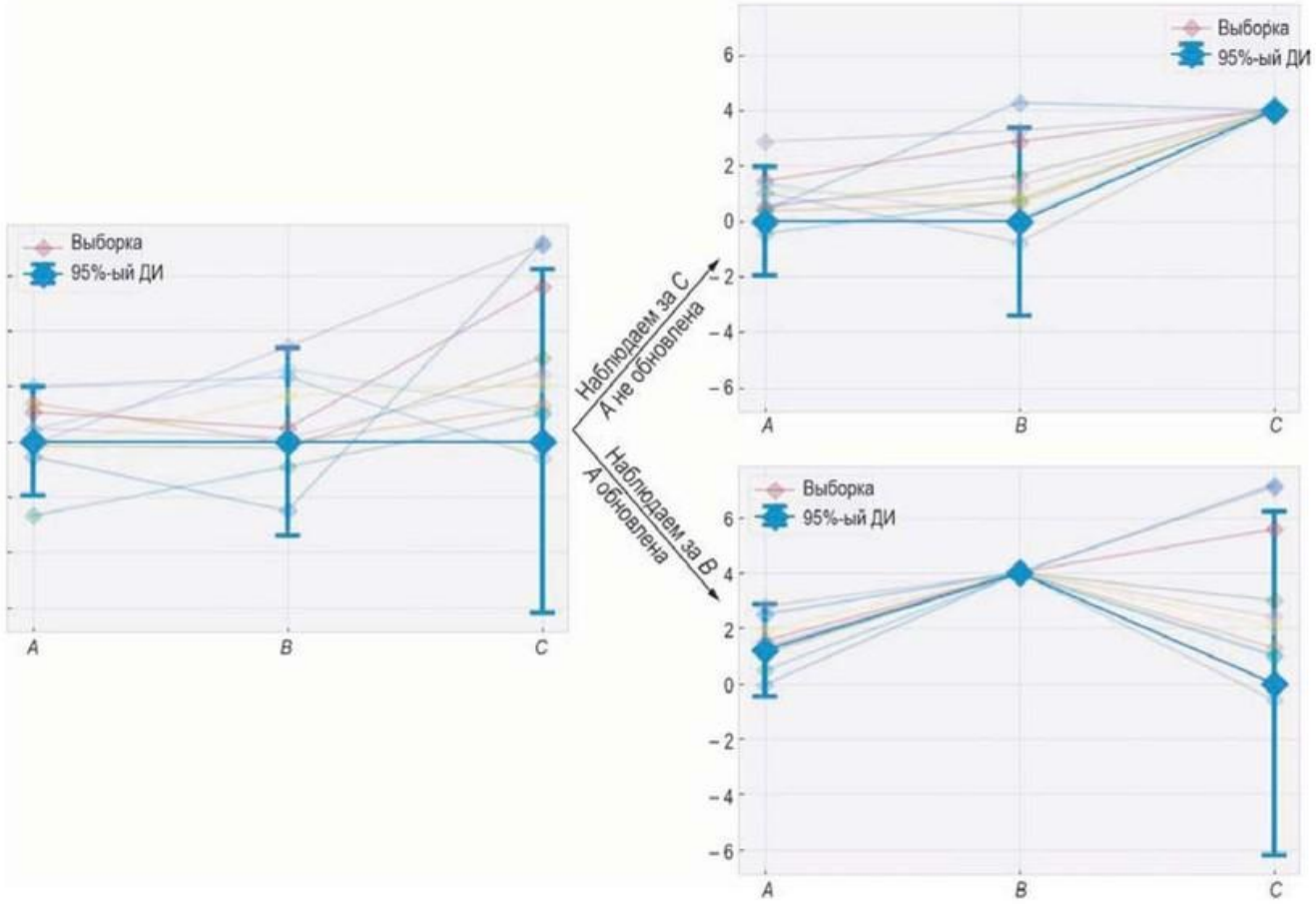


Рис. 2.9. Обновление предположения о цене дома по методу Байеса: в зависимости от того, насколько наблюдаемый дом похож на наш, апостериорное предположение о цене либо остается прежним, либо радикально обновляется

2.2.3. Моделирование множества переменных с помощью многомерных гауссовских распределений

Нормальное распределение может содержать не только три случайные переменные. Фактически их может быть любое конечное количество. Многомерная гауссиана работает аналогично той, что мы рассмотрели ранее. Предположим, вместо трехмерного гауссиана, представляющего 3 дома, есть 20-мерный, кодирующий информацию о массиве домов на одной улице. Еще больше многомерная гауссиана может моделировать дома в городе или стране.

Кроме того, с помощью графиков можно одновременно визуализировать каждую отдельную переменную многомерной гауссианы. Это возможно, поскольку все они соответствует одной полосе ошибок (погрешностей), занимающей только один слот на оси X.

Еще раз нормализуем средний вектор многомерной гауссианы так, чтобы он был нулевым. Показывать его ковариационную матрицу 20×20 неудобно, поэтому можно построить *тепловую карту* (**Heat map**), которая ее визуализирует (рис. 2.10).

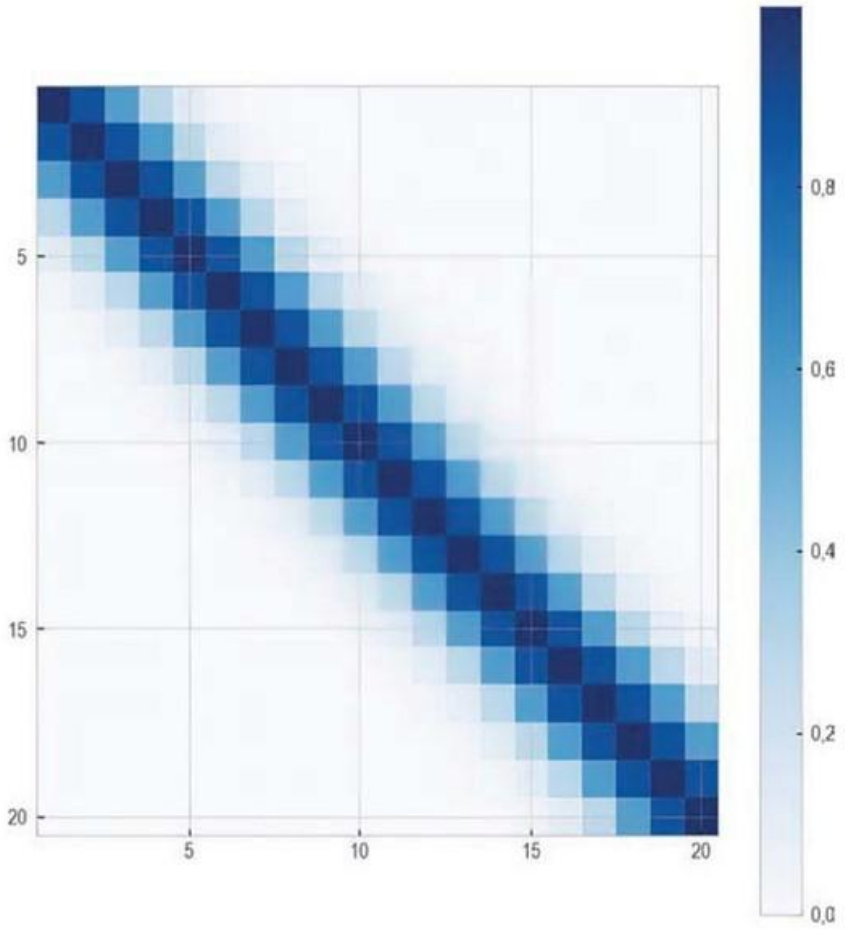


Рис. 2.10. Тепловая карта ковариационной матрицы 20-мерного распределения Гаусса: соседние переменные коррелируют сильнее, чем те, которые находятся далеко друг от друга, на что указывает более темный оттенок

В данном случае все диагональные элементы или отклонения отдельных переменных равны единицам. Далее переменные упорядочиваются так, чтобы близкие друг к другу коррелировали. Иными словами, их ковариация принимает большее значение. С другой стороны, переменные, находящиеся далеко друг от друга, менее коррелированы, а их ковариации близки к нулю. Например, любая пара последовательных переменных в этой гауссиане (первая и вторая, вторая и третья и т. д.) имеет ковариацию примерно 0,87. То есть любые два дома, находящиеся рядом, имеют ковариацию 0,87. Если рассмотреть 1-ю и 20-ю переменные (один дом на одном конце улицы, а другой — на другом), их ковариация фактически равна нулю.

Это интуитивно понятно, поскольку ожидается, что цены на дома, расположенные рядом, будут одинаковыми. Как только известна цена дома, мы получим больше информации о ценах на дома в этом районе, чем о ценах тех, которые находятся далеко за его пределами.

Как это отображается на графике? На рис. 2.11 показаны априорная гауссиана (слева) и апостериорная гауссиана (справа), обусловленная 10-й переменной, имеющей

значение 2. По сути, вторая моделирует событие, в котором становится известно, что цена 10-го дома равна 2 (точная единица измерения опущена):

- Во-первых, еще раз видно, что полосы ошибок и выборки связаны в узел вокруг наблюдения, которое определяется в апостериорном распределении.
- Во-вторых, из-за корреляционной структуры, налагаемой ковариационной матрицей, средние значения переменных, близких к 10-й, «перетаскиваются» вверх так, что средний вектор теперь плавно интерполирует точку (10, 2). Это означает, что мы обновили убеждение, поскольку цены на близлежащие дома теперь отражают полученную нами информацию.
- Наконец, неопределенность (обозначенная полосами ошибок) вокруг точки (10, 2) уменьшается после обусловливания. Это очень полезное свойство, поскольку, если известно значение переменной, уверенность в значениях других переменных, коррелирующих с известной нам переменной, будет больше. То есть если мы знаем цену дома, становимся более уверенными в ценах соседних домов. Это свойство лежит в основе калиброванной количественной оценки неопределенности, которую предлагает ГП и которая будет рассмотрена в следующем разделе этой главы.

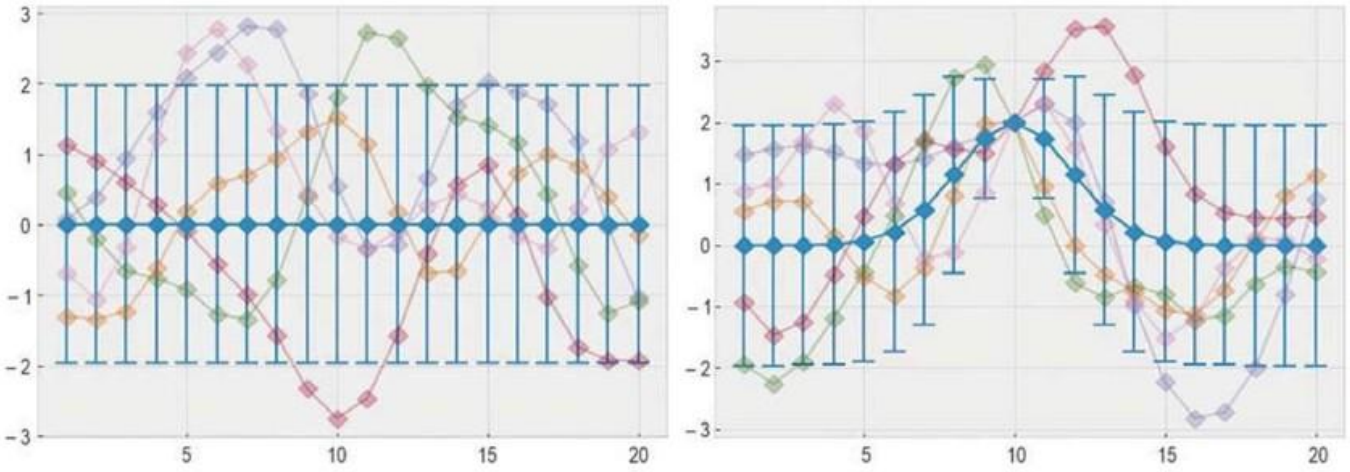


Рис. 2.11. Ошибки и выборки, взятые из априорного (слева) и апостериорного (справа) гауссовских распределений, которые обусловлены тем, что 10-я переменная имеет значение 2: неопределенность в переменных, близких к 10-й, уменьшается в апостериорном распределении, а их средние значения обновляются до близких к значению 2

2.3. Переход от конечной гауссианы к бесконечной

Предположим, что теперь вместо 3 или 20 величин есть бесконечное их количество, и все они соответствуют нормальному распределению. Такая бесконечномерная гауссиана называется *гауссовским процессом* (**Gaussian process**).

Представьте, что нужно спрогнозировать цены на жилье на очень большой и густонаселенной территории. Ее масштаб настолько велик, что, если отойти от дома на небольшое расстояние, можно прийти к другому дому. Учитывая высокую плотность переменных, всю область можно рассматривать как имеющую бесконечное количество домов (распределение Гаусса имеет бесконечно много переменных).

Это продемонстрировано на рис. 2.12. Здесь используется набор данных, содержащий 5000 цен на жилье в Калифорнии. В верхней левой части показаны отдельные точки данных на диаграмме. На остальных частях рисунка моделирование данных осуществляется с использованием различного количества переменных, где каждая из них соответствует региону на карте Калифорнии. По мере увеличения их количества модель становится более мелкозернистой. Когда число точек бесконечно большое (когда можно сделать прогноз в любой области на карте), какой бы маленькой она ни была, модель существует в бесконечномерном пространстве.

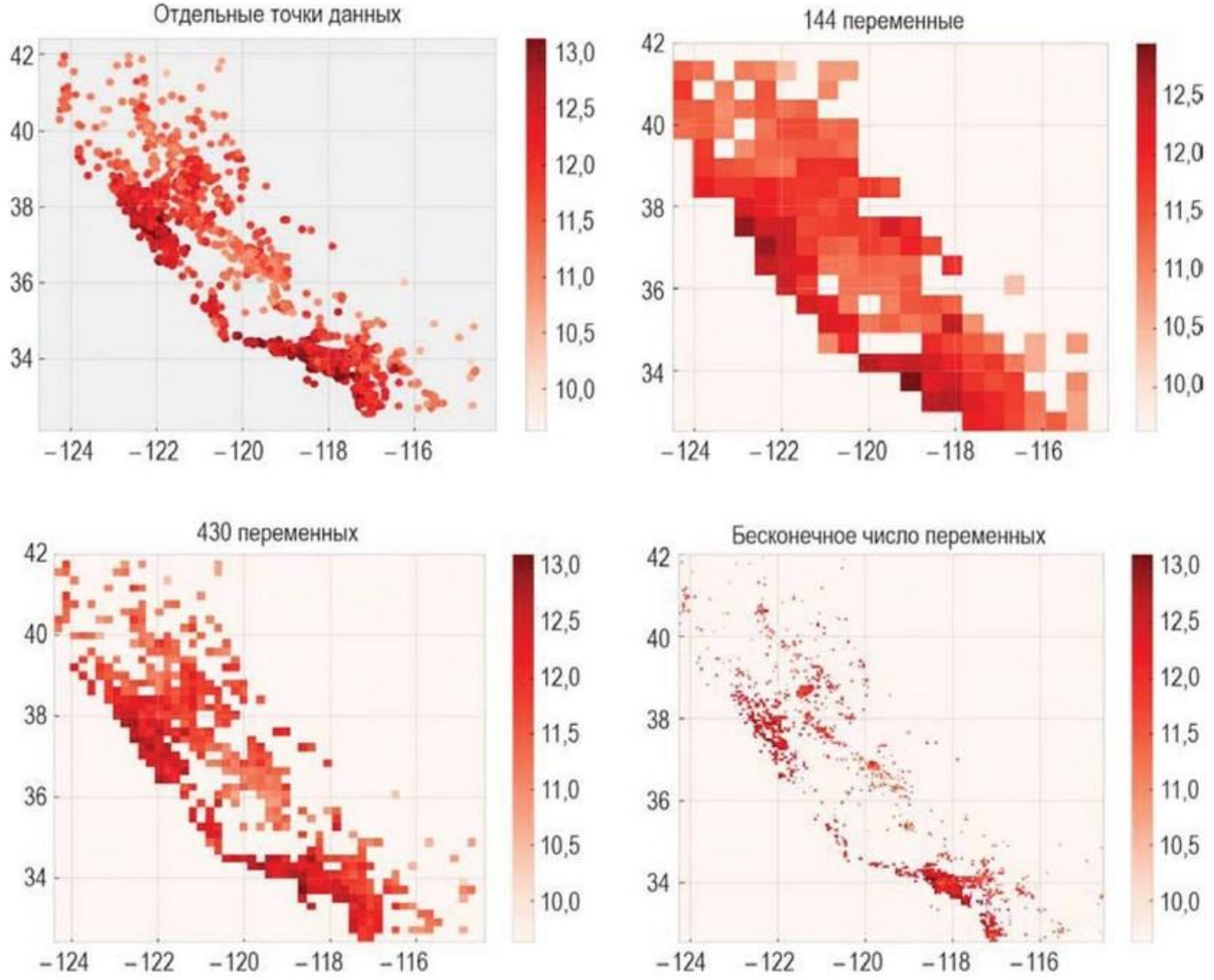


Рис. 2.12. Моделирование цен на жилье в Калифорнии с использованием различного количества переменных: чем больше переменных, тем более гладкой становится модель и тем ближе мы приближаемся к бесконечномерной гауссиане

Это именно то, что представляет собой гауссовский процесс (ГП) — распределение Гаусса в бесконечномерном пространстве. Возможность делать прогноз в любом регионе помогает отойти от конечномерного MVN-распределения и получить *модель машинного обучения* (**Machine learning model**, **ML-model**). Строго говоря, концепция гауссовского распределения неприменима, когда есть бесконечно много переменных, поэтому важно дать некоторое техническое определение.

Определение

Гауссовский процесс — это набор случайных переменных, в котором совместное распределение каждого конечного их подмножества представляет собой нормальное распределение (MVN).

Это определение означает, что если есть модель ГП для описания функции f, то ее значения в любом наборе точек моделируются распределением MVN. Например, вектор переменных $[f(1)\, f(2)\, f(3)]$ соответствует какой-то 3-мерной гауссиане, вектор $[f(1)\, f(0)\, f(10)\, f(5)\, f(3)]$ соответствует другой гауссиане (5-мерной), а вектор $[f(0{,}1)\, f(0{,}2)\, \ldots\, f(9{,}9)\, f(10)]$ соответствует очередной n-мерной гауссиане.

Это продемонстрировано на рис. 2.13. На первых трех частях показаны 3-, 11- и 101-мерные гауссианы соответственно. На последней части рисунка изображена бесконечная гауссиана, которая и является ГП.

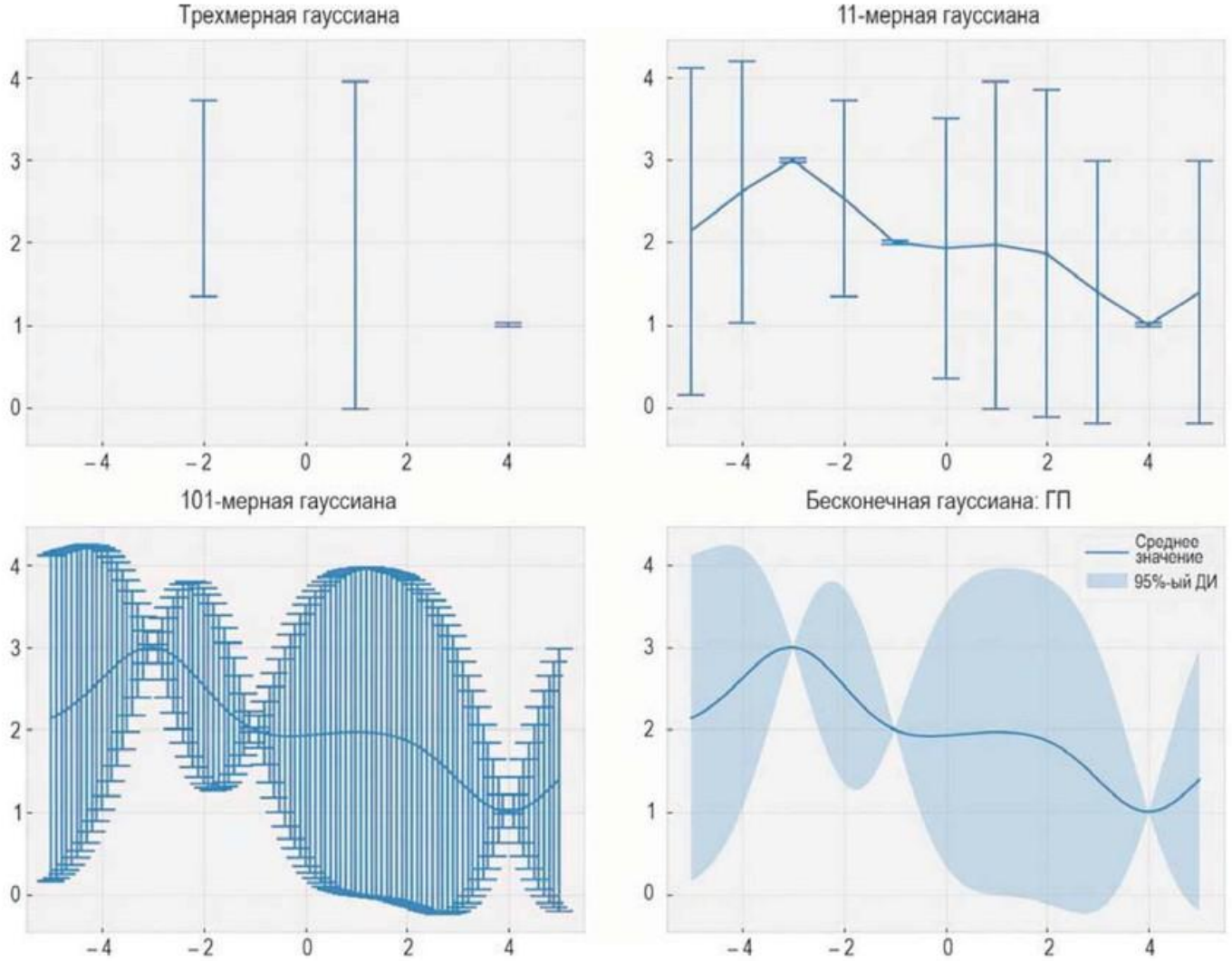


Рис. 2.13. Графики различных гауссовских распределений: любое конечное подмножество ГП является MVN-распределением; когда число переменных приближается к бесконечности, мы получаем ГП и можем делать прогнозы в любом месте области

Поскольку теперь мы находимся в бесконечных измерениях, больше не имеет смысла говорить о среднем векторе и ковариационной матрице. Вместо этого мы теперь имеем *функцию среднего значения* (или просто *функцию среднего*) и *кова-*

риационную функцию, но соответствующие роли этих двух объектов остаются такими же, как и в MVN:

- Во-первых, функция среднего, принимающая один входной сигнал x, вычисляет математическое ожидание для значения функции $f(x)$.
- Во-вторых, ковариационная функция принимает два входных параметра (x_1 и x_2) и вычисляет ковариацию между $f(x_1)$ и $f(x_2)$. Если x_1 совпадает с x_2, то значение ковариации представляет собой просто дисперсию (отклонение) нормального распределения $f(x)$. Если x_1 отличается от x_2, ковариация обозначает корреляцию между ними.

Поскольку среднее и ковариация являются функциями, мы больше не привязаны к фиксированному числу переменных. Вместо этого мы фактически имеем бесконечное количество переменных и можем делать прогнозы где угодно (рис. 2.13). Вот почему, несмотря на то, что ГП и обладает всеми свойствами MVN, он существует в бесконечных измерениях.

По той же причине (как следует из названия этой главы) ГП можно рассматривать как распределение по функциям. Прогресс, пройденный на данный момент в этой главе (от одномерного нормального распределения до ГП), обобщен в табл. 2.1.

Таблица 2.1. Объекты гауссовского распределения и что они моделируют: ГП позволяет работать в бесконечных измерениях, моделируя функции вместо чисел или векторов

Тип распределения	Число моделируемых переменных	Описание
Одномерное нормальное распределение	1	Распределение по числам
MVN-распределение	Большое конечное число	Распределение по векторам конечной длины
ГП	Бесконечно большое число	Распределение по функциям

Чтобы увидеть ГП в действии, еще раз рассмотрим процедуру аппроксимации кривой (рис. 2.1) в начале главы, где область значений ограничена диапазоном от −5 до 5. Результат продемонстрирован на рис. 2.14.

Для каждой из частей рисунка справедливо следующее:

- Сплошная линия посередине — это функция среднего, аналогичная сплошной линии, соединяющей ромбы на рис. 2.1.
- Заштрихованная область представляет собой 95%-ный ДИ по всей области, что соответствует полосам ошибок на рис. 2.1.
- Различные волнистые линии представляют собой выборки, взятые из соответствующего ГП.

Прежде чем рассмотреть какие-либо данные, начнем с априорного ГП в верхней левой части рисунка. Как и априорное MVN-распределение, он дает постоянный

прогноз среднего значения и неопределенность при отсутствии обучающих данных. Это разумное поведение.

Самое интересное происходит, когда мы обусловливаем ГП различными точками данных. Это показано на остальных частях рис. 2.14. Когда ГП работает в большой области, средний прогноз и полученные из апостериорного распределения выборки плавно интерполируют точки данных в обучающем наборе, в то время как неопределенность в отношении значения функции (количественно выраженная в ДИ) плавно уменьшается в областях вокруг этих наблюдений. Это называется *калиброванной количественной оценкой неопределенности* (**calibrated quantification of uncertainty**), и это одно из главных преимуществ ГП.

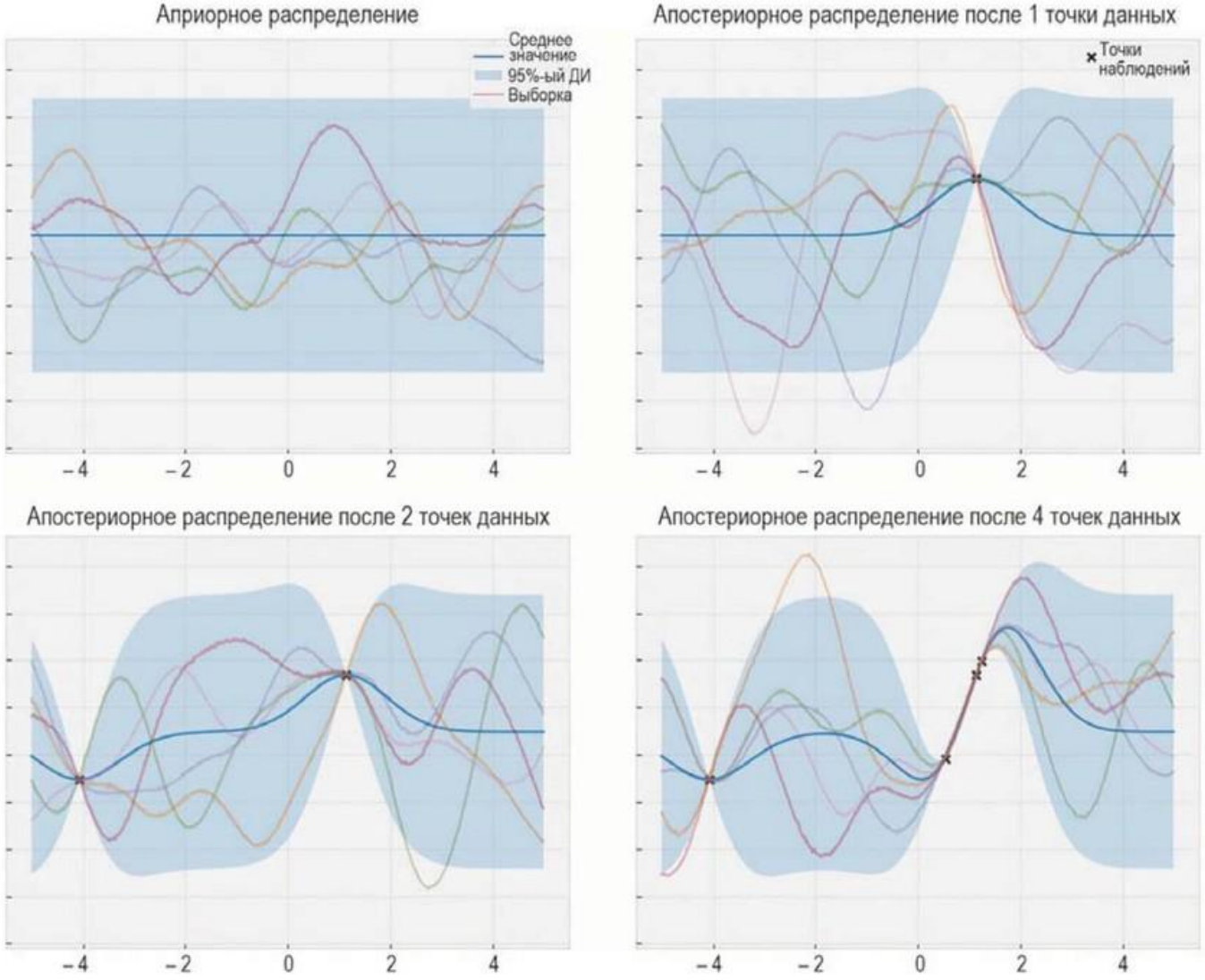


Рис. 2.14. Прогнозы ГП на основе 0, 1, 2 и 4 наблюдений

Плавность ГП

Свойство *плавности* (**Smoothness**) является ограничением, которое требует, чтобы подобные точки коррелировали друг с другом. Другими словами, схожие точки должны приводить к схожим значениям функции. Именно поэтому, когда мы обусловливаем, например, точку данных 3 (верхняя правая часть рис. 2.14), средние прогнозы для 2,9 и 3,1 обновляются и принимают более высокие значения, чем их априорные средние.

Точки 2,9 и 3,1 подобны точке 3, поскольку расположены близко к ней. Данная плавность задается через ковариационную функцию, которая будет представлена в главе 3. Несмотря на то что примеры, рассматриваемые до этого момента, относятся к одному измерению, такая плавность также сохраняется, когда пространство поиска является многомерным.

В целом можно убедиться, что ГП является нормальным распределением, расширенным до бесконечных измерений. И благодаря многим удобным математическим свойствам гауссовых распределений, ГП не только может дать прогноз среднего значения, но также количественно определить неопределенность в отношении значений функции через его прогностические ковариации.

Среднее предсказание проходит точно через точки обучающих данных, и неопределенность в них падает.

Моделирование не-гауссовских данных

В реальной жизни не все данные соответствуют распределению Гаусса. Например, для значений, которые ограничены числовым диапазоном, или переменных, которые не соответствуют колоколообразному распределению, гауссово распределение не подходит и может привести к прогнозам низкого качества.

В этих случаях можно применять различные методы обработки с целью преобразовать точки данных в соответствии с распределением Гаусса. Например, преобразование Бокса — Мюллера — это алгоритм, который генерирует пары *нормально распределенных случайных чисел* из *равномерно распределенных случайных чисел*.

2.4. Реализация ГП в Python

В этом разделе мы сделаем первый шаг к реализации ГП на Python. Основная цель — познакомиться с API и синтаксисом, которые будут использоваться для этой задачи, и научиться воссоздавать визуализации, которые были рассмотрены до этого момента. Данный практический раздел также поможет глубже понять ГП.

Сначала нужно убедиться, что прилагаемый к книге код загружен и установлены необходимые библиотеки. Подробные инструкции, как это сделать, приведены в начале книги. Мы будем использовать код, включенный в Jupyter Notebook CH02/01, — `Gaussian processes.ipynb`.

2.4.1. Настройка обучающих данных

Прежде чем начать реализовывать код, сначала стоит потратить некоторое время на создание целевой функции, которую нужно смоделировать, и набора обучающих данных. Для этого следует импортировать PyTorch для расчета и управления *тензорами*, а также Matplotlib для визуализации данных:

```
import torch
import matplotlib.pyplot as plt
```

Целевая функция в этом примере — это одномерная функция Форрестера. Она является мультимодальной с одним глобальным максимумом и одним локальным

максимумом, что делает подбор и нахождение максимума функции нетривиальной задачей. Функция имеет следующую формулу:

$$f(x) = -\frac{1}{5}(x+1)^2 \sin(2x+2) + 1.$$

Она реализуется так:

```
def forrester_1d(x):
    y = -((x + 1) ** 2) * torch.sin(2 * x + 2) / 5 + 1
    return y.squeeze(-1)
```

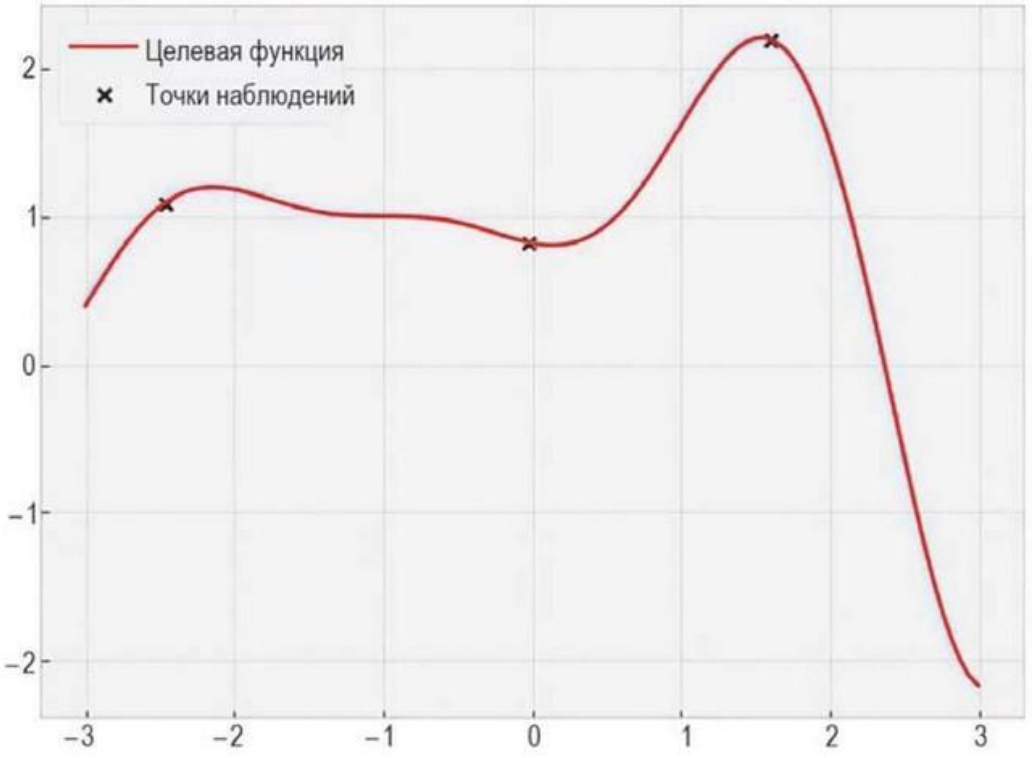


Рис. 2.15. Целевая функция, которая используется в текущем примере, показана сплошной линией; точки в наборе обучающих данных отмечены маркерами

Построим функцию на графике. Здесь мы ограничиваемся областью между –3 и 3 и вычисляем функцию Форрестера на плотной сетке из 100 точек внутри диапазона. Также нам нужны несколько точек выборки для обучения, которые будут сгенерированы путем случайной выборки с помощью `torch.rand()` и сохранены в `train_x`. Переменная `train_y` содержит *метки* точек обучения, которые можно получить, вычислив `forrester_1d(train_x)`. График сгенерирован с помощью кода ниже и изображен на рис. 2.15:

```
xs = torch.linspace(-3, 3, 101).unsqueeze(1)
ys = forrester_1d(xs)

torch.manual_seed(0)
train_x = torch.rand(size=(3, 1)) * 6 - 3
train_y = forrester_1d(train_x)
```

```
plt.figure(figsize=(8, 6))

plt.plot(xs, ys, label="objective", c="r")
plt.scatter(train_x, train_y, marker="x", c="k", label="observations")

plt.legend(fontsize=15);
```

Три маркера на графике — это точки, которые случайным образом выбраны для включения в набор обучающих данных. Их местоположение хранится в `train_x`, а метки (значения функции Форрестера в этих местах) расположены в `train_y`. Это ставит перед нами задачу реализовать и обучить ГП по этим трем точкам и визуализировать его прогнозы в диапазоне от –3 до 3. Здесь мы также создали переменную `xs`, которая представляет собой плотную сетку в этом диапазоне.

2.4.2. Реализация класса ГП

В этом разделе мы узнаем, как реализовать модель ГП в Python. Для этого будем использовать библиотеку GPyTorch.

> **Важное примечание**
>
> Философия разработки с GPyTorch заключается в том, чтобы использовать библиотеку глубокого обучения (Deep learning) PyTorch, и все ее классы моделей должны расширять базовый класс. Если вы знакомы с реализацией нейронных сетей в PyTorch, вероятно, вам известна база `torch.nn.Module`. Используя GPyTorch, обычно расширяют класс `gpytorch.models.ExactGP`.

Для реализации нашего класса модели используем следующую структуру:

```
import gpytorch
class BaseGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
    ...
    def forward(self, x):
    ...
```

Здесь реализуется класс `BaseGPModel`, который имеет два конкретных метода: `__init__()` и `forward()`. Поведение ГП-модели во многом зависит от того, как они будут написаны. И независимо от того, какую модель нужно реализовать, класс модели должен содержать эти методы.

Сначала обсудим метод `__init__()`. Его задача в том, чтобы принять как набор обучающих данных, определенный аргументами `train_x` и `train_y`, так и *функцию вероятности* (**Likehood function**), хранящуюся в переменной `likehood`, а также инициализировать модель ГП, которая является объектом `BaseGPModel`. Реализуем метод следующим образом:

```
def __init__(self, train_x, train_y, likelihood):
    super().__init__(train_x, train_y, likelihood)
    self.mean_module = gpytorch.means.ZeroMean()
    self.covar_module = gpytorch.kernels.RBFKernel()
```

Три входных аргумента передаются методу `__init__()` нашего суперкласса, а встроенная реализация `gpytorch.models.ExactGP` берет на себя всю тяжелую работу за нас. Остается определить среднее значение и ковариационные функции.

GPyTorch имеет широкий выбор вариантов как для функции среднего, так и для ковариации, которые мы рассмотрим в главе 3. На данный момент будем использовать наиболее распространенные параметры ГП:

- `gpytorch.means.ZeroMean()` для функции среднего, которая выводит прогнозы с нулевым средним значением в априорном виде.
- `gpytorch.kernels.RBFKernel()` для функций ковариации, которая реализует ядро *радиальной базисной функции* (**Radial Basis Function**, **RBF**). Это одна из наиболее часто используемых ковариационных функций для ГП, которая реализует идею, что точки данных, близкие друг к другу, коррелируют.

Эти объекты хранятся в атрибутах класса `mean_module` и `covar_module` соответственно. Это все, что нужно сделать для метода `__init__()`.

Теперь обратите внимание на метод `forward()`. Он очень важен, поскольку определяет, как модель должна обрабатывать входные данные. Если вы работали с нейросетями в PyTorch, то знаете, что метод `forward()` сетевого класса последовательно передает свои входные данные через слои сети. А выходные данные последнего слоя — это результат работы нейросети. В PyTorch каждый уровень реализован как модуль, что является базовым строительным блоком любого объекта, обрабатывающего данные.

Метод `forward()` в GPyTorch работает аналогичным образом: функции среднего и ковариации реализованы в виде модулей, и входные данные передаются в них последовательно, а не параллельно. Результаты затем объединяются для создания MVN-распределения. Разница между PyTorch и GPyTorch показана на рис. 2.16.

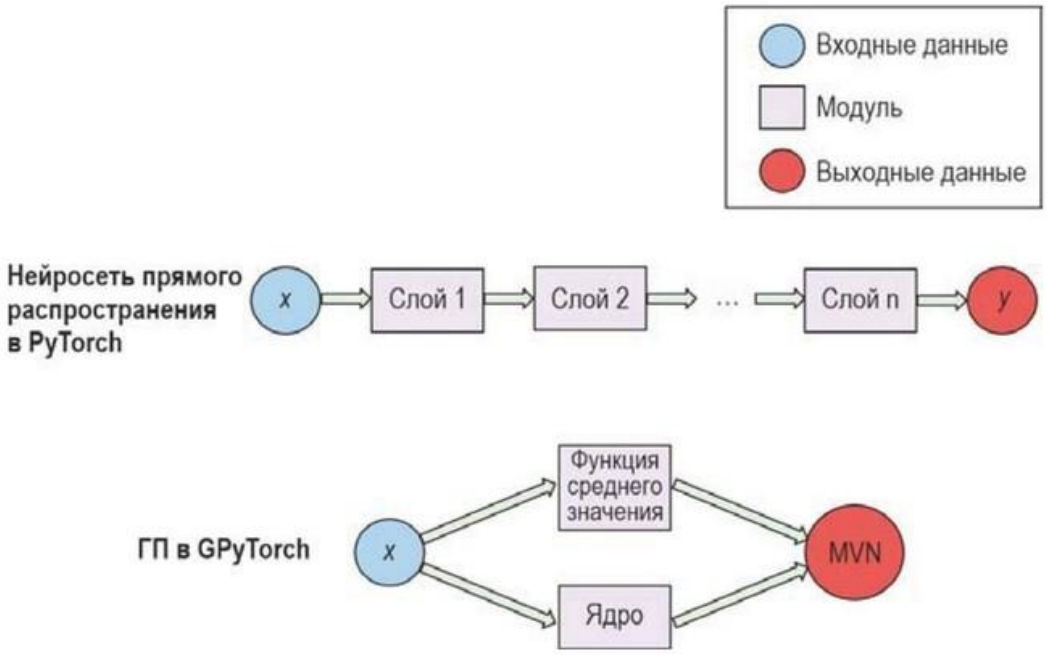


Рис. 2.16. Обработка данных в PyTorch и GPyTorch соответствующими методами `forward()`: входные данные обрабатываются различными модулями для получения окончательного результата; это либо число для нейросети прямого распространения, либо MVN-распределение для ГП

Метод `forward()` реализован следующим кодом:

```
def forward(self, x):
    mean_x = self.mean_module(x)
    covar_x = self.covar_module(x)
    return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

Логика здесь довольно проста. Достаточно вызвать функции среднего и ковариации на входе `x`, чтобы вычислить прогнозы для среднего значения и ковариации соответственно. Затем нужно вернуть MVN-распределение, реализованное в классе `gpytorch.distributions.MultivariateNormal`, с соответствующим средним значением и ковариацией. Другими словами, не нужно ничего делать, кроме создания распределения со средним вектором и ковариационной матрицей, вычисленной на основе атрибутов `mean_module` и `covar_module` класса модели.

И это все, что нужно сделать! Удивительно, насколько легко реализовать модель ГП с помощью GPyTorch. Самым большим выводом для нас здесь является необходимость реализовать функции среднего и ковариации в методе `__init__()`. А в методе `forward()`, когда нужно сделать прогноз, можно просто вызвать их, используя переданные входные данные.

2.4.3. Создание прогнозов с помощью ГП

Имея класс `BaseGPModel`, мы готовы сделать прогнозы с помощью ГП. Напомним, что в методе `__init__()` нужно передать функцию вероятности (`likehood`) в дополнение к обучающим данным. Во многих задачах регрессии объекта `gpytorch.likelihoods.GaussianLikelihood` достаточно. Он создается так:

```
likelihood = gpytorch.likelihoods.GaussianLikelihood()
```

Теперь можно инициализировать объект `BaseGPModel`. Но прежде чем сделать это, используя обучающие данные с тремя входами, сначала нужно сделать прогнозы с помощью априорного ГП.

> **Замечание**
>
> Одним из преимуществ использования ГП является то, что даже без каких-либо данных все равно можно делать прогнозы, используя знания, заложенные в априорную функцию среднего.

Для инициализации объекта ГП без каких-либо обучающих данных передаем `None` как в качестве обучающего объекта (`train_x`), так и в качестве меток (`train_y`). Делается это следующим образом:

```
model = BaseGPModel(None, None, likelihood)
```

Прежде чем можно будет делать какие-либо прогнозы, необходима некоторая подготовка. Сначала установим гиперпараметры:

```
lengthscale = 1
noise = 1e-4
```

```
model.covar_module.lengthscale = lengthscale
model.likelihood.noise = noise

model.eval()
likelihood.eval()
```

Чем управляет каждый из них, мы обсудим в главе 3. На данный момент просто используем некоторые значения по умолчанию: 1 для шкалы длины и 0,0001 для дисперсии шума. Заключительная деталь: включить режим прогнозирования как в модели ГП, так и в ее функции вероятности, вызвав метод `eval()` из соответствующих объектов.

Когда подготовка завершена, можно наконец использовать модель на тестовых данных и делать прогнозы. Это реализуется так:

```
with torch.no_grad():
    predictive_distribution = likelihood(model(xs))
```

Помните, что метод `forward()` возвращает MVN-распределение — результат, когда мы передаем некоторые обучающие данные через модель с помощью `model(xs)`. В синтаксисе PyTorch вызов `model(xs)` — это сокращение вызова метода `forward()` для обучающих данных `xs`. Мы также передаем тот же вывод через функцию `likehood`, которая включает дисперсию шума в прогнозы. Другими словами, то, что мы храним в `predictive_distribution`, — это распределение, которое представляет прогноз для контрольных точек `xs`. Более того, мы вычисляем этот объект внутри `torch.no_grad()`, что является хорошей практикой, когда нужно, чтобы PyTorch не отслеживал *градиенты* этих вычислений.

> **Примечание**
>
> Вычислять *градиенты* операций следует только тогда, когда нужно оптимизировать некоторые параметры модели с помощью *градиентного спуска* (**Gradient descent**). Но когда выполняются прогнозы, модель лучше сохранять полностью «фиксированной», поэтому уместно отключить проверку градиента.

2.4.4. Визуализация прогнозов ГП

Имея распределение Гаусса, теперь можно воссоздать графики ГП, которые мы наблюдали до этого момента. Каждый из них состоит из функции среднего (μ), которую можно было бы получить из MVN-распределения с помощью:

```
predictive_mean = predictive_distribution.mean
```

Кроме того, необходимо продемонстрировать 95%-ный ДИ. Математически это можно сделать, извлекая диагональные элементы прогностической ковариационной матрицы Σ (помните, что эти элементы обозначают отдельные дисперсии σ^2), затем извлекая квадратные корни из этих значений для вычисления стандартных отклонений σ и вычисляя диапазон ДИ от $\mu \pm 1{,}96\sigma$.

К счастью, вычисление 95%-ного ДИ — обычная операция при работе с ГП, поэтому GPyTorch предлагает удобный вспомогательный метод, называемый `confidence_region()`, который можно вызывать непосредственно из объекта распределения:

```
predictive_lower, predictive_upper =
    predictive_distribution.confidence_region()
```

Этот метод возвращает кортеж из двух тензоров, которые хранят нижнюю и верхнюю конечные точки ДИ соответственно.

Нам также могут понадобиться выборки для графика из текущей модели. Это можно сделать напрямую, вызвав метод `sample()` из гауссовского объекта `predictive_distribution`. Если ни одного входного аргумента не будет передано, метод вернет одну выборку. Нам нужно сделать выборку из ГП пять раз. Это можно реализовать следующим образом:

```
torch.manual_seed(0)
 samples = predictive_distribution.sample(torch.Size([5]))
```

Мы передаем объект `torch.Size()` с целью обозначить, что нужно вернуть пять выборок. Установка случайного начального числа перед выборкой — хорошая практика, обеспечивающая воспроизводимость кода. С помощью этого уже можно построить несколько графиков. Первое, что нужно сделать, это просто построить функцию среднего значения:

```
plt.plot(xs, predictive_mean.detach(), label="mean")
```

Что касается 95%-ного ДИ, обычно используется заштрихованная область, подобная той, которую мы наблюдали. Это можно сделать с помощью функции `fill_between()` в Matplotlib:

```
plt.fill_between(
 xs.flatten(),
 predictive_upper,
 predictive_lower,
 alpha=0.3,
 label="95% CI"
)
```

Наконец, можно построить отдельные выборки:

```
for i in range(samples.shape[0]):
 plt.plot(xs, samples[i, :], alpha=0.5)
```

Данный код создаст график, показанный на рис. 2.17.

Можно увидеть, что во всей области априорный ГП дает функцию среднего, которая постоянна при нуле, и 95%-ный ДИ также постоянен. Этого и следовало ожидать, поскольку мы использовали объект `gpytorch.means.ZeroMean()` для реализации функции среднего значения, и без каких-либо обучающих данных априорные прогнозы по умолчанию принимают значение 0.

Учитывая сказанное, среднее значение и ДИ — это всего лишь измерения ожиданий. Они обозначают среднее поведение прогнозов во многих различных реализациях того, что могло бы быть. Однако, когда мы рисуем отдельные выборки, видно, что каждая из них имеет очень сложную форму, которая совсем не является постоянной. Все это означает, что, хотя ожидаемая ценность прогноза в любой момент

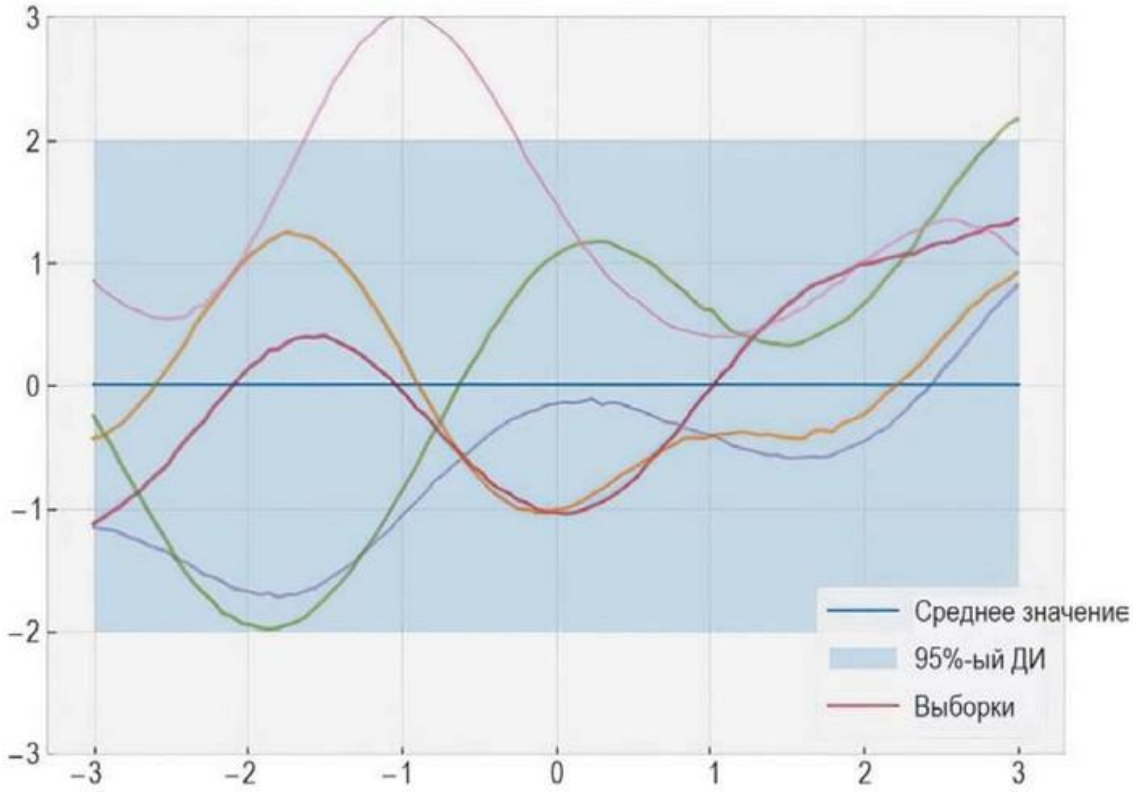


Рис. 2.17. Прогнозы, сделанные априорным ГП с нулевым средним значением и ядром RBF: несмотря на то что среднее значение и ДИ постоянны, отдельные выборки демонстрируют сложное нелинейное поведение

и равна нулю, он может принимать широкий диапазон значений. Это демонстрирует, что ГП может гибко моделировать сложное нелинейное поведение.

До сих пор прогнозы ГП и их визуализации реализовывались без каких-либо обучающих данных. Далее попробуем обучить модель на случайном наборе и посмотрим, как меняются прогнозы. Хорошая новость заключается в том, что уже написанный код можно использовать повторно без изменений, за исключением, что теперь ГП инициализируется вместе с обучающими данными (ранее мы использовали `None` для первого и второго аргументов):

```
model = BaseGPModel(train_x, train_y, likelihood)
```

Результат повторного запуска кода показан на рис. 2.18.

Это именно тот тип прогноза, который нужен. Линии среднего значения и выборки прекрасно интерполируют наблюдаемые точки данных, и неопределенность (измеренная с помощью ДИ) также уменьшается вокруг них.

Мы уже можем убедиться, насколько полезна количественная оценка неопределенности с точки зрения моделирования целевой функции. После наблюдения всего трех точек данных ГП получил довольно хорошее приближение к истинной целевой функции. Фактически почти весь целевой объект находится в пределах 95%-ного ДИ. Это указывает на то, что ГП успешно учитывает, как может вести себя функция даже в тех регионах, где еще нет данных от нее. Такая оценка особенно полезна, когда действительно нужно принимать решения, основываясь на модели, т. е. когда надо решить, в каких точках следует наблюдать значение функции, чтобы найти оптимум. Это мы рассмотрим в следующей части книги.

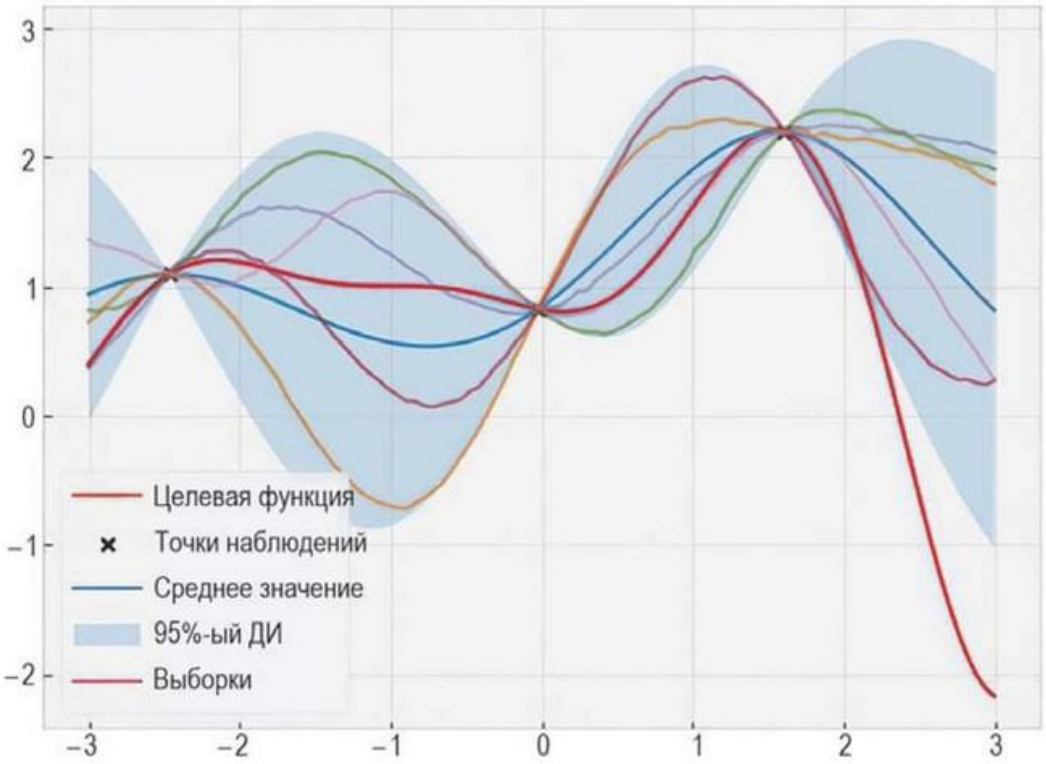


Рис. 2.18. Прогнозы ГП: функция среднего и случайные выборки плавно интерполируют точки обучающих данных, в то время как неопределенность исчезает в областях, окружающих эти точки

2.4.5. Выход за рамки одномерных целевых функций

До этого момента мы наблюдали только примеры, обученные на одномерных целевых функциях. Однако ничего не ограничивает нас использовать больше одного измерения. Фактически, пока функции среднего и ковариации могут обрабатывать многомерные входные данные, ГП может без труда работать в больших размерностях. В этом разделе мы узнаем, как обучать ГП на двумерном наборе данных.

Для этого будем следовать процедуре из предыдущего раздела. Во-первых, нужен набор обучающих данных. Здесь мы искусственно создадим фиктивный набор с точками (0, 0), (1, 2) и (–1, 1) с соответствующими метками 0, –1 и 0,5. Иначе говоря, исследуемая целевая функция имеет значение 0 в точке (0, 0), –1 в точке (1, 2) и 0,5 в точке (–1, 1). Сделаем прогнозы в пределах квадрата от [–3, 3] до [–3, 3].

В Python это можно настроить следующим образом:

```
#Обучающие данные
train_x = torch.tensor(
    [
        [0., 0.],
        [1., 2.],
        [-1., 1.]
    ]
)
```

```
train_y = torch.tensor([0., -1., 0.5])

#Тестовые данные
grid_x = torch.linspace(-3, 3, 101) #Одномерная сетка
grid_x1, grid_x2 = torch.meshgrid(grid_x, grid_x, indexing="ij")     #Двумерная сетка
xs = torch.vstack([grid_x1.flatten(), grid_x2.flatten()]).transpose(-1, -2)
```

Переменная `xs` — это матрица размером 10 201×2, содержащая решетки (сетку) по квадрату, на котором нужно сделать прогноз.

> **Важное примечание**
>
> Всего мы имеем 10 201 точку, потому что берем сетку из 101 конечной точки в каждом из двух измерений. Далее мы просто повторно запускаем код ГП, который ранее выполнялся для обучения, и делаем прогнозы на имеющемся двумерном наборе данных. Обратите внимание, что никаких изменений в классе `BaseGPModel` или каком-либо коде прогнозирования не требуется, и это весьма удивительно!

Однако следует изменить визуализацию прогнозов. Поскольку работа ведется в двух измерениях, построение прогностического среднего значения и ДИ на одном графике становится сложным. Типичное решение — построить одну тепловую карту для прогнозируемого среднего, а другую — для прогностического стандартного отклонения. Несмотря на то что последнее не совсем соответствует 95%-ному ДИ, эти два объекта, по сути, количественно определяют одно и то же — неопределенность в отношении значений функции.

Вместо вызова `predictive_distribution.confidence_region()`, как раньше, теперь мы извлекаем прогностическое стандартное отклонение следующим образом:

```
predictive_stddev = predictive_distribution.stddev
```

Теперь, чтобы нарисовать тепловые карты, используем функцию `imshow()` из Matplotlib. Здесь нужно быть осторожными с формой прогнозов в `predictive_mean` и `predictive_stddev`. Каждый из них представляет собой тензор длиной 10 000, поэтому прежде, чем передать в функцию `imshow()`, их необходимо преобразовать в квадратную матрицу. Это можно сделать следующим образом:

```
fig, ax = plt.subplots(1, 2)

#Первая тепловая карта для прогнозируемого среднего
ax[0].imshow(
    predictive_mean.detach().reshape(101, 101).transpose(-1, -2),
    origin="lower",
    extent=[-3, 3, -3, 3]
)

#Вторая тепловая карта для прогнозируемого стандартного отклонения
ax[1].imshow(
    predictive_stddev.detach().reshape(101, 101).transpose(-1, -2),
    origin="lower",
    extent=[-3, 3, -3, 3]
)
```

Этот код создаст две тепловые карты, показанные на рис. 2.19.

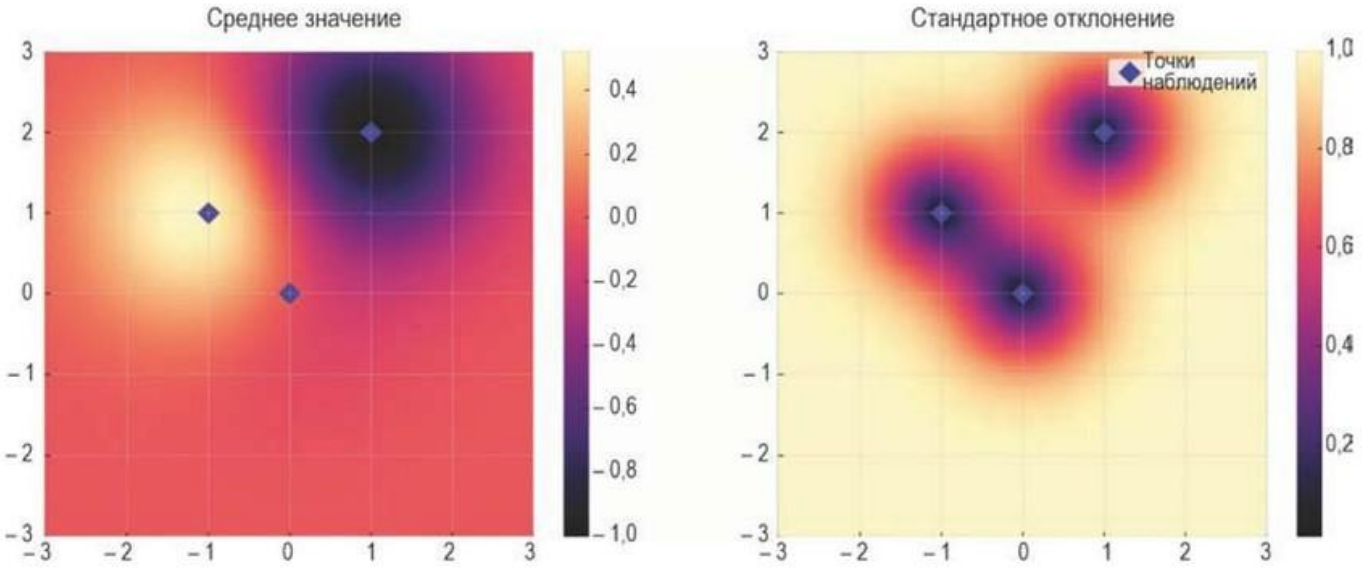


Рис. 2.19. Прогнозы двумерного ГП: средняя функция по-прежнему согласуется с данными обучения, и неопределенность снова исчезает в областях, окружающих точки данных

Теперь можно убедиться, что результат в одномерном примере распространяется и на двумерный сценарий:

- Левая часть показывает, что средний прогноз согласуется с данными обучения. Яркое пятно слева соответствует точке (–1, 1), которая имеет значение 0,5, а темное пятно справа соответствует точке (1, 2), которая имеет значение –1. Наблюдение в точке (0, 0) имеет значение 0, что также является априорным средним, поэтому указывать на левую часть не так важно.
- Неопределенность (измеренная прогностическим стандартным отклонением) близка к нулю вокруг трех точек обучающих данных, как показано в правой части рисунка. Относительно них стандартное отклонение плавно увеличивается до нормализованной максимальной неопределенности, равной 1.

Все полезные свойства ГП, такие как плавная интерполяция и количественная оценка неопределенности, сохраняются, когда выполняется переход к большему количеству измерений.

На этом можно завершить данную главу. В ней мы получили концептуальное понимание, что такое ГП, и научились реализовывать базовую модель на Python с помощью GPyTorch. Как уже упоминалось, в главе 3 мы углубимся в функции среднего и ковариации, включая их гиперпараметры, и увидим, как каждый из компонентов контролирует поведение модели.

2.5. Упражнение

В этом упражнении нужно обучить ГП на реальном наборе данных, который был представлен в главе 1 (повторно приведен в табл. 2.2). Каждая точка данных (строка) соответствует сплаву, созданному путем смешивания свинца (Pb), олова (Sn), германия (Ge) и марганца (Mn) в разных соотношениях (сами металлы называются исходными соединениями). Характеристики содержатся в первых четырех столбцах, которые представляют собой процентное содержание исходных соединений.

Цель прогнозирования — узнать минимальную температуру смешивания. Она указана в последнем столбце и обозначает самую низкую температуру, при которой может образоваться сплав. Задача состоит в том, чтобы спрогнозировать температуру смешивания с учетом процентного содержания сплава.

Таблица 2.2. Данные из задачи по поиску температуры смешивания металлов

% Pb	% Sn	% Ge	% Mn	Температура смешивания (°F)
0,50	0,50	0,00	0,00	192,08
0,33	0,33	0,33	0,00	258,30
0,00	0,50	0,50	0,00	187,24
0,00	0,33	0,33	0,33	188,54

Шаги для выполнения:

1. Создайте четырехмерный набор данных, включенный в табл. 2.2.
2. Нормализуйте пятый столбец, вычитая среднее из всех значений, и разделите затем результаты на их стандартное отклонение.
3. Рассмотрите первые четыре столбца как объекты, а пятый — как метку. Обучите ГП этим данным. Можно повторно использовать класс модели, который был реализован ранее в этой главе.
4. Создайте тестовый набор данных, содержащий составы с нулевым процентом германия и марганца. Другими словами, тестовый набор должен представлять собой сетку на единичном квадрате, оси которого отображают процентное содержание свинца и олова.

   Набор тестов должен выглядеть как тензор PyTorch:

```
tensor([[0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0100, 0.0000, 0.0000],
        [0.0000, 0.0200, 0.0000, 0.0000],
          ...,
        [1.0000, 0.9800, 0.0000, 0.0000],
        [1.0000, 0.9900, 0.0000, 0.0000],
        [1.0000, 1.0000, 0.0000, 0.0000]])
```

   Обратите внимание, что третий и четвертый столбцы — нулевые.
5. Спрогнозируйте температуру смешивания на данном тестовом наборе. А именно вычислите апостериорное среднее значение и стандартное отклонение нормализованной температуры смешивания для каждой точки тестового набора.
6. Визуализируйте предсказания. Это предполагает отображение среднего и стандартного отклонения в виде тепловых карт так же, как на рис. 2.19. Решение включено в `CH02/02 - Exercise.ipynb`.

Заключение

- Многомерное распределение Гаусса (MVN) моделирует совместное распределение многих случайных величин. Средний вектор обозначает ожидаемые значения переменных, а ковариационная матрица моделирует как дисперсии (отклонения), так и ковариации между этими переменными.
- Применяя теорему Байеса, можно вычислить апостериорное распределение. Благодаря этому переменные, похожие на наблюдаемую величину, обновляются и могут отобразить сходство с ней. В целом аналогичные величины дают схожие прогнозы.
- ГП расширяет MVN до бесконечных измерений, превращая его в распределение по функциям. Однако поведение ГП по-прежнему остается похожим на поведение MVN.
- Даже без каких-либо обучающих данных ГП все равно может делать апостериорные прогнозы из априорных.
- После обучения на наборе данных средний прогноз плавно интерполирует точки данных обучения.
- Одним из самых больших преимуществ ГП является калиброванная количественная оценка неопределенности, которую предлагает модель. Прогнозы на основе наблюдаемых точек данных более достоверны, а прогнозы, далекие от обучающих данных, более неопределенны.
- Обусловливание с помощью MVN или ГП визуально похоже на «завязывание узла» в точках наблюдения. Это заставляет модель точно проходить через них и снижает неопределенность в этих местах до нуля.
- При использовании GPyTorch для реализации ГП можно написать класс модели, который модульным образом расширяет базовый класс. В частности, это реализуется двумя конкретными методами: `__init__()`, который объявляет функции среднего и ковариации, и `forward()`, который создает распределение MVN для заданных входных данных.

3 Настройка ГП с помощью функций среднего значения и ковариации

Темы главы:

- Управление поведением ГП с помощью функции среднего.
- Управление гладкостью ГП с помощью функции ковариации.
- Исследование оптимальных гиперпараметров ГП с использованием градиентного спуска.

В главе 2 мы рассмотрели функции среднего и ковариации как основные компоненты гауссовского процесса. В реализации были использованы нулевое среднее и RBF-ядро, но существуют также и другие варианты. Делая конкретный выбор в пользу какого-либо из них в качестве компонентов ГП, мы фактически указываем предварительные (априорные) знания для него. Это крайне полезно делать с любой байесовской моделью, особенно в условиях, когда сбор данных обходится дорого (например, в БО).

Например, если стоит задача оценить температуру обычного январского дня в Миссури, никакие сложные вычисления не нужны, чтобы предположить, что температура будет довольно низкой. Или что летний день в Калифорнии будет относительно жарким. Такие грубые оценки могут использоваться в качестве первоначальных предположений в байесовской модели. Они, по сути, являются априорными знаниями для нее. Если бы их не было, тогда пришлось бы проводить более сложное моделирование для получения прогнозов.

Включение предварительных знаний в ГП может радикально изменить поведение модели и способно привести к повышению производительности прогнозирования (и в конечном итоге к более эффективному принятию решений). Не использовать предварительные знания уместно только в том случае, когда нет абсолютно никаких предположений о поведении функции (иначе это равносильно потере информации).

В этой главе мы обсудим различные варианты функций среднего и ковариации и то, как они влияют на результирующую модель ГП. В отличие от главы 2, здесь

будет применяться практический подход, а обсуждения будут основаны на реализации кода. К концу главы будет разработан конвейер, помогающий выбрать подходящие функции среднего и ковариации, а также оптимизировать их гиперпараметры.

3.1. Важность априорных знаний в байесовских моделях

Вопрос: почему нельзя изменить мнение некоторых людей? Ответ: из-за их приоритетов. Рассмотрим на следующем примере, насколько важны априорные знания в байесовской модели.

Допустим, вы «тусуетесь» со своими друзьями Бобом и Алис на карнавале и общаетесь с кем-то, кто утверждает, что она — экстрасенс. Она предлагает проверить это утверждение с помощью следующей процедуры: вы и ваши друзья будете загадывать число от 0 до 9, а «экстрасенс» скажет, о каком числе думает каждый из вас. Этот фокус можно повторять сколько угодно раз.

Вам становится интересно, и вы решаете провести этот тест 100 раз. Удивительно, но предполагаемый экстрасенс при каждой попытке правильно угадывает числа, которые каждый загадывает. Однако после окончания теста у всех участников будет разная реакция (рис. 3.1).

Дополнительная информация о теореме Байеса

В этой книге теорема представлена лишь в общих чертах. Если вы хотите изучить этот процесс более глубоко, я рекомендую главы 1 и 2 книги Уилла Курта *«Bayesian Statistics the Fun Way»* (*«Байесовская статистика: увлекательный путь»*, издательство No Starch Press, 2019).



Рис. 3.1. Реакция вас и ваших друзей, когда человек правильно угадал загаданное число 100 раз: каждый из вас пришел к разным выводам в зависимости от своего изначального (априорного) убеждения

Как можно наблюдать одно и то же событие, но прийти к разным выводам? Чтобы ответить на этот вопрос, рассмотрим процесс обновления убеждений с помощью теоремы Байеса:

1. Каждый из вас начинает со своим определенным изначальным убеждением по поводу того, является человек экстрасенсом или нет.
2. Затем вы наблюдаете, что номер каждого угадан правильно в первый раз.
3. Далее каждый для себя определяет вероятности различных вариантов. Например, вероятность того, что она действительно экстрасенс, верна и строго равна 1, поскольку настоящий экстрасенс всегда может пройти этот тест. Вероятность, что она не экстрасенс, равна 1 к 10, поскольку каждый раз вы случайным образом выбираете число от 0 до 9. Поэтому любое предположение из 10 чисел имеет равный шанс быть верным.
4. Наконец, ваши априорные предположения обновляются и становятся апостериорными с учетом изначальных убеждений и полученных вновь данных. Апостериорная вероятность будет пропорциональна произведению априорной вероятности и первого члена вероятности.
5. Затем процесс повторяется 100 раз, используя при этом апостериорную вероятность предыдущей итерации в качестве априорной для текущей итерации.

На высоком уровне здесь важно, что после каждого теста апостериорное убеждение вас и ваших друзей никогда не уменьшается, поскольку оно не согласуется с наблюдаемыми данными. На рис. 3.2 показана прогрессивная апостериорная вероятность каждого из друзей в зависимости от того, сколько тестов прошел «экстрасенс».

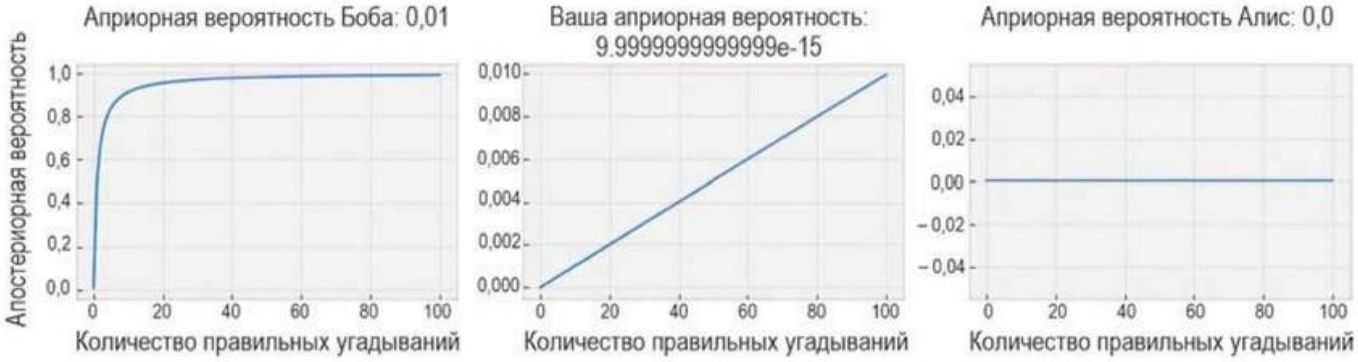


Рис. 3.2. Прогрессивная апостериорная вероятность того, что женщина на карнавале является экстрасенсом, как функция количества успешных догадок: она никогда не уменьшается, а ведет себя по-разному в зависимости от начальной априорной вероятности

Как видно, каждая из трех кривых либо увеличивается, либо остается ровной. Ни одна из них не снижается, поскольку это противоречило бы факту, что числа были угаданы верно 100 раз. Но почему кривые выглядят такими разными? Как можно догадаться, причиной является их начальное положение, т. е. априорная вероятность каждого, является ли человек экстрасенсом или нет.

В случае Боба (левая часть рисунка): он начал с относительно высокой вероятности, что некто является экстрасенсом (1%). По мере того как он наблюдает все

больше и больше данных, согласующихся с этим убеждением, его апостериорная вероятность также увеличивается.

Вы (середина рисунка), будучи скептиком, начали с гораздо более низкой априорной вероятности (1e-14). Однако наблюдения действительно предполагают, что человек является экстрасенсом. Поэтому апостериорная вероятность также увеличивается по мере поступления новых данных, достигая в конце 1%.

Случай Алис (правая часть рисунка) отличается. С самого начала она не верила, что экстрасенсы реальны, поэтому ее априорная вероятность равна 0. Согласно теореме Байеса апостериорная вероятность пропорциональна априорной, умноженной на вероятность. Поскольку априорная вероятность Алис равна нулю, в результате произведение также дает ноль. Поэтому, даже после 100 успешных угадываний результат останется прежним. Картина соответствует правилу байесовского обновления, но поскольку априорная вероятность Алис не допускает возможности существования экстрасенсов, никакие данные не смогут убедить ее в обратном.

Это подчеркивает важный аспект байесовского обучения — априорный подход будет определять, как осуществляется обучение (рис. 3.3):



Рис. 3.3. Как разные предварительные убеждения каждого человека обновляются одними и теми же данными: по сравнению с априорным показателем Боба, ваш намного ниже и увеличивается медленнее, приоритет Алис равен 0 и остается неизменным на протяжении всего процесса

Помимо примера, байесовская процедура обновления применима также ко всем ситуациям, где есть вероятностное убеждение о каком-либо событии, которое часто обновляется в свете новых данных. Фактически именно по этой причине мы иногда не можем изменить чье-то мнение даже перед лицом неопровержимых доказательств: они начинают с нулевой априорной вероятностью, и ничто не может поменять их убеждение.

Эта дискуссия интересна с философской точки зрения. Она показывает, что убедить кого-то в чем-либо можно, имея ненулевую изначальную вероятность. Данный пример показывает важность наличия хороших априорных знаний для байесовской модели. Как уже упоминалось, предварительные знания ГП уточняются с помощью функций среднего значения и ковариации. Каждый выбор приводит к различному поведению в прогнозе.

3.2. Включение предварительных знаний в ГП

В этом разделе мы рассмотрим сценарии, в которых важно указать предварительные знания для ГП.

Априорный ГП изначально может иметь постоянные среднее значение и ДИ. Затем он обновляется для плавной интерполяции наблюдаемых точек данных, как показано на рис. 3.4. Другими словами, средний прогноз точно проходит через точки данных, а 95%-ный ДИ в этих регионах исчезает.

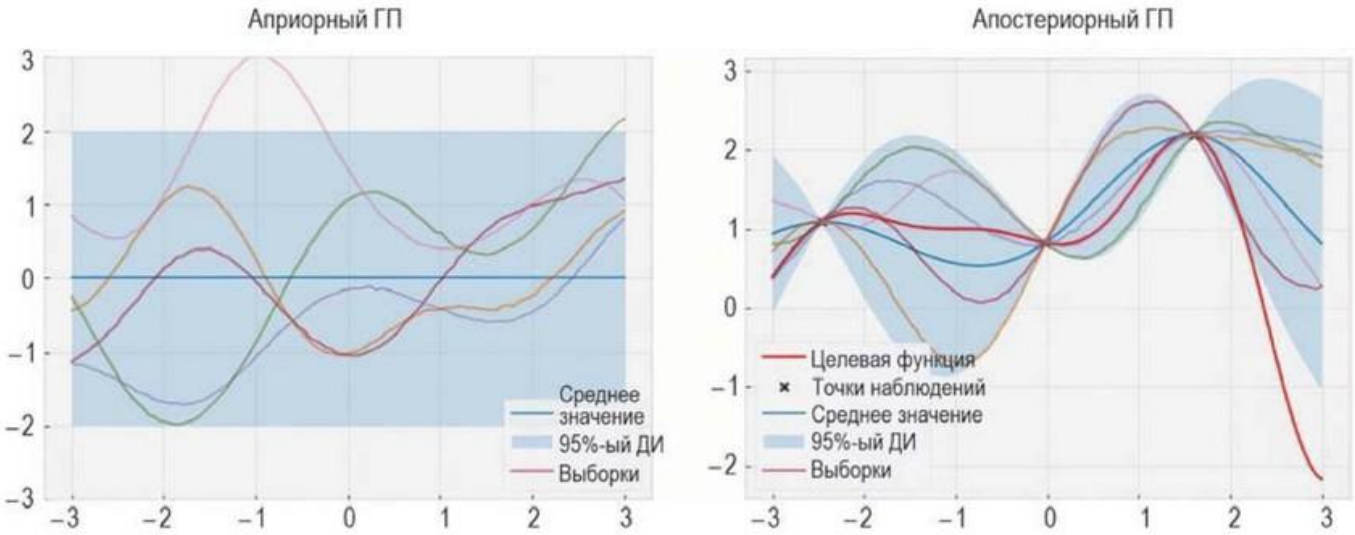


Рис. 3.4. Сравнение априорного и апостериорного ГП: априорный содержит изначальную информацию о целевой функции, а апостериорный объединяет эту информацию с фактическими наблюдениями

Априорный ГП на рисунке не имеет никаких предположений относительно моделируемой целевой функции, поэтому средний прогноз везде равен нулю. Однако во многих случаях, даже если мы не знаем точную форму целевой функции, она имеет аспекты, которые нам известны.

Рассмотрим следующее:

- Когда моделируется точность модели при настройке гиперпараметров, мы знаем, что диапазон целевой функции находится в пределах от 0 до 1.
- В примере с ценами на жилье из раздела 2.1 значения функции являются строго положительными, а также должны возрастать при увеличении какого-либо наблюдаемого свойства дома (например, жилая площадь).
- В примере с жильем: значение функции более чувствительно к одним характеристикам, чем к другим. Например, цена дома увеличивается быстрее в зависимости от количества этажей, чем в зависимости от жилой площади. Один дополнительный этаж увеличивает цену дома больше, чем один дополнительный квадратный фут.

Такая информация является именно тем изначальным знанием, которое важно представить с помощью ГП. Также одним из самых больших преимуществ его использования является то, что включить предварительные знания можно многими способами. Это помогает сократить разрыв между суррогатом ГП и фактической

целевой функцией, которую он моделирует, что также будет более эффективно определять оптимизацию в дальнейшем.

Объединение предварительных знаний и ГП

Мы учитываем предварительные знания, выбирая подходящие функции среднего и ковариации и устанавливая значения их параметров. Во-первых, функция среднего определяет ожидаемое поведение целевой функции. Во-вторых, функция ковариации определяет структуру целевой функции (если точнее, взаимосвязь между любой парой точек данных), а также насколько быстро и плавно та изменяется в своей области.

Каждый из предшествующих вариантов приводит к совершенно разному поведению итогового ГП. Например, линейная функция среднего приводит к линейному поведению прогнозов, а квадратичная — к квадратичному соответственно. Используя различные параметры в ковариационной функции, также можно контролировать изменчивость ГП.

3.3. Определение поведения целевой функции с помощью функции среднего значения

Для начала рассмотрим функцию среднего, которая определяет ожидаемое поведение ГП или то, как, по нашему мнению, целевая функция выглядит в среднем во всех возможных сценариях ее использования. Это поможет указать любые предварительные знания, связанные с общим поведением и формой функции. Код, используемый в этом разделе, включен в файл `CH03/01-Meanfunctions.ipynb`. Внесем больше конкретики и используем набор данных с ценами на жилье, который включает пять точек данных в табл. 3.1.

Таблица 3.1. Пример обучающего набора данных: объект прогнозирования (цена) увеличивается в зависимости от его свойства (жилой площади)

Жилая площадь (в квадратных футах, умноженная на 1000)	Цена (в долларах, умноженная на 100 000)
0.5	0.0625
1	0.25
1.5	0.375
3	2.25
4	4

В этом наборе моделируемые значения функции представляют собой цены на жилье. Они строго положительные и возрастают с увеличением жилой площади. Даже не зная цен, интуитивно понятно, что они также обладают этими свойствами.

Цель здесь — включить свойства в функцию среднего, поскольку они описывают ожидания относительно того, как функция будет себя вести. Прежде чем перейти

непосредственно к моделированию, сначала стоит написать вспомогательную функцию, которая принимает модель ГП (вместе с ее функцией вероятности) и визуализирует ее прогнозы в диапазоне от 0 до 10 (жилая площадь — 10 000 квадратных футов). Это реализуется следующим образом:

```
def visualize_gp_belief(model, likelihood):
    #Вычисляем прогнозы
    with torch.no_grad():
        predictive_distribution = likelihood(model(xs))
        predictive_mean = predictive_distribution.mean
        predictive_upper, predictive_lower = predictive_distribution.
          confidence_region()

    plt.figure(figsize=(8, 6))

    plt.plot(xs, ys, label="objective", c="r")
    plt.scatter(train_x, train_y, marker="x", c="k", label="observations")

    #Отображаем линию среднего
    plt.plot(xs, predictive_mean, label="mean")
    plt.fill_between(
        xs.flatten(), predictive_upper, predictive_lower, alpha=0.3,
        label="95% CI"
    ) #Отображаем область 95%-ного ДИ

    plt.legend(fontsize=15);
```

Как работает данный код, мы видели в разделе 2.4.3. Поместим его в удобную функцию. Теперь можно реализовать модель ГП и посмотреть, как выбор повлияет на получаемые прогнозы.

3.3.1. Использование функции нулевого среднего в качестве базовой стратегии

Простейшая форма среднего значения — это постоянная нулевая функция. При отсутствии данных она выдает ноль в качестве прогноза по умолчанию. Она используется, когда нет дополнительной информации о целевом объекте, который можно было бы включить в ГП в качестве предварительных знаний.

ГП с функцией нулевого среднего значения реализуется следующим образом:

```
class ConstantMeanGPModel(gpytorch.models.ExactGP):
    #Постоянная функция среднего с нулевым значением по умолчанию
    def __init__(self, train_x, train_y, likelihood):
      super().__init__(train_x, train_y, likelihood)
      self.mean_module = gpytorch.means.ConstantMean()
      self.covar_module = gpytorch.kernels.RBFKernel()
```

```
    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

Для построения модели с помощью GPyTorch (раздел 2.4.2) мы реализовали методы `__init__()` и `forward()`. В первом мы инициализируем функции среднего и ковариации, а во втором пропускаем через них входные данные `x` и возвращаем соответствующее многомерное гауссово распределение.

> **Примечание**
>
> Вместо класса `gpytorch.means.ZeroMean` в реализации из раздела 2.4.2 используется класс `gpytorch.means.ConstantMean` для инициализации функции среднего. Однако она имеет значение по умолчанию, равное нулю. Поэтому фактически реализуется идентичная модель ГП. Несмотря на это, в данной главе мы рассмотрим, как класс `pytorch.means.ConstantMean` позволяет корректировать постоянное среднее значение и добиться лучшей модели.

Инициализируем объект этого класса, обучим на наших данных и визуализируем его прогнозы. Это можно сделать следующим образом:

```
lengthscale = 1
noise = 1e-4

#Объявляем ГП
likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = ConstantMeanGPModel(train_x, train_y, likelihood)

#Настраиваем гиперпараметры
model.covar_module.lengthscale = lengthscale
model.likelihood.noise = noise

model.eval()
likelihood.eval()

visualize_gp_belief(model, likelihood)
```

Здесь выполняется инициализация модели и устанавливаются ее гиперпараметры: масштаб длины и дисперсия шума, равные 1 и 0,0001 соответственно. Позже в этой главе мы рассмотрим, как правильно установить их значения, а пока просто будем придерживаться имеющихся. В конце вызывается только что написанная вспомогательная функция `visualize_gp_belief()`, результат которой можно увидеть на рис. 3.5.

Все свойства ГП, указанные в разделе 2.4.4, по-прежнему здесь:

- Апостериорная функция среднего плавно интерполирует значения `xs`, которые являются точками обучающих данных.
- 95%-ный ДИ исчезает вокруг этих точек, что указывает на хорошо откалиброванную количественную оценку неопределенности.

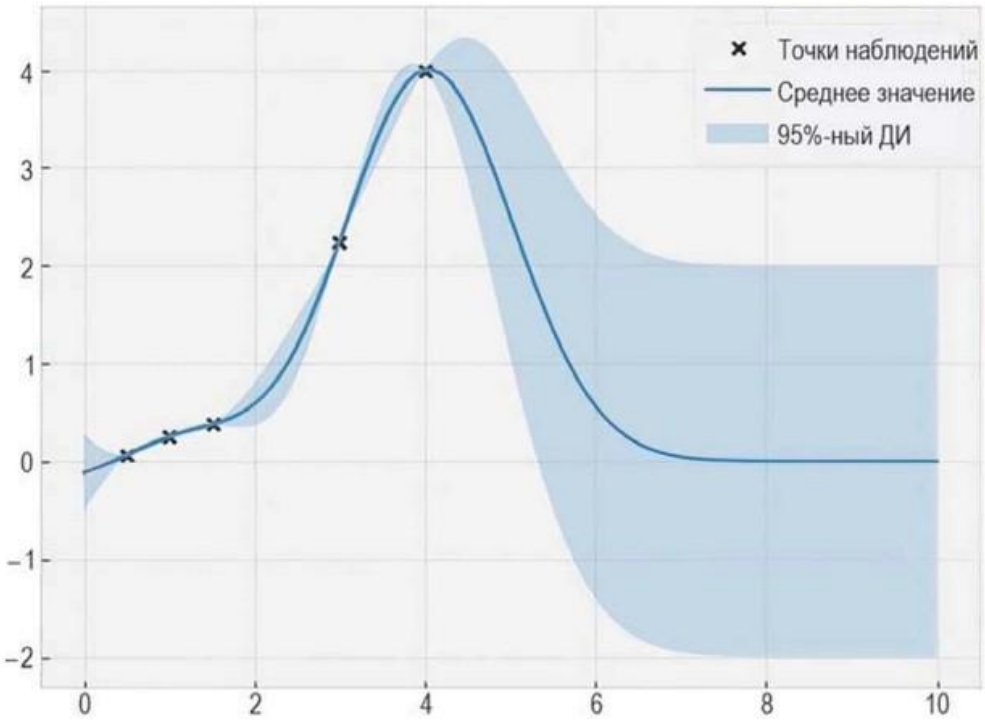


Рис. 3.5. Прогнозы ГП с функцией нулевого среднего: апостериорная функция среднего интерполирует наблюдаемые точки данных и возвращается к нулю в регионах, которые находятся далеко от наблюдений

На этом графике также можно заметить, что если уйти достаточно далеко от точек данных (правая часть графика), апостериорная функция среднего возвращается к предыдущему среднему значению, которое равно нулю. Это важная особенность ГП: при отсутствии данных (в регионах без наблюдений) априорная функция среднего является основной движущей силой процедуры вывода. Это имеет смысл, поскольку без реальных наблюдений лучшее, что может сделать прогностическая модель, просто обратиться к априорным знаниям, заданным в ее функции среднего.

> **Примечание**
>
> На этом этапе становится понятно, почему так важно иметь четко определенные предварительные знания. Они — это единственное, что управляет прогнозами при отсутствии данных.

Тогда возникает естественный вопрос: возможно ли использовать функцию ненулевого среднего, чтобы вызвать другое поведение ГП в этих неисследованных регионах; и если да, то как это можно реализовать? Оставшаяся часть этого раздела призвана ответить на данный вопрос, и начнем мы с использования постоянной функции среднего, которая не равна нулю.

3.3.2. Использование постоянной функции с градиентным спуском

Постоянная функция среднего (отличная от нуля) подходит, когда есть ожидание, что моделируемая целевая функция примет некоторый диапазон значений, известный априори. Поскольку мы моделируем цены на жилье, использование постоянной функции среднего с константой больше нуля имеет смысл (ожидается, что цены будут положительными).

Конечно, во многих случаях невозможно узнать, какое значение в среднем принимает целевая функция. Как тогда найти для нее подходящее значение? Используемая стратегия заключается в обращении к определенной величине: насколько вероятен набор обучающих данных, учитывая значение средней функции. Грубо говоря, эта величина показывает, насколько хорошо модель объясняет данные обучения. В этом разделе мы рассмотрим, как использовать такую величину с целью выбрать лучшую среднюю функцию для ГП.

Как установить гиперпараметры для ГП?

Нужно выбирать значения, которые лучше всего объясняют данные!

Если вероятность обучающих данных при некотором значении c_1 выше, чем при другом некотором значении c_2, то предпочтительнее использовать первый вариант. Это дает количественную оценку при использовании ненулевых функций среднего для моделирования положительных функций. Постоянная функция среднего с положительным значением объясняет наблюдения (исключительно для положительной функции) лучше, чем функция, значение которой равно нулю (или отрицательному).

Как можно вычислить такую вероятность? GPyTorch предлагает удобный класс `gpytorch.mlls.ExactMarginalLogLikelihood`, который принимает модель ГП и вычисляет *предельную логарифмическую вероятность* ее обучающих данных с учетом гиперпараметров.

Рассмотрим рис. 3.6, чтобы увидеть эффективность вероятностной величины при количественной оценке соответствия данных. На нем изображены прогнозы, сделанные с помощью двух отдельных моделей ГП: с нулевым средним из предыдущего раздела (слева) и с функцией среднего, значение которой равно 2 (справа). Обратите внимание, как справа функция среднего возвращается к 2 вместо 0. Здесь

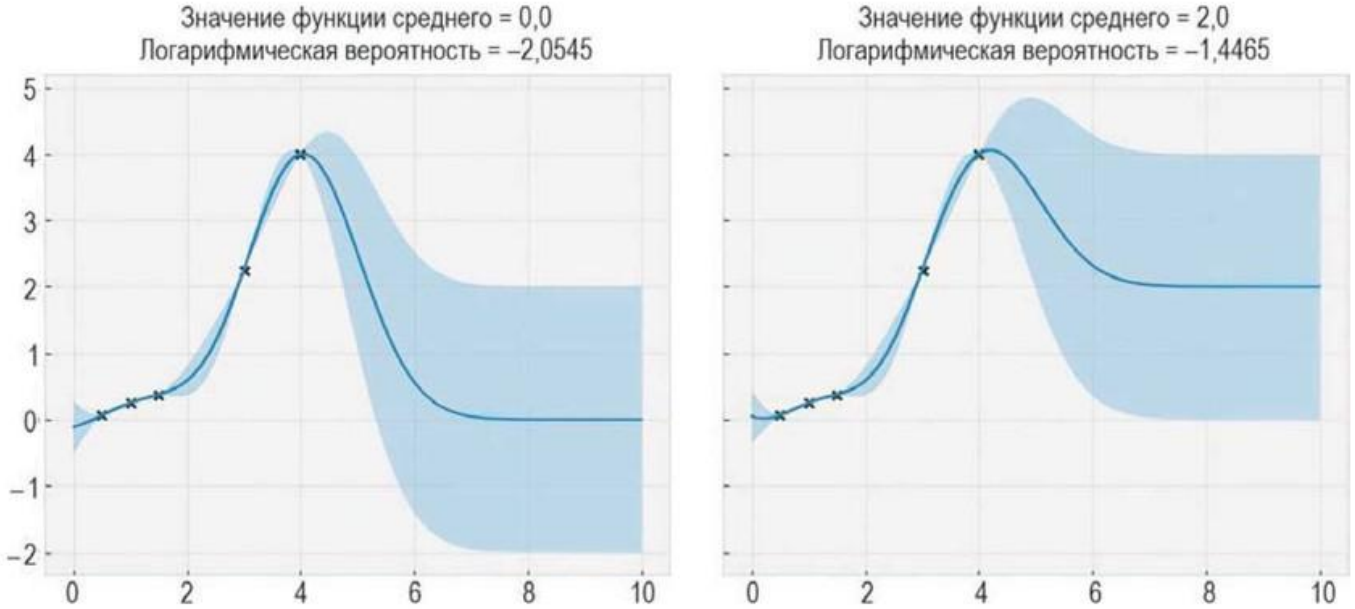


Рис. 3.6. Прогнозы ГП с учетом двух разных постоянных функций среднего: значение 2 дает более высокую вероятность, чем 0, указывая на то, что первая функция среднего лучше подходит, чем вторая

второй ГП имеет более высокую логарифмическую вероятность, чем первый. Это означает, что значение 2 объясняет данные обучения лучше, чем значение 0.

> **Замечание**
>
> При работе со значениями вероятности часто используется логарифмическая шкала, которая позволяет многим скрытым вычислениям быть более стабильными в числовом отношении.

Вычислив логарифмическую вероятность, последним шагом будет просто найти значение функции среднего, при котором та будет максимальной. Мы стремимся найти среднее значение, которое лучше всего объясняет данные обучения. Поскольку есть доступ к вычислению логарифмической вероятности, можно использовать алгоритмы оптимизации на основе градиента (например, градиентный спуск) для итеративного уточнения имеющегося среднего значения. После сходимости мы придем к хорошему среднему значению, обеспечивающему высокую вероятность получения данных.

> **Примечание**
>
> Если нужно освежить знания о том, как работает градиентный спуск, вы можете обратиться к приложению «B» в книге Луиса Серрано *«Grokking Machine Learning»* (*«Машинное обучение Грокинга»*, издательство Manning, 2021), которое хорошо объясняет эту концепцию.

Теперь рассмотрим, как можно реализовать этот процесс с помощью кода. Поскольку модель ГП для функции среднего в нашем примере реализована с помощью класса `gpytorch.means.ConstantMean`, нам не нужно здесь ничего менять. Снова инициализируем модель:

```
#Объявляем ГП
lengthscale = 1
noise = 1e-4

likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = ConstantMeanGPModel(train_x, train_y, likelihood)

#Настраиваем гиперпараметры
model.covar_module.lengthscale = lengthscale
model.likelihood.noise = noise
```

Центральным шагом этой процедуры является определение функции логарифмической вероятности и алгоритма градиентного спуска. Как упоминалось ранее, первая является экземпляром класса `gpytorch.mlls.ExactMarginalLogLikelihood`, который реализуется следующим образом:

```
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)
```

В качестве градиентного спуска мы будем использовать «Adam» — современный алгоритм, который пользуется большим успехом во многих задачах машинного обучения (особенно DL). Он объявляется с помощью PyTorch следующим образом:

```
optimizer = torch.optim.Adam([model.mean_module.constant], lr=0.01)
```

Обратите внимание, что мы переходим к классу `torch.optim.Adam`, представляющему собой среднее значение, которое мы стремимся оптимизировать. Когда процедура градиентного спуска запущена, Adam итеративно обновляет значение `model.mean_module.constant`, чтобы улучшить функцию вероятности.

Последнее, что осталось сделать, — это запустить градиентный спуск, который реализуется так:

```
#Включаем режим обучения
model.train()
likelihood.train()

losses = []
constants = []
for i in tqdm(range(500)):
    optimizer.zero_grad()

    #loss (потеря) – отрицательная предельная логарифмическая вероятность
    output = model(train_x)
    loss = -mll(output, train_y)

    #Градиентный спуск по потерям
    loss.backward()
    losses.append(loss.item())
    constants.append(model.mean_module.constant.item())
    optimizer.step()

#Включаем режим прогнозирования
model.eval()
likelihood.eval()
```

Вызовы `train()` в начале и `eval()` в конце — это формальные обязательные шаги, которые включают режимы обучения и прогнозирования соответственно. Сброс градиентов на каждом этапе с помощью `optimizer.zero_grad()` — это еще одна обязательная составляющая, позволяющая убедиться, что градиенты вычисляются верно.

В середине у нас есть процедура градиентного спуска из 500 шагов, в которой итеративно вычисляется потеря `loss` (является отрицательным значением логарифмической вероятности) и выполняется спуск к ней на основе ее градиентов. Во время цикла `for` мы отслеживаем полученные отрицательные значения логарифмической вероятности, а также среднее значение, корректируемое на каждом шаге. Это сделано, чтобы после обучения можно было визуально проверить значения и определить, сошлись ли они.

На рис. 3.7 представлена визуализация, показывающая текущие значения отрицательной логарифмической вероятности (которую мы стремимся минимизировать) и значение функции среднего. Потери постоянно уменьшаются (и это хорошо!) с увеличением значения средней константы, показывая, что положительная константа действительно дает более высокую вероятность, чем ноль. Обе кривые

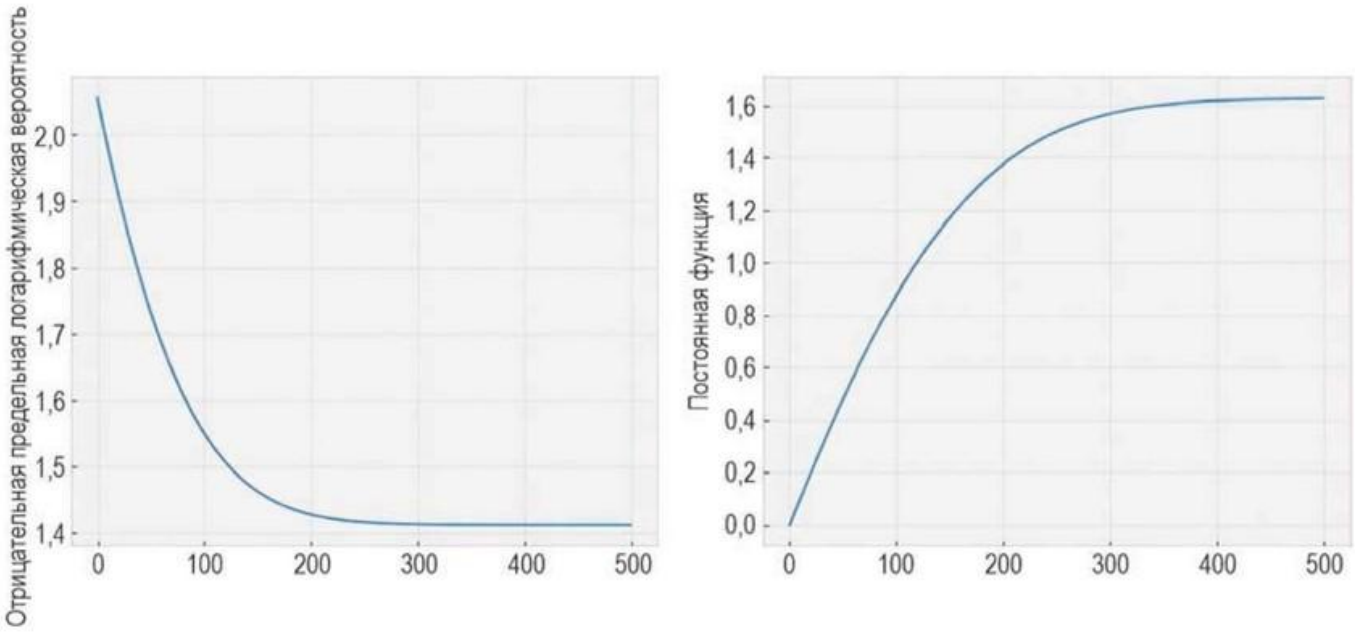


Рис. 3.7. Вероятность отрицательного логарифма (чем ниже, тем лучше) и среднее значение во время градиентного спуска: на обеих частях рисунка значения сошлись, что указывает на то, что мы достигли оптимума

выходят на плато после 500 итераций. Это указывает на то, что оптимальное значение средней константы достигнуто.

> **Примечание**
>
> Хорошая рекомендация — всегда рассчитывать прогрессирующие потери (как мы только что сделали) при использовании градиентного спуска, чтобы отслеживать, достигнуто ли оптимальное значение. Остановка до сходимости может привести к снижению производительности модели. Мы не будем снова демонстрировать подобные графики потерь в этой главе, но прилагаемый код их включает.

На данный момент мы научились использовать нулевую функцию среднего значения по умолчанию, а также оптимизировать значение постоянной функции среднего с учетом вероятности данных. Однако во многих случаях можно заранее знать, как должна вести себя целевая функция, и поэтому включить больше информации в функцию среднего.

Например, как можно реализовать идею, что цена дома возрастает, зная что жилая площадь возрастает? Дальше мы научимся делать это с помощью ГП, используя линейную или квадратичную функцию среднего.

3.3.3. Использование линейной функции с градиентным спуском

Линейная функция среднего имеет вид $\mu = \omega^{\mathrm{T}} x + b$, где μ — прогностическое среднее значение в контрольной точке x, ω — весовой вектор, объединяющий коэффициенты для каждого свойства в точке x, а b — член постоянного смещения.

Используя ее, мы кодируем предположение, что ожидаемое поведение целевой функции равно линейной комбинации свойств точки данных x. В примере с ценами на жилье мы рассматриваем только один *признак* — жилую площадь. И ожидается,

что он будет иметь положительный вес, поэтому с увеличением жилой площади модель будет прогнозировать более высокую цену.

С другой стороны мы имеем модель линейной регрессии, которая также предполагает, что целевая метка представляет собой линейную комбинацию свойств. И мы помещаем вероятностное убеждение (модель ГП) поверх прогнозов. Это дает возможности, сохраняя при этом все преимущества моделирования с помощью ГП (а именно количественную оценку неопределенности).

> **Примечание**
>
> При постоянной функции среднего весовой вектор ω фиксируется на нулевом векторе, а смещение b — это среднее значение, которое мы научились оптимизировать в предыдущем разделе. Другими словами, линейная функция представляет собой более общую модель, чем постоянная функция среднего.

Что касается реализации, построение модели с линейной функцией среднего является довольно простым. Постоянное среднее значение заменяется экземпляром `gpytorch.means.LinearMean` следующим образом (метод `forward()` остается неизменным):

```
class LinearMeanGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
    super().__init__(train_x, train_y, likelihood)
    self.mean_module = gpytorch.means.LinearMean(1) #Линейное среднее
    self.covar_module = gpytorch.kernels.RBFKernel()
```

Здесь используется аргумент `(1)` для инициализации модуля среднего. Он указывает, что мы работаем с одномерной целевой функцией. При работе с многомерной функцией можно просто передать ее размерность. Помимо этого, все остальное в модели аналогично тому, что было раньше. Подгоняя и обучая новую модель на трехточечном наборе данных, мы получаем прогнозы, представленные на рис. 3.8.

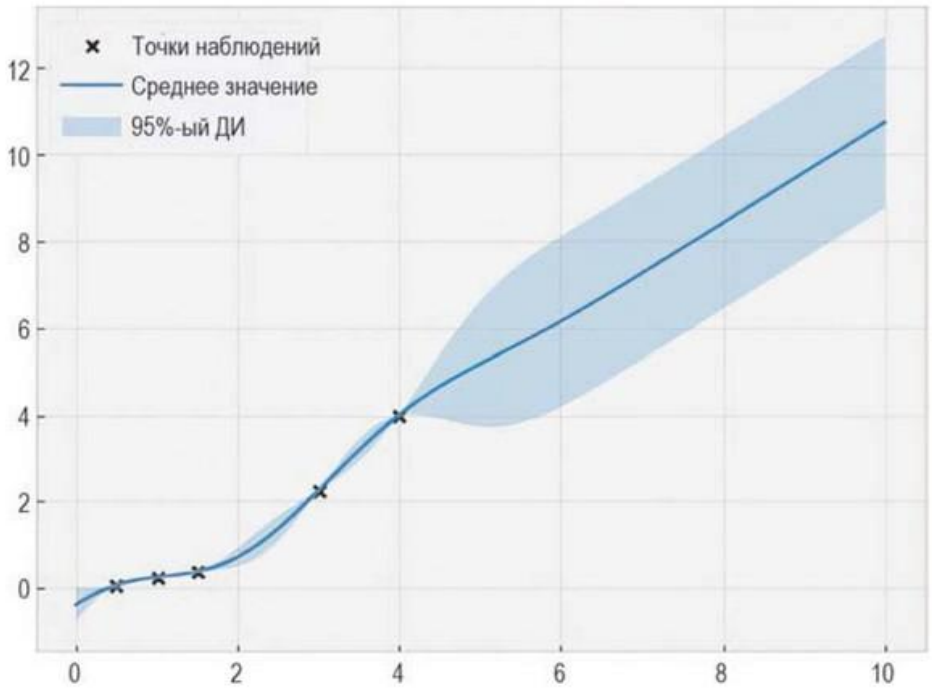


Рис. 3.8. Прогнозы ГП с линейной функцией среднего: ГП имеет тенденцию к росту, что является прямым результатом положительного наклона линейной функции среднего

В отличие от постоянной, линейная функция среднего приводит к тому, что вся модель имеет восходящую тенденцию. Это связано с тем, что наиболее подходящая линия из пяти точек в обучающих данных имеет положительный наклон. Это именно то, что нужно было смоделировать (взаимосвязь между жилой площадью и ценой).

3.3.4. Использование квадратичной функции с помощью реализации пользовательской функции среднего

Линейная функция среднего успешно показывает тенденцию роста цены, но предполагается, что скорость ее роста постоянна. Однако во многих случаях можно заранее знать, что целевая функция увеличивается с непостоянной скоростью, которую линейное среднее не может моделировать. Фактически точки данных, которые использовались, были сгенерированы таким образом, что цена является квадратичной функцией жилой площади. Вот почему видно, что большие дома дорожают быстрее, чем маленькие. В этом разделе мы реализуем средний ГП как квадратичную функцию.

На момент написания книги GPyTorch предоставляет реализации только для постоянных и линейных функций среднего значения. Но красота этого инструмента заключается в его модульности. Все компоненты модели, такие как функции среднего и ковариации, стратегия прогнозирования и даже функция предельной логарифмической вероятности, реализованы как модули. Следовательно, они могут быть изменены, переопределены и расширены объектно-ориентированным способом. Это можно увидеть воочию, реализуя собственную функцию квадратичного среднего значения.

Первое, что нужно сделать, это определить класс функции среднего:

```
class QuadraticMean(gpytorch.means.Mean):
    def __init__(self, batch_shape=torch.Size(), bias=True):
        ...
    def forward(self, x):
        ...
```

Он расширяет класс `gpytorch.means.Mean`, который является основой для всех реализаций функции среднего в GPyTorch. Чтобы создать пользовательскую логику, нужно переписать два метода: `__init__()` и `forward()`, которые аналогичны тем, какие использованы в модели.

В `__init__()` нужно объявить, какие параметры содержит функция среднего. Этот процесс называется *регистрацией параметров*.

В то время как линейная функция имеет два параметра (наклон и точку пересечения с осью координат Y), квадратичная имеет три: коэффициент для члена второго порядка x^2, коэффициент для члена первого порядка x и коэффициент для члена нулевого порядка, который обычно называется *смещением*. Это показано на рис. 3.9.

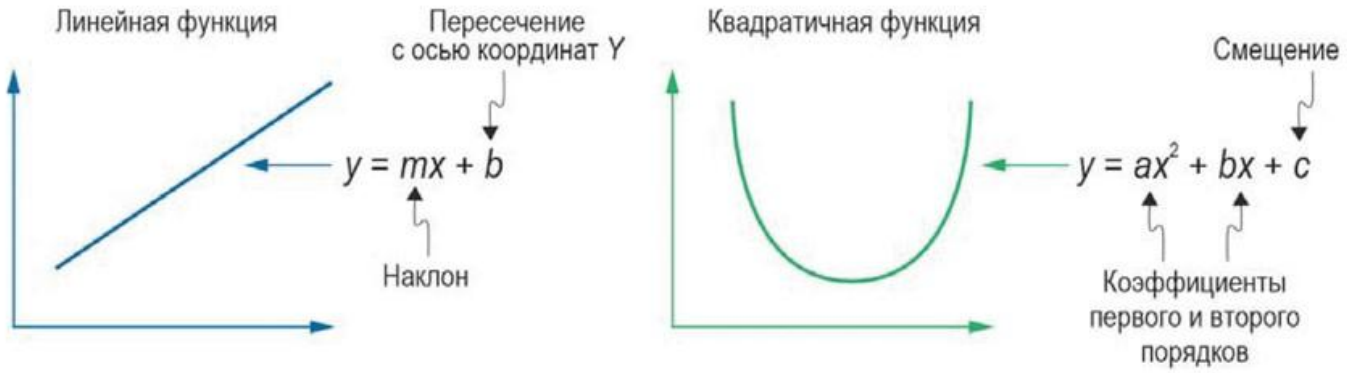


Рис. 3.9. Функциональные формы линейной и квадратичной функций: линейная имеет два параметра, а квадратичная — три; когда они используются в качестве функции среднего для ГП, соответствующие параметры являются гиперпараметрами ГП

Имея это в виду, реализуем метод `__init__()` следующим образом:

```
class QuadraticMean(gpytorch.means.Mean):

    #Коэффициент второго порядка
    def __init__(self, batch_shape=torch.Size(), bias=True):
        super().__init__()
        self.register_parameter(
        name="second",
        parameter=torch.nn.Parameter(torch.randn(*batch_shape, 1, 1))
    )

    #Коэффициент первого порядка
    self.register_parameter(
        name="first",
        parameter=torch.nn.Parameter(torch.randn(*batch_shape, 1, 1))
    )

    #Смещение
    if bias:
        self.register_parameter(
            name="bias",
            parameter=torch.nn.Parameter(torch.randn(*batch_shape, 1))
    )
    else:
        self.bias = None
```

Функция `register_parameter()` последовательно вызывается для регистрации коэффициентов первого и второго порядков, а также смещения. Поскольку нет четкого представления, какие значения они должны принимать, просто инициализируем их случайным образом с помощью `torch.randn()`.

> **Примечание**
>
> Эти параметры необходимо зарегистрировать как экземпляры класса `torch.nn.Parameter`, который позволяет корректировать (обучать) их значения во время градиентного спуска.

Для метода `forward()` нужно определить, как функция среднего должна обрабатывать входные данные. Как мы уже говорили, квадратичная функция имеет вид $ax^2 + bx + c$, где a, b и c — коэффициент второго порядка, коэффициент первого порядка и смещение соответственно. Поэтому нужно только реализовать данную логику следующим образом:

```
class QuadraticMean(gpytorch.means.Mean):
    def __init__(self, train_x, train_y, likelihood):
        ... #Опустим детали

    def forward(self, x):
        res = x.pow(2).matmul(self.second).squeeze(-1)
            + x.matmul(self.first).squeeze(-1) #Формула квадратичной функции
        if self.bias is not None:
            res = res + self.bias
        return res
```

Имея функцию квадратичного среднего, теперь можно написать модель ГП, инициализирующую свой модуль среднего значения с помощью специального класса `QuadraticMean`, который мы только что реализовали:

```
class QuadraticMeanGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super().__init__(train_x, train_y, likelihood)
        self.mean_module = QuadraticMean()
        self.covar_module = gpytorch.kernels.RBFKernel()

    def forward(self, x):
        ... #Опустим детали
```

Повторное выполнение всей процедуры обучения с градиентным спуском даст прогнозы, представленные на рис. 3.10.

Этот ГП успешно моделирует неравномерную скорость роста цен на жилье в зависимости от жилой площади. Прогнозы растут гораздо быстрее на правой стороне графика, чем на левой.

Это можно сделать для функции любой формы, поведение которой нужно предположить относительно целевой функции (например, для полинома более высокой степени или сублинейной функции). Все, что нужно сделать, это реализовать класс функции среднего с соответствующими параметрами и использовать градиентный спуск для присвоения им значений, которые хорошо подойдут для обучающих данных.

Модели ГП имеют хорошую математическую гибкость, поскольку могут использовать функцию среднего для любой структуры и при этом давать вероятностные прогнозы. Такая гибкость мотивирует и стимулирует разработку GPyTorch, чей упор на модульность помогает без особых усилий расширять и реализовывать собственные функции среднего значения. Ту же гибкость и модульность можно наблюдать в ковариационных функциях GPyTorch, о которых мы поговорим далее.

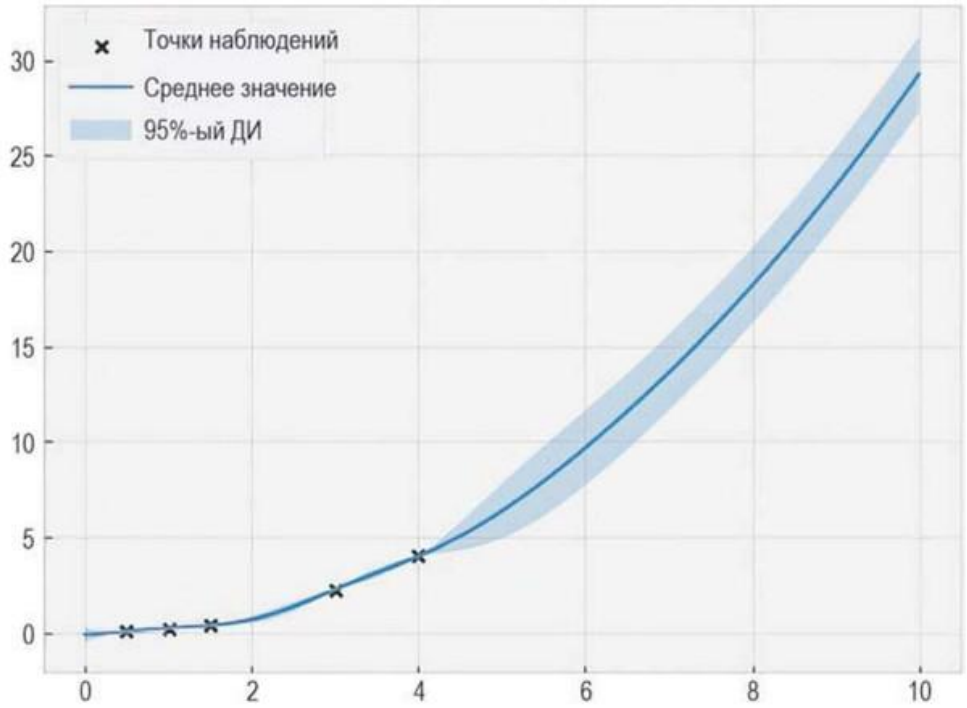


Рис. 3.10. Прогнозы ГП с функцией квадратичного среднего: чем больше жилая площадь, тем с большей скоростью будет расти цена

3.4. Определение изменчивости и гладкости с помощью ковариационной функции

В то время как функция среднего определяет ожидание относительно общего поведения целевого объекта, функция ковариации (ядро ГП) играет более сложную роль: определяет взаимосвязь между точками данных внутри области, а также контролирует структуру и гладкость гауссовского процесса. В этом разделе мы сравним, как меняются прогнозы при изменении различных компонентов модели. Благодаря этому мы получим практическое представление о том, как выбрать подходящую ковариационную функцию для модели. Используемый код находится в `CH03/02 – Covariance functions.ipynb`.

В этих примерах в качестве целевого объекта используется функция Форрестера, которую мы задействовали ранее в разделе 2.4.1. Мы случайным образом выбираем три точки данных между −3 и 3 и используем их в качестве набора обучающих данных. Все прогнозы, представленные здесь, сделаны ГП на основе выбранных точек.

3.4.1. Настройка масштабов функции ковариации

Первый способ управлять поведением ГП, используя его ковариационную функцию, заключается в том, чтобы установить *масштаб длины* и *масштаб вывода*. Они являются гиперпараметрами ковариационной функции (как константы или коэффициенты в функции среднего):

- Масштаб длины контролирует размер входных данных и, следовательно, насколько быстро ГП может изменяться вдоль оси. Иными словами, как целевая функция варьируется в зависимости от размерности входных данных.
- Масштаб вывода определяет диапазон выходных данных ГП или, другими словами, диапазон его прогнозов.

Устанавливая для масштабов различные значения, можно либо увеличить, либо уменьшить неопределенность в прогнозах, а также масштабировать их диапазон. Сделать это можно так:

```
class ScaleGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super().__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ZeroMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(
          gpytorch.kernels.RBFKernel()) #gpytorch.kernels.ScaleKernel реализует
                                        #масштаб вывода


    def forward(self, x):
      ... # Опустим некоторые детали
```

Обратите внимание, код для атрибута `covar_module` здесь отличается от предыдущего: мы размещаем объект `gpytorch.kernels.ScaleKernel` вне обычного RBF-ядра. Это позволяет эффективно масштабировать выходные данные RBF-ядра с некоторым постоянным коэффициентом. С другой стороны, масштаб длины уже включен в `gpytorch.kernels.RBFKernel`:

```
#ScaleKernel умножает свои входные данные на выходной масштаб
#RBFKernel уже имеет реализованкую шкалу длины
gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())
```

В используемом коде есть одна строка, в которой устанавливается масштаб длины ядра:

```
model.covar_module.base_kernel.lengthscale = lengthscale
```

Используя аналогичный API, можно установить масштаб вывода ядра с помощью:

```
model.covar_module.outputscale = outputscale
```

Теперь убедиться, что масштаб длины эффективно контролирует скорость изменения функции, можно, сравнивая прогнозы, сделанные двумя ГП (один с масштабом длины 1, а другой с масштабом длины 0,3), как показано на рис. 3.11.

Резкая разница между двумя частями рисунка ясно показывает эффект масштаба длины:

- Более короткий масштаб длины соответствует большей изменчивости целевой функции при постоянном изменении входных данных.
- С другой стороны, длинный масштаб делает функцию более плавной, и она имеет меньшую изменчивость при том же входном значении.

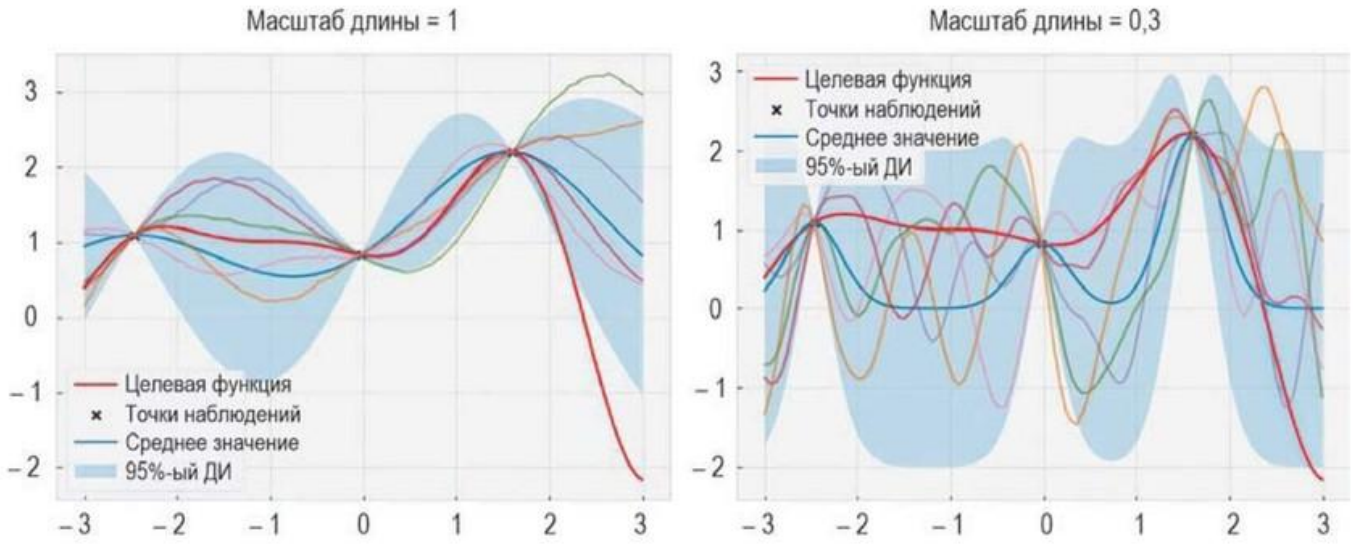


Рис. 3.11. Прогнозы ГП с масштабом длины, установленным в значения 1 (слева) и 0,3 (справа): при небольшом масштабе длины прогнозы имеют большую изменчивость, что приводит к большей неопределенности

Например, если сместиться на одну единицу по оси X, выборки на левой части рис. 3.11 будут различаться меньше, чем на правой.

> **Замечание**
>
> Чем больше изменчивость, тем больше неопределенность. Это означает, что более короткий масштаб длины приводит к более широкому достоверному интервалу. Так происходит потому, что мы меньше уверены в поведении функции.

А как насчет масштаба вывода? Ранее мы говорили, что этот параметр масштабирует выходные данные ковариационной функции в другой диапазон прогнозов. Это делается путем простого умножения выходных данных на данный параметр. Следовательно, большой масштаб приводит к расширению диапазона прогнозов ГП, тогда как малый их сужает. Чтобы убедиться в этом, еще раз запустим код и восстановим прогнозы, на этот раз установив масштаб вывода равным 3. Полученный результат показан на рис. 3.12.

Несмотря на то что левые части рисунков 3.11 и 3.12 могут выглядеть одинаково (поскольку ГП и его выборки имеют одинаковую форму на двух графиках), можно заметить, что на рис. 3.12 ось Y больше, поскольку и ее прогнозы, и ее выборки принимают более высокие значения (как отрицательные, так и положительные). Это прямой результат масштабирования значений ковариации из RBF-ядра с большим масштабом вывода.

Имея всего два гиперпараметра для ковариационной функции, мы убедились, что можно объяснить широкий диапазон поведения функции, моделируемой с помощью ГП. Общие выводы приведены в табл. 3.2. Вы можете перезапустить код с разными значениями масштабов длины и вывода, чтобы увидеть их эффект и проверить таблицу самостоятельно.

> **Примечание**
>
> Такая гибкость моделирования порождает естественный вопрос: как правильно задать значения этих гиперпараметров? К счастью, уже есть хороший способ их установки.

Это можно сделать, выбрав значения, которые лучше всего объясняют данные, или, другими словами, максимизируя предельную логарифмическую вероятность (в частности, с помощью градиентного спуска).

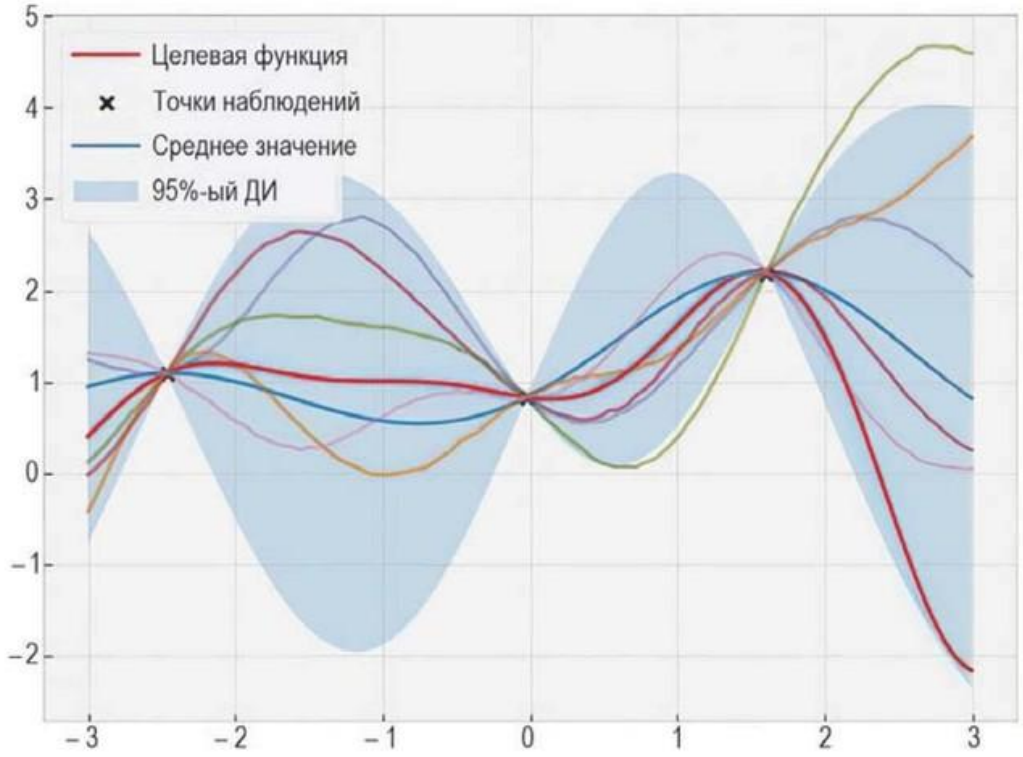


Рис. 3.12. Прогнозы ГП с масштабом вывода, равным 3: при большом масштабе ГП моделирует функцию с более широким диапазоном, что также допускает больше неопределенности в прогнозах

Таблица 3.2. Краткий итог, какие роли могут играть масштабы длины и вывода в ГП

Параметр	С большим значением	С малым значением
Масштаб длины	Более плавные прогнозы, меньше неопределенности	Большая изменчивость, больше неопределенности
Масштаб вывода	Большие выходные данные, больше неопределенности	Более узкий диапазон выходных значений, меньше неопределенности

Точно так же (когда мы хотели оптимизировать гиперпараметры функции среднего) можно просто передавать переменные, которые нужно оптимизировать (параметры ковариационной функции) в алгоритм Adam:

```
optimizer = torch.optim.Adam(model.covar_module.parameters(), lr=0.01)
```

Запустив градиентный спуск, можно получить хорошие значения этих параметров. Например, в моем случае получились значения 1,3 для масштаба длины и 2,1 для масштаба вывода. Для хорошего соответствия набору обучающих данных из трех точек нужно, чтобы ГП был более гладким (с масштабом длины больше 1), а также чтобы диапазон прогнозов был больше (с большим масштабом вывода). Это, безусловно, обнадеживающий результат, поскольку наша целевая функция имеет широкий диапазон значений. На входе с масштабом 3 она принимает значение –2, что находится далеко за пределами ДИ с масштабом вывода 1.

3.4.2. Управление гладкостью с помощью различных ковариационных функций

В качестве ковариационной функции пока что мы использовали RBF. Однако если она не подходит, вполне возможно использовать и другое ядро. В этом разделе мы научимся использовать другое семейство ядер (а именно *ядро Матерна*), а также посмотрим, какой эффект оно окажет на ГП.

> **Примечание**
>
> Используя ядро Матерна, мы задаем гладкость функции моделей. *Гладкость* (**Smoothness**) — это технический термин, обозначающий дифференцируемость функции. Чем больше раз функция дифференцируема, тем она более гладкая. Грубо говоря, какими скачками значения функции «прыгают» вверх и вниз.

Ядро RBF моделирует функции, которые являются бесконечно дифференцируемыми. Этим свойством обладают немногие функции в реальном мире. Между тем ядро Матерна порождает функции, которые конечно дифференцируемы. Сколько раз их можно дифференцировать (насколько гладкими они являются), контролируется настраиваемым параметром, который мы вскоре обсудим.

Прежде чем увидеть ядро Матерна в действии, сначала переопределим класс модели ГП:

```
class MaternGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood, nu):
        super().__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ZeroMean()
        self.covar_module = gpytorch.kernels.MaternKernel(nu)

    def forward(self, x):
      ... #Опустим некоторые детали
```

Здесь атрибут `covar_module` инициализируется как экземпляр класса `gpytorch.kernels.MaternKernel`. Параметр `nu` определяет уровень плавности, который также является параметром метода `__init__()`.

> **Важное примечание**
>
> На момент написания книги GPyTorch поддерживает три значения параметра `nu`: 1/2, 3/2 и 5/2, что соответствует функциям, которые являются недифференцируемыми, дифференцируемыми однократно и дифференцируемыми дважды. Чем больше этот параметр, тем более гладким будет ГП.

Для начала попробуем `nu = 0,5`, установив его значение при инициализации ГП:

```
likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = MaternGPModel(train_x, train_y, likelihood, 0.5)

... #Исправляем гиперпараметры и включаем режим прогнозирования

visualize_gp_belief(model, likelihood)
```

Результат выполнения кода изображен на рис. 3.13.

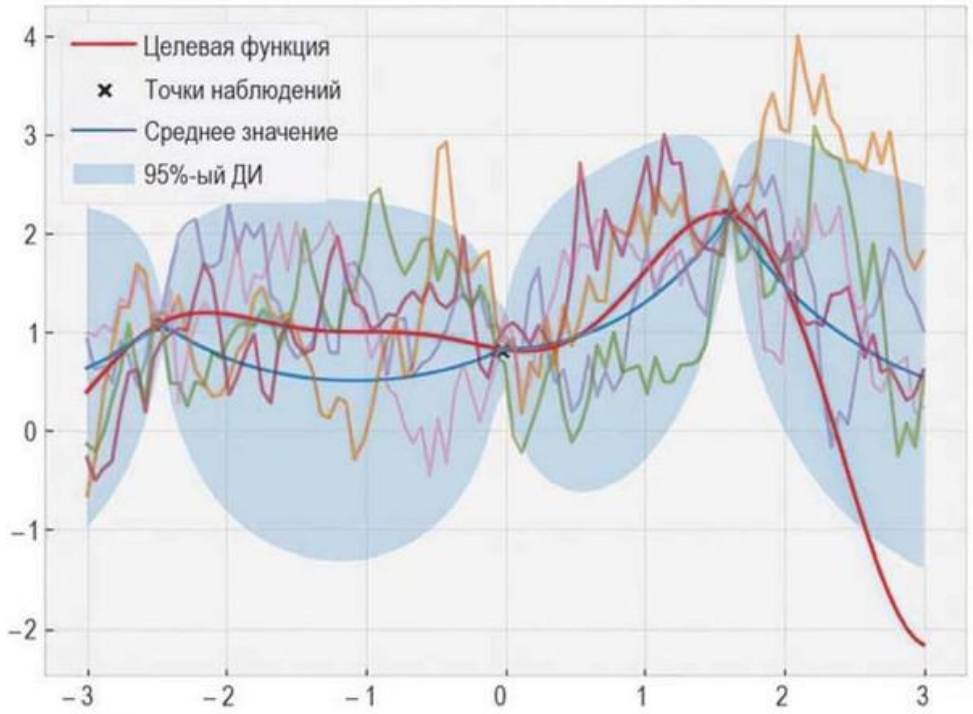


Рис. 3.13. Прогнозы ГП с помощью ядра Матерна со значением 1/2: значение определяет убеждение, что целевая функция не дифференцируется, что соответствует очень грубым выборкам

В отличие от сценария с RBF, все выборки из ядра Матерна являются очень неровными («зубчатыми»). На самом деле ни одна из них не является дифференцируемой. Параметр `nu = 0,5` — хорошее значение при моделировании данных временных рядов (например, цены на акции).

Однако это значение обычно не используется в БО, поскольку зубчатые функции (подобные изображенным на рис. 3.13) очень изменчивы (они могут прыгать вверх и вниз непредсказуемым образом) и обычно не являются целью для автоматической оптимизации. Нужен определенный уровень гладкости целевой функции. В противном случае эффективная оптимизация станет нереальной.

Обычно предпочитают ядро Матерна 5/2. Его прогнозы (а также прогнозы при значении 3/2), показаны на рис. 3.14.

Можно видеть, что выборки из ядра 5/2 более гладкие, что приводит к более эффективному обучению ГП. Однако эти примеры достаточно грубы и напоминают функции, которые можно было бы увидеть в реальном мире. По этой причине в БО при большинстве попыток, как исследовательских, так и прикладных, используют ядро Матерна 5/2. В будущих главах (при обсуждении принятия решений для БО) мы по умолчанию будем использовать именно это ядро.

> **Примечание**
>
> Несмотря на то что мы не включаем сюда соответствующие детали, ядро Матерна имеет свои собственные масштабы длины и вывода, которые можно указать для дальнейшей настройки поведения результирующего ГП таким же образом, как и в предыдущем разделе.

Объединяя функцию среднего с ядром, можно вызвать сложное поведение в предсказаниях ГП. Точно так же, как априорное убеждение влияет на вывод, сделанный

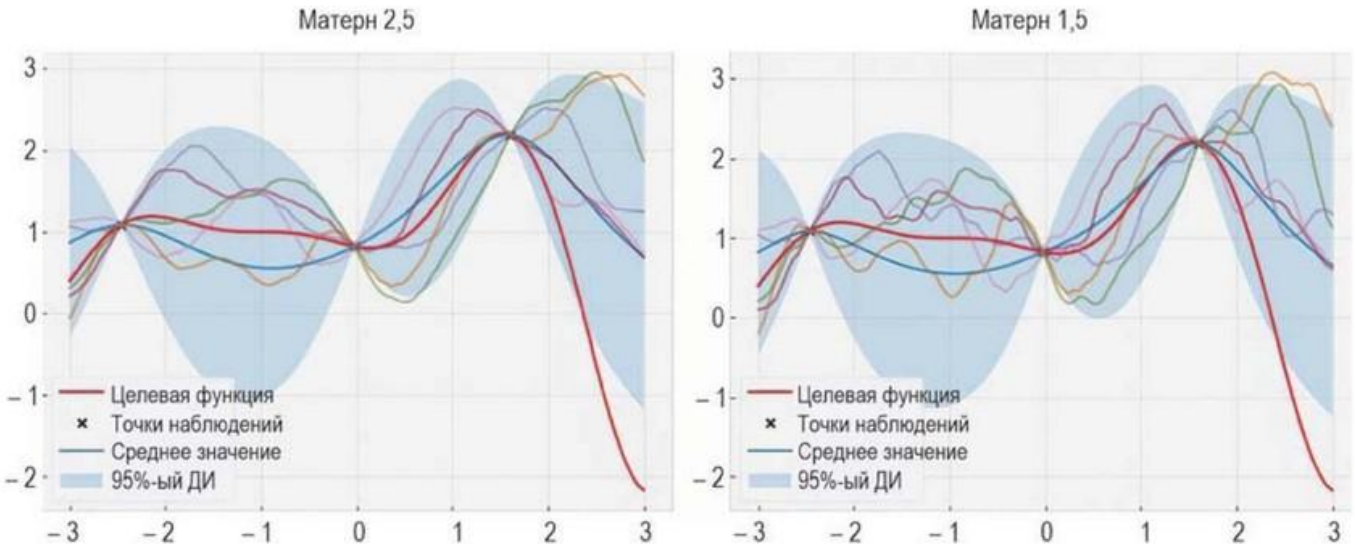


Рис. 3.14. Прогнозы ГП с ядрами Матерна 5/2 (слева) и 3/2 (справа): выборки здесь достаточно гладкие, чтобы ГП мог эффективно учиться на данных, но также достаточно неровные, чтобы реалистично моделировать процессы

на основе 100 угадываний числа, выбор функции среднего и ядра определяет прогнозы ГП. На рис. 3.15 показаны три примера, в которых каждая комбинация функции среднего и ядра приводит к совершенно разному поведению.

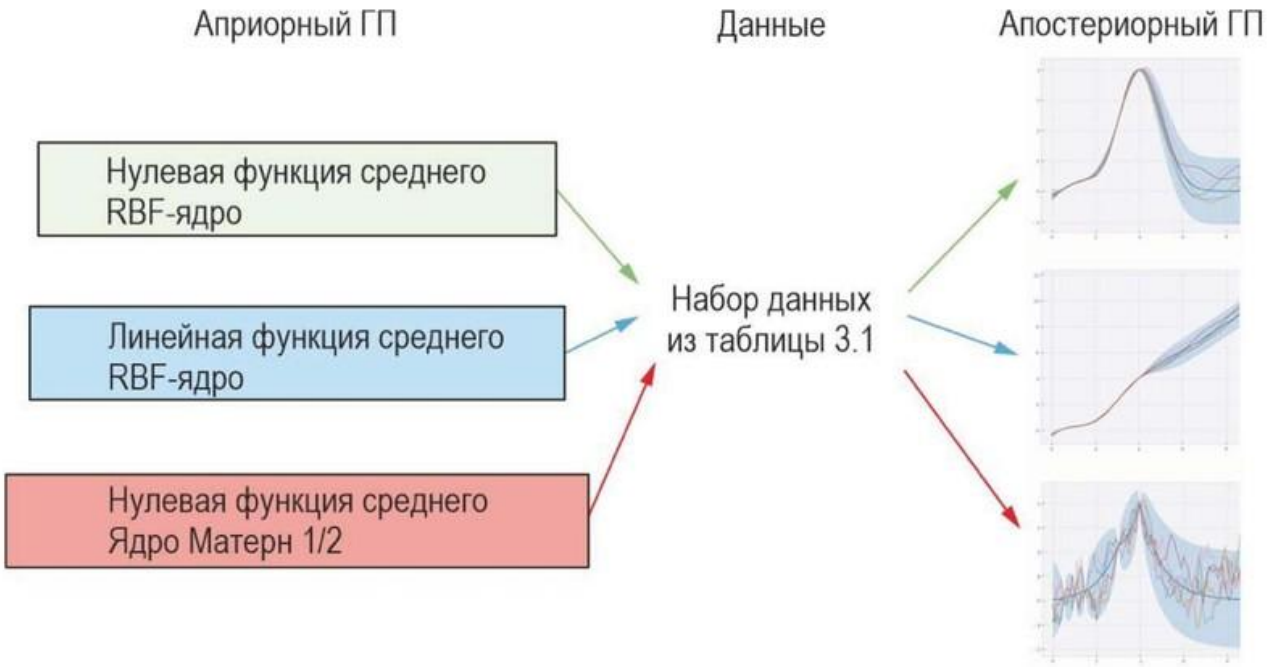


Рис. 3.15. Три разных варианта сочетания функции среднего и ядра, а также прогнозы, сделанные соответствующими апостериорными ГП при обучении на одном и том же наборе данных: каждый вариант приводит к различному поведению в прогнозировании

3.4.3. Моделирование различных уровней изменчивости с помощью множественных масштабов длины

В одномерных целевых функциях (входные данные которых имеют один признак) необходимо учитывать только один масштаб длины. Однако есть сценарии, когда

многомерная целевая функция (входные данные которой имеют более одного признака) имеет более сильную изменчивость в одних измерениях и более гладкую в других. Иначе говоря, некоторые измерения имеют малые масштабы, а другие — большие. Вернемся к примеру в начале этой главы: прогнозируемая цена дома увеличивается в большей степени при наличии дополнительного этажа, чем при наличии дополнительного квадратного фута жилой площади. В этом разделе мы будем исследовать, как поддерживать несколько масштабов длины в ГП для моделирования функций.

Если бы для всех измерений использовался только один масштаб длины, точно смоделировать целевую функцию было бы невозможно. Такая ситуация требует, чтобы модель ГП поддерживала отдельный масштаб длины для каждого измерения с целью полностью отразить изменчивость в каждом из них. В этом разделе мы узнаем, как это можно сделать с помощью GPyTorch.

В качестве примера используется конкретная двумерная целевая функция Экли, которую можно модифицировать, чтобы она имела различные уровни изменчивости в разных измерениях. Реализуем функцию следующим образом:

```
def ackley(x):
    return -20 * torch.exp(
        -0.2 * torch.sqrt((x[:, 0] ** 2 + x[:, 1] ** 2) / 2)
    )
        - torch.exp(torch.cos(2 * pi * x[:, 0] / 3)
        + torch.cos(2 * pi * x[:, 1]))
```

Мы специально ограничиваем область определения этой функции квадратной областью между –3 и 3 в обоих измерениях (обычно обозначается как $[-3, 3]^2$). Для визуализации используется тепловая карта на рис. 3.16.

Каждое темное пятно можно рассматривать как «впадину» с низким значением на поверхности целевой функции. По оси *Y* гораздо больше впадин, чем по оси *X*. Это указывает на то, что второе измерение более изменчиво, чем первое. Целевая функция имеет больше подъемов и спадов по оси *Y*, чем по оси *X*.

Это означает, что использование только одного масштаба длины для обоих измерений — не лучший выбор. Более эффективно будет иметь его для каждого измерения (в данном случае их два). После каждый из них можно оптимизировать независимо с помощью градиентного спуска.

> **Важное примечание**
>
> Использование ядра с масштабом длины для каждого измерения называется *автоматическим определением релевантности* (**Automatic Relevance Determination**, **ARD**). Этот термин обозначает факт, что после использования градиентного спуска для оптимизации масштабов длины можно сделать вывод, насколько актуально каждое измерение целевой функции по отношению к ее значениям. Измерение с большим масштабом длины имеет низкую изменчивость и, следовательно, менее актуально при моделировании значений целевой функции, чем измерение с малым масштабом длины.

Реализовать ARD с помощью GPyTorch очень просто: нужно указать параметр `ard_num_dims`, передающий количество измерений, которые имеет целевая функция

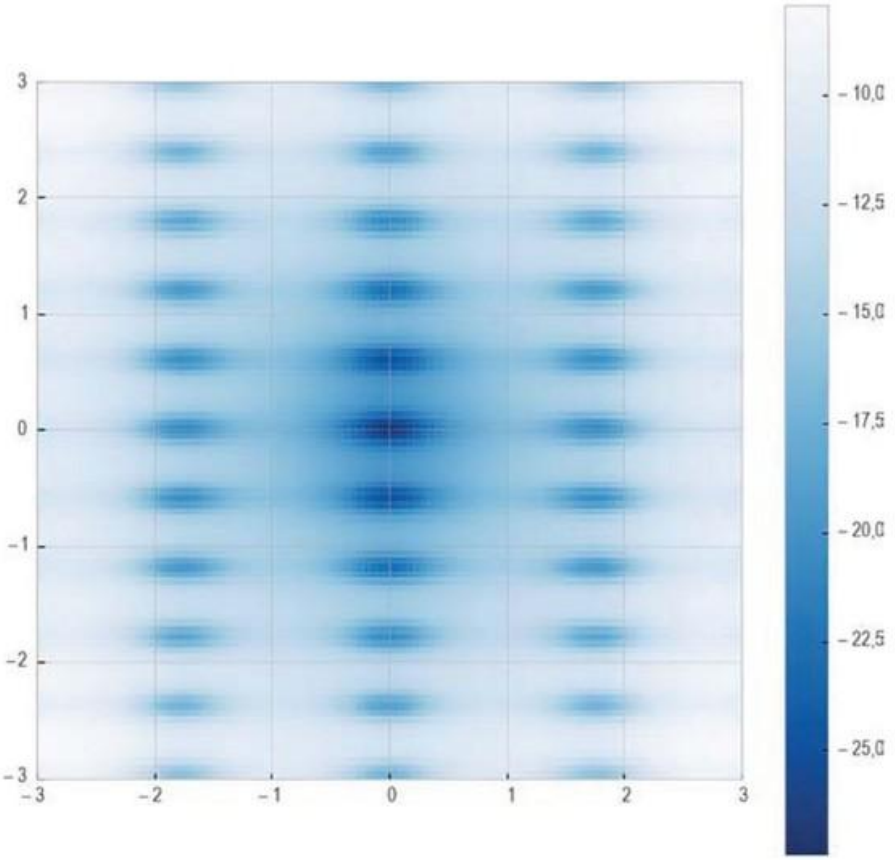


Рис. 3.16. Двумерная функция Экли в качестве целевой: ось *X* имеет меньшую изменчивость (она меняется меньше), чем ось *Y*, что требует разных масштабов длины

при инициализации ковариационной функции. Это делается с помощью ядра RBF следующим образом:

```
class ARDGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super().__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ZeroMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.RBFKernel(ard_num_dims=2)
        )

    def forward(self, x):
        ... #Опустим некоторые детали
```

Рассмотрим, даст ли эта модель при обучении функции Экли разные масштабы длины для двух измерений. Для этого сначала создадим случайный набор обучающих данных, состоящий из 100 точек:

```
torch.manual_seed(0)
train_x = torch.rand(size=(100, 2)) * 6 - 3
train_y = ackley(train_x)
```

После обучения с использованием градиентного спуска можно проверить оптимизированные значения для масштабов длины. Это можно сделать, выведя на печать следующее:

```
>>> model.covar_module.base_kernel.lengthscale
tensor([[0.7175, 0.4117]])
```

Это действительно тот результат, которого мы ожидали: больший масштаб длины для первого измерения, где вариативность значений функции меньше, и меньший масштаб длины для второго измерения, где вариативность больше.

> **ПРИМЕЧАНИЕ**
>
> Если функция имеет множество входных измерений с одинаковым уровнем изменчивости, соответствующие масштабы длины, скорее всего, будут оптимизированы для получения схожих значений.

Больше информации о ядрах

Ядра сами по себе вызвали большой интерес со стороны ML-сообщества. В дополнение к тому, что мы рассмотрели к текущему моменту, следует отметить, что они также могут кодировать сложные структуры, такие как периодичность, линейность и шум. Для более подробного и технического обсуждения ядер и их роли в машинном обучении можно обратиться к книге *«The Kernel Cookbook»* Дэвида Дювено.

На этом можно завершить обсуждения в главе 3. В ней мы подробно исследовали, как на модель ГП влияют функции среднего и ковариации, в частности их различные параметры. Научились включать априорную информацию в модели. Также использовали градиентный спуск для оценки значений этих параметров, чтобы получить модель, которая лучше всего объясняет данные.

На этом также можно завершить первую часть книги, в которой мы сосредоточились на ГП. Со следующей главы начинается изучение второго компонента структуры БО — *принятие решений*. Мы начнем с двух наиболее часто используемых политик, направленных на улучшения: *вероятность улучшения* и *ожидаемое улучшение*.

3.5. Упражнение

В этом упражнении представлена практика реализации модели ГП с помощью ARD. Для этого нужно создать целевую функцию, которая меняется вдоль одной из осей больше, чем вдоль другой. Затем необходимо обучить модель (как с помощью ARD, так и без) на точках данных этой функции и сравнить полученные значения масштаба длины. Решение включено в `CH03/03 - Exercise.ipynb`.

Этот процесс состоит из нескольких этапов:

1. Реализуйте следующую двумерную функцию в Python с помощью PyTorch:

$$f(x_1, x_2) = \sin\left(\frac{5x_1 - 5}{2}\right)\cos(2.5 - 5x_2) + \frac{1}{10}\left(\frac{5x_2 + 1}{2}\right)^2$$

   Она моделирует поверхность модели для *машины опорных векторов* (**Support-Vector Machine**, **SVM**) в задаче по настройке гиперпараметров. Ось X обозначает значение штрафного параметра c, а ось Y обозначает значение параметра γ

ядра RBF. Эта функция также будет использоваться в качестве целевой в будущих главах.

2. Визуализируйте функцию в области $[0, 2]^2$. Тепловая карта должна выглядеть так, как на рис. 3.17.
3. Случайным образом извлеките 100 точек данных из области $[0, 2]^2$. Они будут использоваться в качестве обучающих данных.
4. Создайте модель ГП с постоянной функцией среднего и ядром Матерна 5/2, который имеет масштаб вывода, реализованной как объект `gpytorch.kernels.ScaleKernel`.
5. Не указывайте параметр `ard_num_dims` при инициализации объекта ядра и не устанавливайте для него значение `None`. Это создаст модель ГП без ARD.
6. Настройте гиперпараметры модели ГП, используя градиентный спуск, и проверьте масштаб длины после обучения.
7. Переопределите класс модели, на этот раз установив `ard_num_dims = 2`. Переобучите модель с градиентным спуском и убедитесь, что оба масштаба длины имеют существенно разные значения.

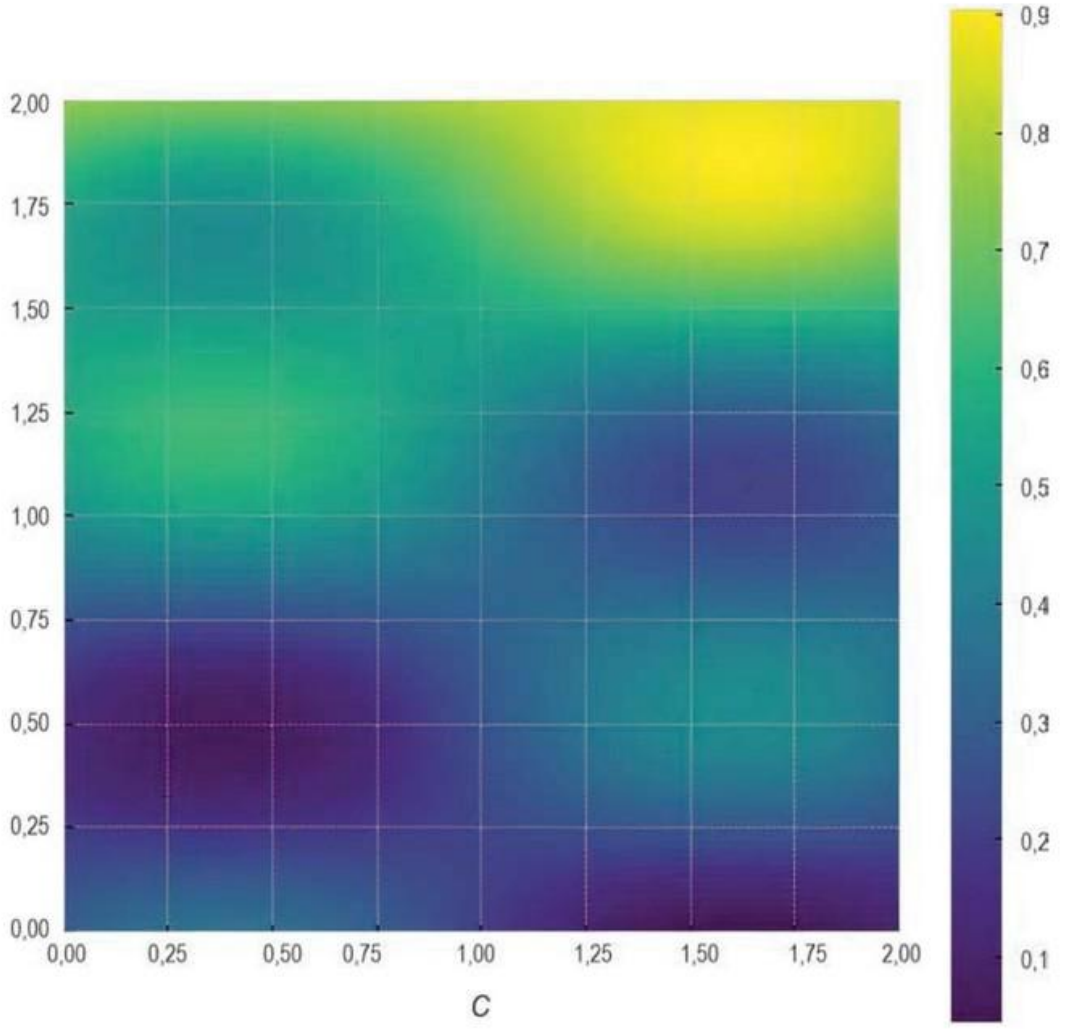


Рис. 3.17. Точность SVM-модели на тестовом наборе данных как функция штрафного параметра (*c*) и параметра RBF-ядра (*γ*): функция меняется быстрее по *γ*, чем по *c*

Заключение

- Предварительные (априорные) знания играют важную роль в байесовской модели и могут сильно влиять на апостериорные предсказания.
- В ГП априорные знания можно указать, используя функции среднего значения и ковариации.
- Функция среднего описывает ожидаемое поведение модели ГП. При отсутствии данных апостериорное среднее предсказание возвращается к априорному среднему значению.
- Функция среднего может принимать любую функциональную форму, включая постоянную (константу), а также линейную и квадратичную функции, которые могут быть реализованы с помощью GPyTorch.
- Ковариационная функция контролирует гладкость модели ГП.
- Масштаб длины определяет уровень изменчивости вывода относительно входных данных функции. Больший масштаб приводит к большей плавности и, следовательно, к меньшей неопределенности прогнозов.
- Каждое измерение в ГП может иметь свой собственный масштаб длины. Это называется *автоматическим определением релевантности* (ARD) и используется для моделирования целевых функций, которые имеют разные уровни изменчивости в разных измерениях.
- Масштаб вывода определяет диапазон выходных данных функции. Большой масштаб приводит к большему диапазону выходных данных и, следовательно, к большей неопределенности прогнозов.
- Класс ядра Матерна является обобщением класса ядра RBF. Указав в нем параметр `nu`, можно моделировать различные уровни гладкости прогнозов ГП.
- Гиперпараметры можно оптимизировать, максимизируя предельную вероятность данных с помощью градиентного спуска.

Часть II

Принятие решений с помощью байесовской оптимизации

Гауссовский процесс — это лишь часть уравнения. Чтобы полностью реализовать метод БО, нужна *политика принятия решений*. Она диктует, как следует выполнять оценку для максимально быстрой оптимизации целевого объекта. В этой части книги перечислены наиболее популярные политики БО, включая их принципы, математическую интуицию и реализацию. Несмотря на то что разные политики мотивированы разными целями, все они направлены на баланс между *исследованием* (**Exploration**) и *эксплуатацией* (**Exploitation**). Это является основной задачей БО в частности и принятия решений в условиях неопределенности в целом.

В главе 4 мы рассмотрим *оценку приобретения* (**Acquisition score**), как способ измерить значимость (важность) оценки целевой функции. Мы также познакомимся с эвристикой стремления к улучшению, что приведет нас к двум популярным политикам БО: *вероятность улучшения* (**Probability of Improvement**, **PoI**) и *ожидаемое улучшение* (**Expected Improvement**, **EI**).

В главе 5 мы объединим БО и тесно связанную с ней проблему *многоруких бандитов* (**Multi-armed bandit**). Здесь также будут исследованы две популярные политики: *верхняя доверительная граница* (**Upper Confidence Bound**, **UCB**), которая использует эвристику *оптимизма* в условиях неопределенности, и *выборка Томпсона* (**Thompson Sampling**, **TS**), которая использует вероятностную природу ГП для облегчения принятия решений.

Глава 6 познакомит нас с *теорией информации* — разделом математики, который имеет множество приложений в задачах принятия решений. Используя централь-

ную концепцию данной теории, называемую *энтропией*, мы разработаем политику, которая направлена на получение как можно большего количества информации о цели поиска.

В этой части мы узнаем о различных способах, которые помогут решить проблему *компромисса между исследованием и эксплуатацией* целевых объектов, а также создадим разнообразный набор инструментов для методов оптимизации. Практика, содержащаяся здесь, будет сосредоточена на другой библиотеке Python — BoTorch. Мы научимся объявлять политики, использовать их для упрощения цикла оптимизации и сравнивать их производительность в широком диапазоне задач. К концу этой части вы получите практические знания о внедрении и использовании политик БО.

4 Достижение наилучшего результата с помощью политик, основанных на улучшениях

Темы главы:

- Цикл БО.
- Компромисс между исследованием и эксплуатацией в политике БО.
- *Улучшение* как критерий для поиска новых точек данных.
- Политики БО, использующие улучшения.

Для начала стоит вспомнить итеративную природу байесовской оптимизации: обучение гауссовского процесса на собранных данных чередуется с поиском следующей точки с использованием политики БО. Это формирует цикл, в котором прошлые данные определяют будущие решения. Затем в этой главе мы поговорим о том, что нам нужно от политики БО, а именно об алгоритме принятия решений, который определяет, какую точку данных маркировать. «Хорошая» политика должна обеспечивать баланс между достаточным исследованием пространства поиска и сосредоточением внимания на высокоэффективных регионах.

Мы также познакомимся с двумя политиками, которые направлены на то, чтобы улучшить наиболее заметную точку данных в цикле БО: *вероятность улучшения* и *ожидаемое улучшение*. Предположим, есть приложение настройки гиперпараметров, в котором нужно определить нейросеть, обеспечивающую наивысшую точность при проверке набора данных. При этом самая высокая точность, наблюдаемая к текущему моменту, составляет 90%. Нам, определенно, необходимо увеличить этот порог. Политики, изучаемые в этой главе, способны помочь добиться такого улучшения. В задаче по обнаружению материалов (табл. 1.2) с самой низкой температурой смешивания было найдено значение 187,24. Две вышеупомянутые политики будут стремиться найти значения ниже этой отметки.

Удивительно, но благодаря «гауссовости» представлений о целевой функции можно в закрытой форме вычислить, насколько более сильных улучшений стоит ожидать по сравнению с самой заметной точкой данных. Это означает, несмотря на то,

что нам не известно, как выглядит целевая функция в невидимых местах, вычисление величин, основанных на улучшении, все же может быть легко выполнено под руководством гауссовского процесса. К концу главы вы получите полное понимание, что должна делать политика, а также как этого достичь, используя две политики, основанные на улучшениях, которые описаны выше. Вы также узнаете, как в Python интегрировать библиотеку BoTorch, которая будет использоваться для реализации политик не только в этой главе, но и до конца книги.

4.1. Навигация по пространству поиска в БО

Как можно убедиться, что данные для информирования будущих решений используются правильно? Что нам нужно от политики как от автоматизированной процедуры принятия решений? Данный раздел отвечает на эти вопросы и дает четкое представление, как работает байесовская оптимизация при использовании гауссовских процессов.

В следующем разделе мы еще раз рассмотрим цикл БО, который был кратко представлен в разделе 1.2.2. Это нужно, чтобы увидеть, как принятие решений через политики реализовано в тандеме с обучением ГП. Затем мы обсудим основную задачу, которую политика призвана решить, — баланс между исследованием регионов с высокой неопределенностью и эксплуатацией подходящих регионов в пространстве поиска.

Для примера рассмотрим рис. 4.1, на котором показан ГП, обученный на двух точках данных: 1 и 2. Здесь политика призвана помочь решить, в какой точке между –5 и 5 стоит оценивать целевую функцию в следующий раз. Компромисс между исследованием и эксплуатацией очевиден. Нужно решить, где дальше проверять целевую функцию: на левом и правом краях пространства (около –5 и 5), где существует значительная неопределенность в прогнозах, или остаться в области вокруг 0, где на данный момент имеется самый высокий средний прогноз. Баланс между исследованием и эксплуатацией подготовит почву для различных политик, которые будут обсуждаться в этой и последующих главах.

4.1.1. Цикл БО и политики

Для начала стоит вспомнить, как выглядит цикл БО и какую роль политики играют в этом процессе. В данной главе мы также реализуем «каркас» данного цикла, который будет использоваться для проверки политик в будущих главах. Рис. 4.2 повторяет рис. 1.6 и показывает, как БО работает на высоком уровне.

БО выполняется циклом, который включает следующие составляющие:

- Обучение ГП на текущем обучающем наборе (как это сделать, мы подробно рассмотрели в предыдущих главах).
- Использование обученного ГП для оценки точек данных на пространстве поиска в зависимости от их полезности, которая помогает определить оптимум целевой функции (шаг 2 на рис. 4.2). Точка, которая максимизирует эту оценку, марки-

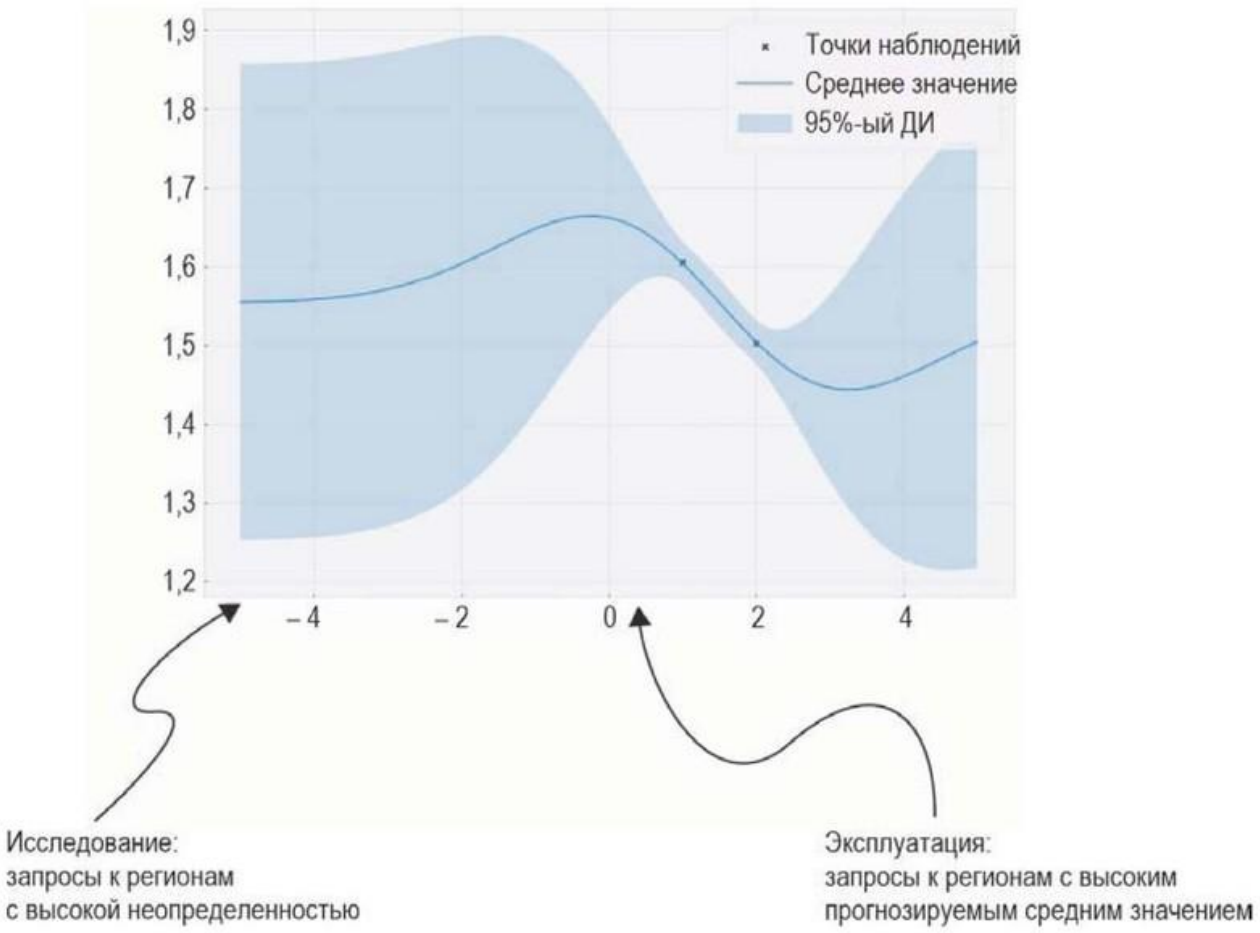


Рис. 4.1. Компромисс между исследованием и эксплуатацией в БО: каждая из политик должна решить, что следует делать, запрашивать регионы с высокой неопределенностью (исследование) или запрашивать регионы с высоким прогнозируемым средним значением (эксплуатация)

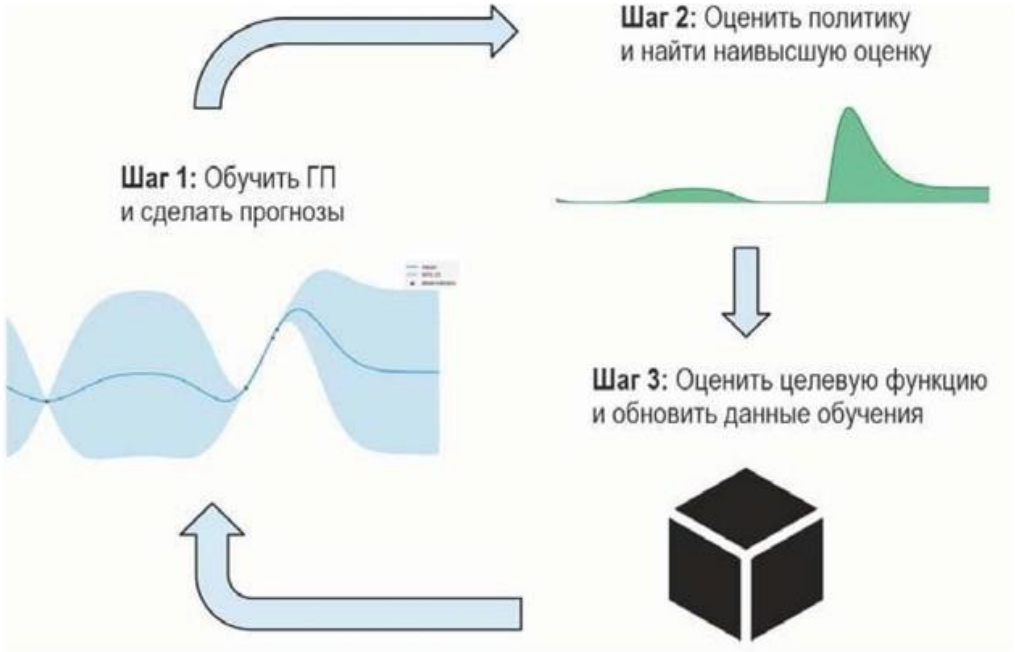


Рис. 4.2. Цикл БО, сочетающий в себе ГП для моделирования и политику для принятия решений: этот полный рабочий процесс теперь можно использовать для оптимизации функций «черного ящика»

руется и добавляется к обучающим данным (шаг 3 на рис. 4.2). Способ оценки определяется используемой политикой.

Цикл повторяется, пока не будет достигнуто завершающее условие (чаще всего, когда будет вычислена целевая функция для определенного числа итераций цикла). Эта процедура является сквозной, поскольку мы используем ГП не просто для предсказания ради самих прогнозов. Они влияют на решения, какие точки данных следует собирать дальше. Это, в свою очередь, определяет, как генерируются будущие прогнозы.

Определение

Цикл БО — это эффективный цикл как обучения модели (ГП), так и сбора данных (политика). Каждый из компонентов не только помогает другим, но и извлекает выгоду из них, чтобы найти оптимум целевой функции. Полезный цикл БО — это петля обратной связи, которая движется к равновесию с желаемыми свойствами. Его компоненты действуют согласованно для достижения нужного результата, а не усугубляют друг друга, образуя порочный круг, приводящий к нежелательному итогу.

Правило, которое определяет, как точки данных оцениваются в соответствии с их ценностью (значимостью, важностью), определяется политикой и, следовательно, имеет большое значение для оптимизации производительности. Хорошая политика, присваивающая высокие оценки точкам данных, которые действительно важны для оптимизации, быстрее и эффективнее укажет на оптимум целевой функции, в то время как плохо разработанная политика может ввести в заблуждение исследования и потратить важные ресурсы.

Определение

Политика БО оценивает каждый потенциальный запрос в соответствии с его значением и, таким образом, решает, где следует запрашивать целевую функцию в следующий раз (максимальная оценка). Это называется *оценкой приобретения* (**Acquisition score**), поскольку она используется в качестве метода оценки непосредственно для сбора данных (приобретения данных).

Политики обучения с подкреплением

Если у вас есть опыт *обучения с подкреплением* (**Reinforcement Learning**, **RL**), то вы можете заметить связь между политикой БО и политикой RL. В обоих методах они подсказывают, какое действие следует предпринять при возникновении проблемы принятия решений. В RL политика может присваивать оценку каждому действию. Поэтому мы, в свою очередь, выбираем действие с наивысшей оценкой. Или политика может просто выводить действие, которое следует предпринять. В БО всегда выполняется первый вариант: политика выводит оценку, определяющую количественную значимость каждого возможного запроса. Поэтому наша задача — определить запрос, максимизирующий такую оценку.

К сожалению, не существует идеального решения при разработке хорошей политики. Точнее сказать, нет единого решения, которое последовательно превосходило бы все целевые функции. Различные политики, как видно в этой и последующих главах, используют разные фокусы внимания и эвристики. Несмотря на то что некоторые из них хорошо работают с одним типом целевой функции, другие могут

оказаться эффективными для других типов. Это означает, что нам необходимо ознакомиться с широким спектром политик и понять, на что они направлены, чтобы иметь возможность применять их в соответствующих ситуациях. Именно об этом будет идти речь в главах 4–6.

Что такое политика БО?

Каждая политика представляет собой правило принятия решений, которое оценивает точки данных, насколько они полезны при оптимизации в соответствии с заданным критерием или эвристикой. Разные критерии и эвристики порождают разные политики, и не существует заранее определенного их набора. Фактически исследователи БО публикуют статьи, просто предлагая новые политики. В этой книге мы будем обсуждать только наиболее популярные и часто используемые на практике стратегии.

Далее реализуем некую «заглушку» цикла БО, который с данного момента будет использоваться для проверки различных политик. Данный код реализован в файле `CH04/01 – BayesOptloop.ipynb`. Первый необходимый компонент — это целевая функция, которую нужно оптимизировать с помощью БО. В качестве нее мы будем использовать уже знакомую одномерную функцию Форрестера, определенную в диапазоне от –5 до 5. Также вычислим значение внутри ее области определения [–5, 5], используя `xs` и `ys` в качестве основной истины (целевая функция):

```
#Форма целевой функции, которая считается неизвестной
def forrester_1d(x):
    y = -((x + 1) ** 2) * torch.sin(2 * x + 2) / 5 + 1
    return y.squeeze(-1)


#Тестовые данные, рассчитанные по сетке от -5 до 5
bound = 5
xs = torch.linspace(-bound, bound, bound * 100 + 1).unsqueeze(1)
ys = forrester_1d(xs)
```

Еще одна вещь, которую нужно сделать, — это изменить способ реализации моделей, чтобы их можно было использовать с политиками в BoTorch. Реализация ГП составляет первый шаг цикла БО.

Поскольку BoTorch создан на основе GPyTorch, требуются лишь минимальные изменения в коде. В частности, мы будем использовать реализацию ГП, в которой наследуется класс `botorch.models.gpytorch.GPyTorchModel` в дополнение к обычному классу `gpytorch.models.ExactGP`. Далее объявим атрибут `num_outputs` (специфичный для класса) и установим для него значение 1. Это минимальные изменения, которые необходимо внести, чтобы использовать модель с BoTorch. Политики, которые он реализует, мы будем позже использовать в этой главе:

```
class GPModel(gpytorch.models.ExactGP,
  botorch.models.gpytorch.GPyTorchModel): #Модификации для интеграции BoTorch
    num_outputs = 1

    def __init__(self, train_x, train_y, likelihood):
        super().__init__(train_x, train_y, likelihood)
```

```
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(
          gpytorch.kernels.RBFKernel()
        )

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

Помимо этого, все остальное в реализации ГП останется прежним. Теперь напишем вспомогательную функцию, которая обучает ГП на наборе данных:

```
#Обучение ГП с помощью градиентного спуска
def fit_gp_model(train_x, train_y, num_train_iters=500):
    noise = 1e-4

    #Объявляем ГП
    likelihood = gpytorch.likelihoods.GaussianLikelihood()
    model = GPModel(train_x, train_y, likelihood)
    model.likelihood.noise = noise

    optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
    mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

    #Обучаем ГП, используя градиентный спуск
    model.train()
    likelihood.train()

    for i in tqdm(range(num_train_iters)):
        optimizer.zero_grad()

        output = model(train_x)
        loss = -mll(output, train_y)

        loss.backward()
        optimizer.step()

    model.eval()
    likelihood.eval()

    return model, likelihood
```

Примечание

Весь данный код был использован в предыдущих главах. Если у вас возникли проблемы с его пониманием, вы можете обратиться к разделу 3.3.2 за более подробной информацией.

Этот фрагмент кода охватывает шаг 1 рис. 4.2 (на данный момент шаг 2, на котором реализуются политики, мы пропускаем). Он будет сохранен для следующего

раздела и будущих глав. Другой компонент, который необходимо реализовать, — это визуализация данных для текущего ГП, которые были собраны к настоящему моменту времени, а также отображение того, как политика оценивает остальные точки. Результат такой визуализации показан на рис. 4.3. Аналогичный рисунок мы видели в главе 1. Верхняя часть графика показывает прогнозы, сделанные моделью ГП, в сравнении с истинной целевой функцией (основной истиной). Нижняя часть демонстрирует вычисленные оценки приобретения, рассчитанные с помощью политики.

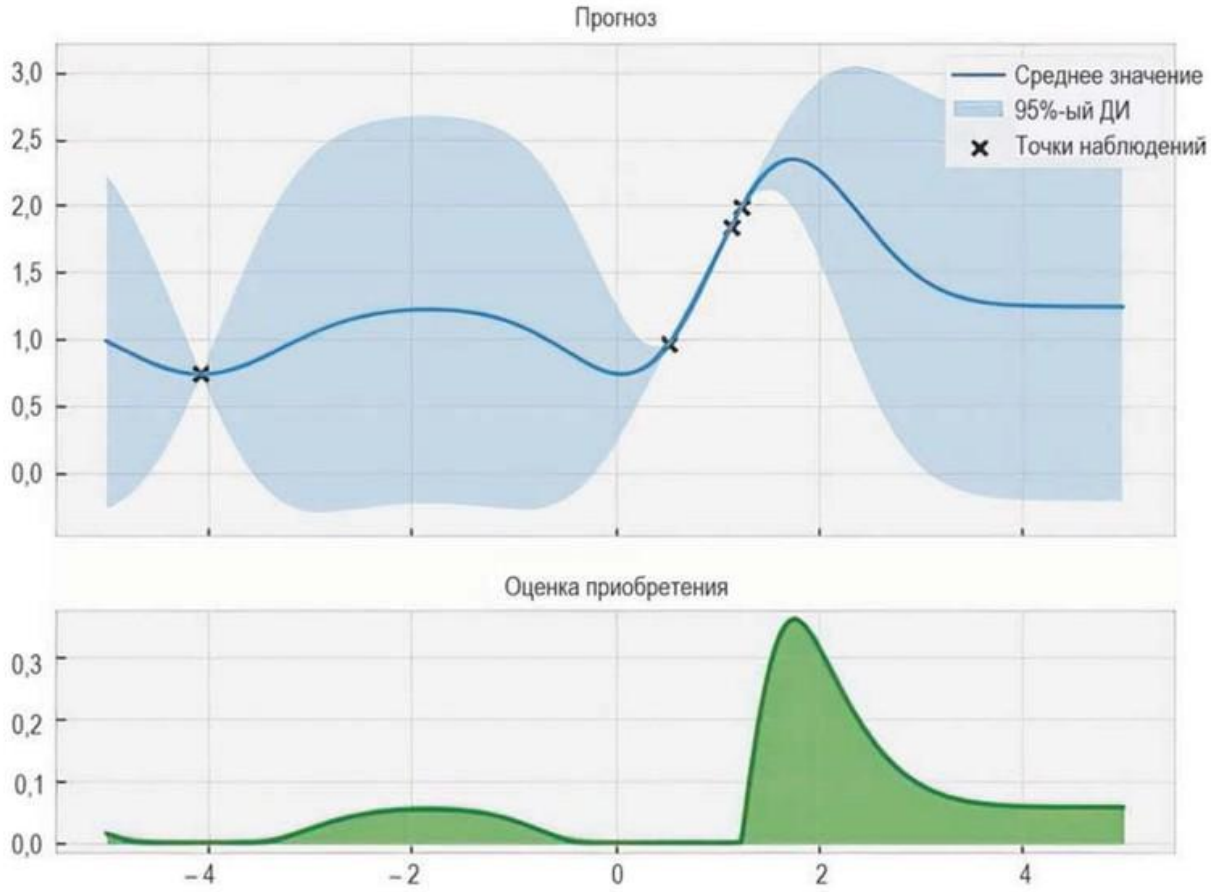


Рис. 4.3. Типичная визуализация прогресса в БО: сверху показаны прогнозы ГП и основная истинная целевая функция; снизу показаны оценки приобретения, полученные с помощью политики под названием «Ожидаемое улучшение», о которой более подробно будет рассказано в разделе 4.3

Нам уже известно, как создать верхнюю часть графика. Нижняя часть реализуется так же просто. Это можно сделать, используя вспомогательную функцию, аналогичную той, что была использована в разделе 3.3. Она принимает модель ГП и ее функцию вероятности, а также два дополнительных входных параметра:

- Атрибут `policy` относится к объекту политики, который можно вызывать аналогично любому модулю PyTorch. Здесь он вызывается в сетке `xs` (пространство поиска), чтобы получить оценки приобретения по всему пространству. Как реализовать эти объекты с помощью BoTorch, будет обсуждаться в следующем разделе. Сейчас нам достаточно знаний об этих объектах.
- Атрибут `next_x` — это местоположение точки, максимизирующей оценку приобретения, которая должна быть добавлена к текущим обучающим данным:

```
    def visualize_gp_belief_and_policy(model, likelihood, policy=None, next_x=None):
        with torch.no_grad():
            #Прогнозы ГП
            predictive_distribution = likelihood(model(xs))
            predictive_mean = predictive_distribution.mean
            predictive_upper, predictive_lower =
              predictive_distribution.confidence_region()

            Оценка приобретения
            if policy is not None:
                acquisition_score = policy(xs.unsqueeze(1))

        ... #Опустим некоторые детали
```

Этот фрагмент генерирует прогнозы ГП и оценки приобретения на основе тестовых данных `xs`. Обратите внимание, что оценки приобретения не вычисляются, если политика не передана. В таком случае прогнозы также визуализируются уже известным способом: разбросанные точки обозначают данные обучения, сплошная линия — средние прогнозы, а заштрихованные области — 95%-ный ДИ:

```
if policy is None:
    plt.figure(figsize=(8, 3))

    plt.plot(xs, ys, label="objective", c="r") # Основная истина
    plt.scatter(train_x, train_y, marker="x", c="k",
      label="observations") # Обучающие данные

    #Средние прогнозы и 95%-ный ДИ
    plt.plot(xs, predictive_mean, label="mean")
    plt.fill_between(
        xs.flatten(),
        predictive_upper,
        predictive_lower,
        alpha=0.3,
        label="95% CI",
    )

    plt.legend()
    plt.show()
```

> **Примечание**
>
> Для получения дополнительной информации об этой визуализации вы можете обратиться к разделу 2.4.4.

С другой стороны, если объект политики передан, то создается еще один подграфик, демонстрирующий оценки приобретения в пространстве поиска:

```
else:
    fig, ax = plt.subplots(2,1,figsize = (8, 6),sharex=True,
      gridspec_kw = {"height_ratios": [2, 1]}
    )
```

```
... #Прогнозы ГП (такие же, как и раньше)

#Точка, максимизирующая оценку приобретения, визуализирована
#с помощью пунктирной вертикальной линии

if next_x is not None:
    ax[0].axvline(next_x, linestyle="dotted", c="k")

#Оценки приобретения
ax[1].plot(xs, acquisition_score, c="g")
ax[1].fill_between(xs.flatten(),acquisition_score,0,color="g",alpha=0.5)

#Точка, максимизирующая оценку приобретения, визуализирована
#с помощью пунктирной вертикальной линии
if next_x is not None:
    ax[1].axvline(next_x, linestyle="dotted", c="k")

ax[1].set_ylabel("acquisition score")

plt.show()
```

Когда `policy` и `next_x` переданы, эта функция создаст нижнюю часть графика, показывающую оценку приобретения в соответствии с политикой. Теперь можно реализовать шаг 3 цикла БО (рис. 4.2), на котором выполняется поиск точки с наивысшей оценкой приобретения (шаг 1). Затем она добавляется к обучающим данным, и происходит обновление ГП (шаг 2). Для задачи по определению точки (шаг 1), дающей наивысшую оценку приобретения (несмотря на то что в примере с функцией Форрестера можно сканировать одномерное пространство поиска), исчерпывающий поиск становится все более дорогостоящим по мере увеличения числа измерений целевой функции.

> **Примечание**
>
> Мы можем использовать вспомогательную функцию BoTorch — `botorch.optim.optimize.optimize_acqf()`. Она находит точку, которая максимизирует оценку любой политики. Вспомогательная функция использует L-BFGS — квазиньютоновский метод оптимизации, который обычно работает лучше, чем градиентный спуск.

Это можно сделать следующим образом:

- `policy` — объект политики, который подробно будет рассмотрен позже.
- `bounds` хранит границы пространства поиска, которые в данном случае равны –5 и 5.
- `q = 1` хранит количество точек, которые нужно вернуть вспомогательной функции (в данном случае одну). В главе 7 будет подробно рассмотрена настройка, которая позволяет выполнять несколько запросов к целевым функциям одновременно.

- `num_restarts` и `raw_samples` — количество повторов и начальных точек данных (соответственно), которые L-BFGS использует при поиске оптимального кандидата, дающего наивысшую оценку приобретения. Рекомендуется использовать для этих параметров в 20 и 50 раз больше измерений соответственно.
- Возвращаемые значения `next_x` и `acq_val` — это местоположение точки, дающей наивысшую оценку приобретения, и максимальная оценка приобретения соответственно:

```
next_x, acq_val = botorch.optim.optimize_acqf(
    policy,
    bounds=torch.tensor([[-bound * 1.0], [bound * 1.0]]),
    q=1,
    num_restarts=20,
    raw_samples=50,
)
```

Настройка количества перезапусков и необработанных выборок

Чем выше значения `num_restarts` и `raw_samples`, тем более исчерпывающим будет L-BFGS при поиске оптимального кандидата, максимизирующего оценку приобретения. Это также означает, что выполнение алгоритма займет больше времени. Можно увеличить эти два значения, если L-BFGS не удается максимизировать оценку приобретения, или уменьшить их, если алгоритм работает слишком долго.

На данном этапе в цикле БО можно объединить все, что уже известно на данный момент. Цикл БО имеет следующие итерации:

1. Выводится лучшее значение, найденное к этому моменту (`train_y.max()`). Оно показывает, как продвигается оптимизация.
2. ГП переобучается на текущих данных. Повторно объявляется политика БО.
3. С помощью вспомогательной функции `botorch.optim.optimize_acqf()` из BoTorch определяется точка в пространстве поиска, которая максимизирует оценку приобретения.
4. Вызывается вспомогательная функция `visualize_gp_belief_and_policy()`, которая визуализирует текущее убеждение ГП и прогресс оптимизации.
5. Наконец, запрашивается значение функции в определенной точке (`next_x`), а также обновляются наблюдаемые данные.

Вся процедура обобщена на рис. 4.4, где показаны шаги цикла БО и соответствующий код. Каждый шаг реализуется с помощью модулей, состоящих из пользовательских вспомогательных функций или функций BoTorch, что упрощает выполнение всей процедуры в целом.

Фактическая реализация кода выглядит так:

```
#Количество оценок целевой функции, которое можно сделать
num_queries = 10

for i in range(num_queries):
    print("iteration", i)
    print("incumbent", train_x[train_y.argmax()], train_y.max())
```



Рис. 4.4. Шаги цикла БО и соответствующий им код, который для каждого шага является модульным, что позволяет легко отслеживать весь цикл

```
#Обновляем модель на текущих данных
model, likelihood = fit_gp_model(train_x, train_y)

#Инициализируем политику БО, которая будет рассмотрена позже
policy = ...

#Находим точку, которая дает наивысшую оценку приобретения
next_x, acq_val = botorch.optim.optimize_acqf(
  policy,
  bounds=torch.tensor([[-bound * 1.0],[bound * 1.0]]),
  q=1,
  num_restarts=20,
  raw_samples=50,
)
```

```
#Визуализируем текущую модель ГП и оценки приобретения
visualize_gp_belief_and_policy(model, likelihood, policy, next_x=next_x)

#Проводим наблюдение в указанной точке и обновляем данные обучения
next_y = forrester_1d(next_x)
train_x = torch.cat([train_x, next_x])
train_y = torch.cat([train_y, next_y])
```

Таким образом, мы реализовали «каркас» цикла БО. Все, что осталось сделать, — это заполнить инициализацию `policy = ...` фактической политикой, которую нужно использовать. Обратите внимание, несмотря на то, что вызов `visualize_gp_belief_and_policy()` не является обязательным (цикл БО может работать без этой строки кода), эта функция очень полезна. Она помогает наблюдать за поведением и характеристиками политик, а также диагностировать любые потенциальные проблемы, о чем мы поговорим далее в этой главе.

Одной из наиболее важных характеристик политики является баланс между исследованием и эксплуатацией (классический компромисс во многих проблемах искусственного интеллекта и машинного обучения). Возможность обнаружить высокопроизводительный регион, о котором в настоящее время неизвестно (исследование), противопоставляется шансу сосредоточиться на заведомо хорошем регионе (эксплуатация).

4.1.2. Баланс между исследованием и эксплуатацией

В этом разделе мы обсудим одну из проблем, присущих любой процедуре принятия решений, включая БО, — баланс между исследованием всего пространства поиска и своевременным использованием (эксплуатацией) регионов, которые дают хорошие результаты. Это обсуждение поможет сформировать представление о том, что должны делать хорошие политики, и позволит понять, как каждая из них будет решать этот компромисс.

Чтобы проиллюстрировать это, представьте, что вы обедаете в ресторане, в котором до этого были всего несколько раз (рис. 4.5). Вы знаете, что здесь отличные гамбургеры, но не уверены, хороши ли там рыба и стейки. Здесь вы сталкиваетесь с проблемой исследования-эксплуатации, где нужно выбирать между заказом чего-то нового, что могло бы стать отличным выбором (исследование), и заказом привычной, но надежной еды (эксплуатация).

Чрезмерное исследование может привести к тому, что вы закажете то, что вам не понравится. Тогда как, постоянно заказывая привычное, вы рискуете упустить блюдо, которое действительно будет превосходным. Поэтому разумный баланс между ними имеет решающее значение.

Подобная проблема касается не только заказа еды, а даже искусственного интеллекта (ИИ): обучение с подкреплением, рекомендации продуктов и научные открытия. В БО мы сталкиваемся с тем же компромиссом: нужно достаточно изучить пространство поиска, чтобы не упустить хороший регион, но также необходимо сосредоточиться на областях с высокими целевыми значениями, где точно будет оптимизация.

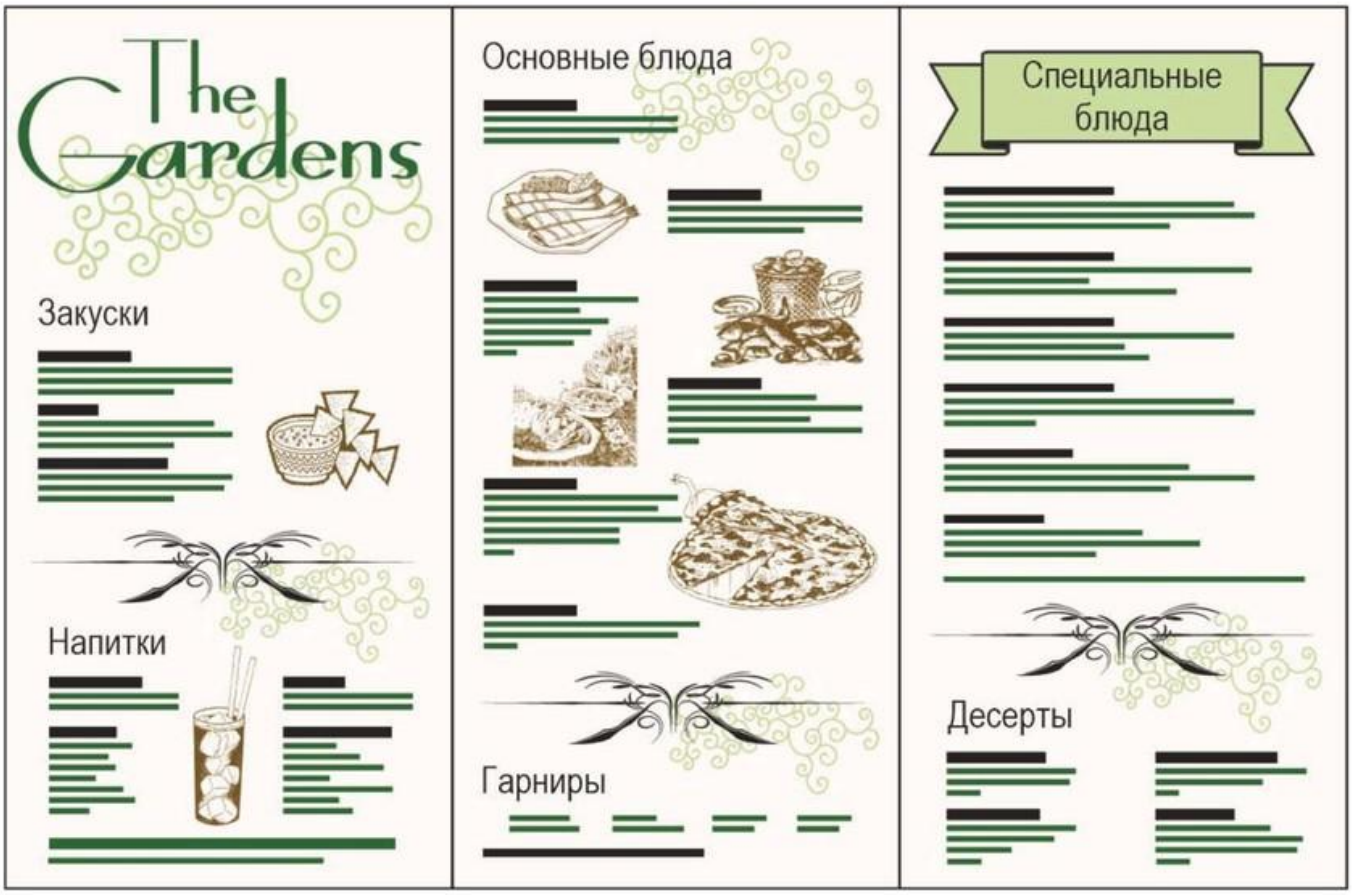


Рис. 4.5. Заказ из меню в ресторане предполагает компромисс между исследованием (попробовать что-то новое) и эксплуатацией (заказать привычное)

ПРИМЕЧАНИЕ

Под «областями с высокими целевыми значениями» подразумеваются регионы, которые содержат входные данные x, дающие высокие значения выходных данных $f(x)$. Они являются целью оптимизации (в частности, максимизации).

Вернемся к коду и предположим, что вначале набор обучающих данных содержит две точки наблюдения целевой функции Форрестера ($x = 1$ и $x = 2$):

```
train_x = torch.tensor([
  [1.],
  [2.]
])
train_y = forrester_1d(train_x)

model, likelihood = fit_gp_model(train_x, train_y)

print(torch.hstack([train_x, train_y.unsqueeze(1)]))
```

Результат будет следующий:

```
tensor([[1.0000, 1.6054],
        [2.0000, 1.5029]])
```

Это означает, что точка 1 имеет оценку примерно 1,6, а точка 2 — 1,5. Визуализируя прогнозы, сделанные обученным ГП, получаем знакомый график, показанный

на рис. 4.6. Он отображает компромисс между исследованием и эксплуатацией, с которым мы сталкиваемся: следует ли оценивать целевую функцию там, где есть большая неопределенность, или стоит оставаться в области, где средний прогноз высок?

Каждая политика по-своему подходит к решению и, следовательно, предлагает разные советы, как лучше всего исследовать пространство поиска. На рис. 4.6 некоторые политики могут побудить к дальнейшему изучению неизвестных регионов, в то время как другие будут предлагать сосредоточить внимание на высоких значениях. Не существует универсального подхода (политики, которая всегда работала бы хорошо).

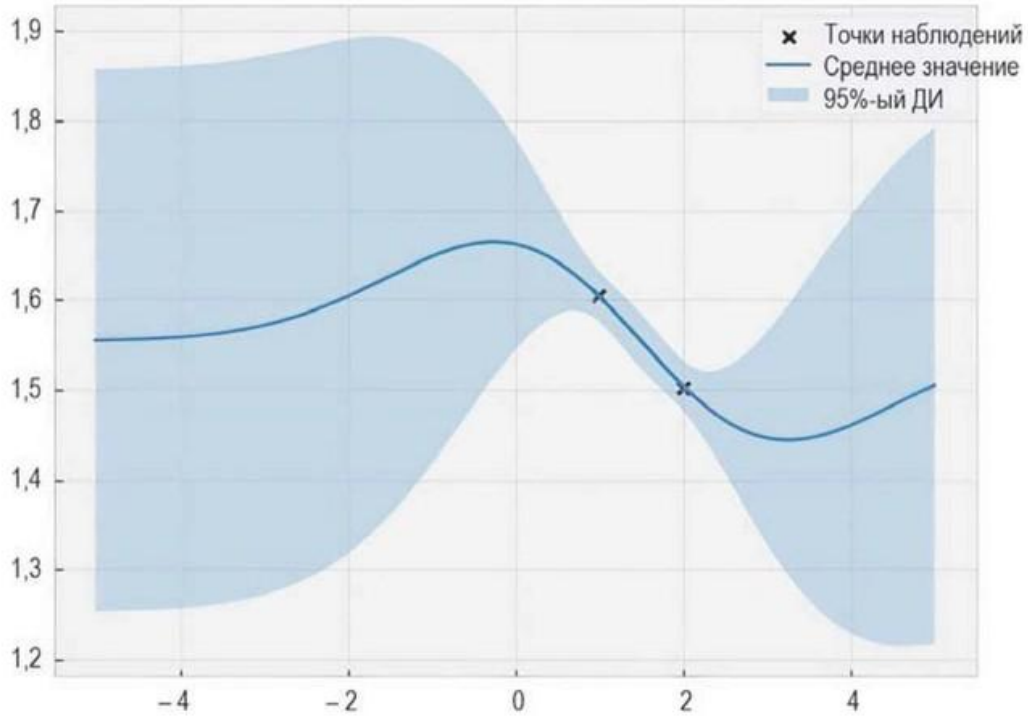


Рис. 4.6. Прогнозы ГП, обученного на двух точках данных функции Форрестера

4.2. Поиск улучшений в БО

У нас есть почти все необходимое, чтобы запустить БО для заданной целевой функции. Теперь нужна политика с правилом оценки, насколько в поиске оптимума значима каждая точка данных, которую потенциально можно промаркировать. Стоит напомнить, что каждая политика дает разные правила оценки, основанные на разных эвристиках оптимизации.

В этом разделе мы подробно рассмотрим одну из них. Она интуитивно понятна, когда целью является оптимизация. Это проявляется в стремлении к улучшению с наиболее перспективной точки, которую можно найти в ходе оптимизации. Далее мы узнаем, как ГП может облегчить расчет улучшения. А затем рассмотрим, как различные способы определения улучшения приводят к появлению двух наиболее распространенных политик: *вероятность улучшения* и *ожидаемое улучшение*.

4.2.1. Измерение улучшения с помощью ГП

Рассмотрим, как определяется улучшение в БО, каким образом оно является хорошей мерой полезности, а также то, что работа с величинами, связанными с улучшением, проста при нормальном распределении. Поскольку наша конечная цель — определить глобальный оптимум целевой функции (точка, которая дает наивысшее целевое значение), чем выше значения, которые наблюдаются при оценке целевой функции, тем выше должна быть полезность. Предположим, оценивая целевую функцию в некоторой точке x_1, мы наблюдаем значение 2, а в точке x_2 — значение 10. Интуитивно вторая точка имеет ценность больше, поскольку это дает более высокое значение функции.

Но что если до наблюдения x_1 и x_2 уже было наблюдение точки x_0, которая дает значение 20? В этом случае нормально предполагать, что хотя x_2 и лучше, чем x_1, ни одна из них не приводит к какой-либо дополнительной полезности, поскольку уже есть лучшее наблюдение в x_0. С другой стороны, если бы была точка x_3 со значением 21, это можно было бы считать улучшением, поскольку она имеет большее значение, чем x_0. Все это показано на рис. 4.7.

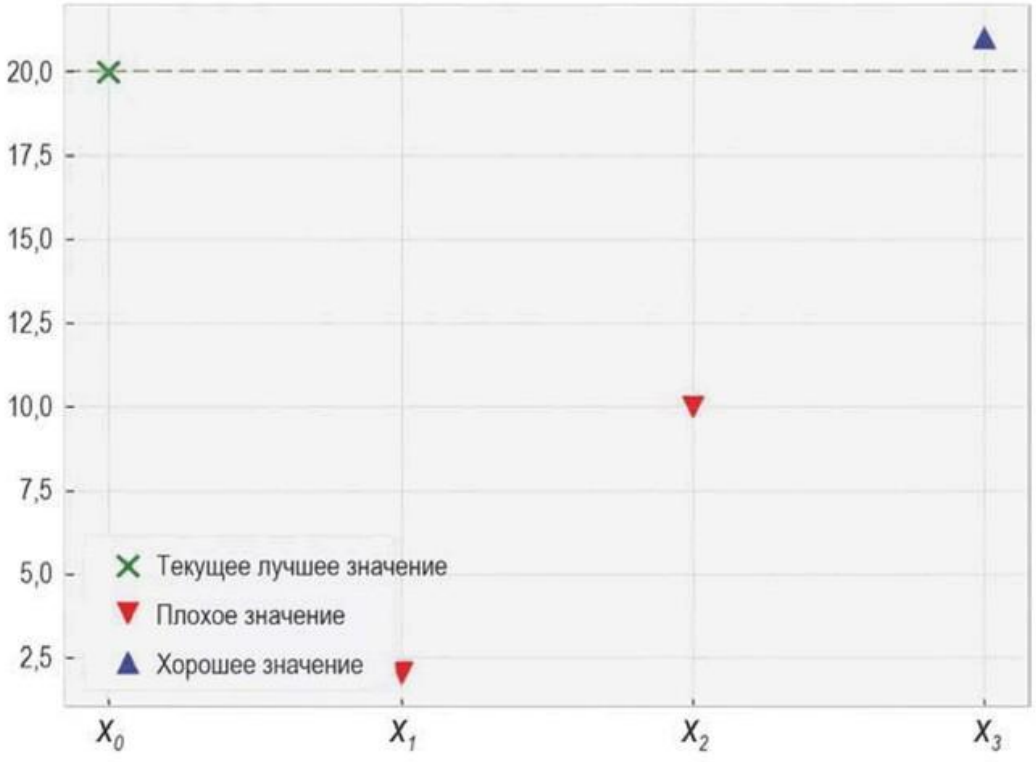


Рис. 4.7. Поиск улучшений относительно наилучшей точки обзора: несмотря на то, что точка x_2 лучше, чем x_1, обе они являются «плохими» относительно x_0

Такое сравнение показывает, что в БО важны не только необработанные значения наблюдений, но и то, насколько лучше вновь найденные наблюдения по сравнению с имеющимися. В этом случае x_0 устанавливает очень высокую планку, поэтому ни x_1, ни x_2 не являются улучшением (по крайней мере, тем улучшением, которое нас интересует при оптимизации). Другими словами, разумная цель оптимизации — стремиться к улучшению самой полезной точки, наблюдаемой к настоящему моменту, поскольку это и означает прогресс.

Определение

Точка, которая видна лучше всего или дает наибольшее значение функции на данный момент, часто называется *действующей* (**Incumbent point**). Этот термин обозначает, что она в настоящее время имеет наибольшее *действующее значение* (**Incumbent value**) среди всех других точек, запрошенных во время поиска.

Учитывая, что у нас есть общее представление о целевой функции, можно легко проверить, насколько сильные улучшения можно ожидать относительно точки, которую видно лучше всего. Рассмотрим уже известный пример функции Форрестера, текущий ГП которого показан на рис. 4.6.

В обучающем наборе данных точка ($x = 1$, $y = 1{,}6$) является лучшей, поскольку она имеет более высокое значение функции. Задача — улучшить порог 1,6. А именно нужно найти точки данных, которые будут давать значения функции выше этого порога.

Визуально можно представить эту идею, как горизонтальный срез ГП по действующей точке, как показано на рис. 4.8. Часть, выделенная более темным цветом, соответствует «улучшению позиции действующей точки» (1,6). Все, что ниже этой линии, не является улучшением и, следовательно, не дает никакой дополнительной пользы при оптимизации. Неизвестно, даст ли точка (например, $x = 0$) более высокое значение функции, но все равно, используя информацию из модели ГП, можно попытаться рассуждать о вероятности того, что это утверждение истинно.

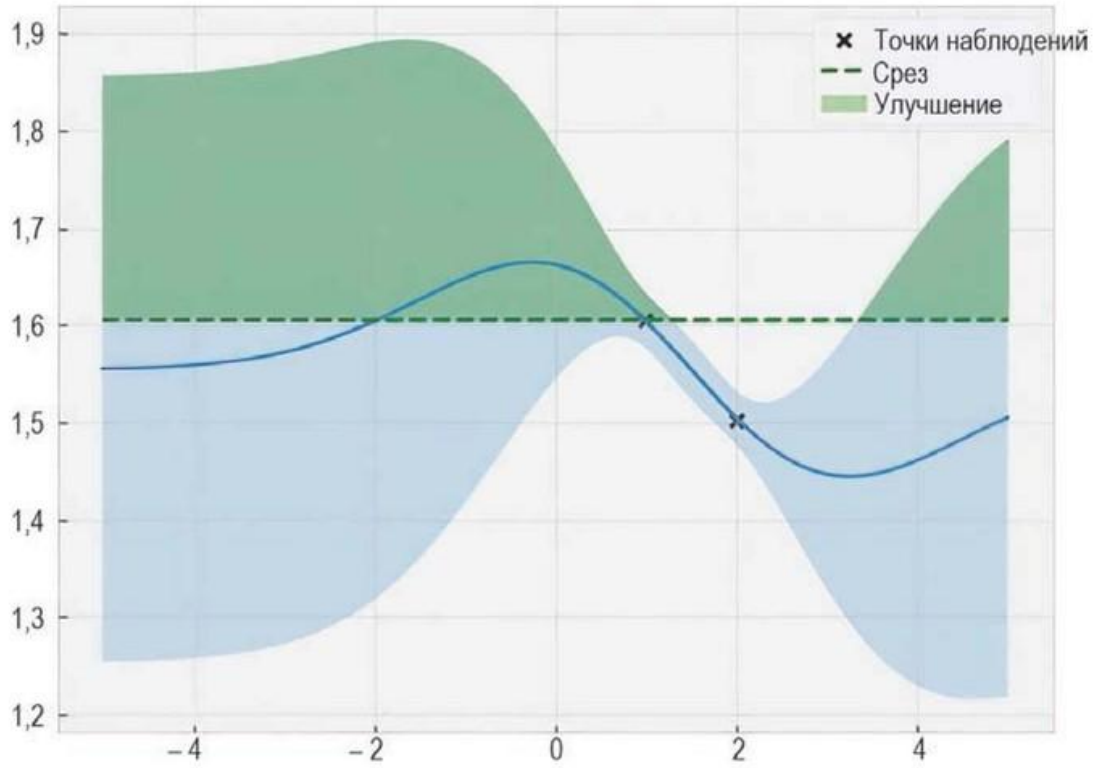


Рис. 4.8. Улучшение действующего значения с точки зрения ГП: часть прогнозов, которая соответствует улучшению относительно действующей точки, выделена более темным оттенком

Рассуждать о вероятности, что точка $x = 0$ приведет к более высокому значению функции, легко, поскольку, запрашивая ее, улучшение действующей точки точно соответствует частично усеченному нормальному распределению (рис. 4.9).

Левая часть содержит тот же ГП, что и на рис. 4.8, который усечен по действующей точке, с дополнением, демонстрирующим ДИ для прогноза нормального распределения в точке $x = 0$. Разрезая ГП по вертикали в точке 0, мы получаем правую часть рисунка, где ДИ такой же, как на левой части. Можно убедиться, что только выделенная часть нормального распределения на правой части представляет улучшение, которое можно наблюдать у действующей точки. Это именно то, что нас интересует. Выделенная область является частью нормального распределения, которое дает множество математических удобств.

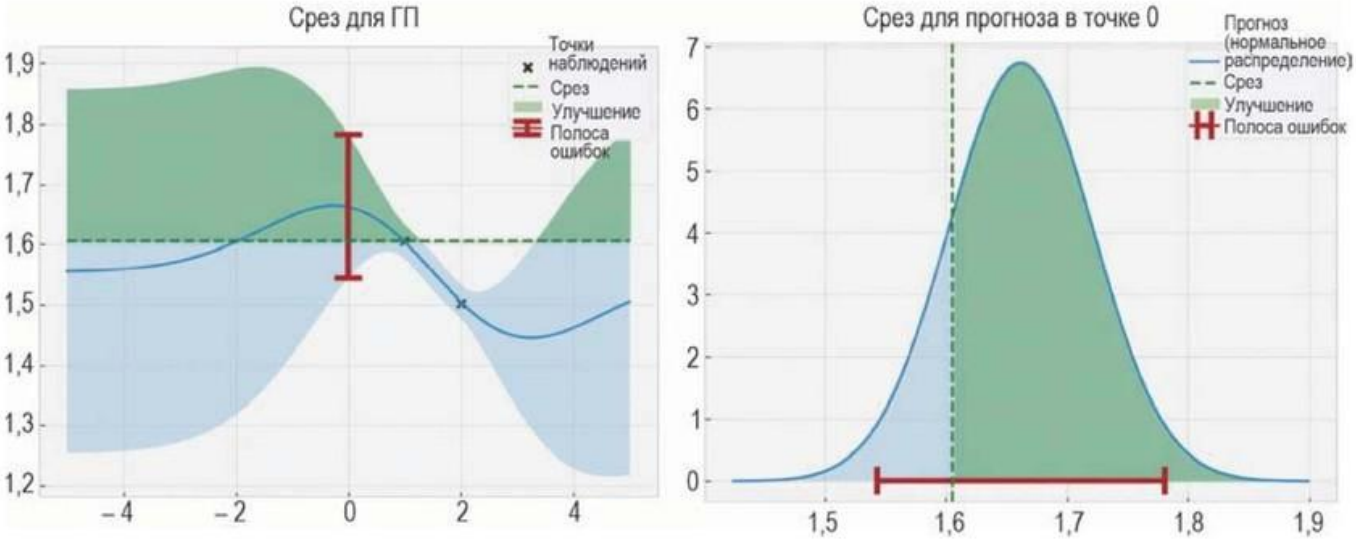


Рис. 4.9. Улучшение по сравнению с действующей точкой в значении 0 (выделено более темным оттенком): на левой части показан весь ГП, а на правой — только нормальное распределение, соответствующее прогнозу при 0 (полосы ошибок одинаковы на обеих частях); здесь улучшение по сравнению с действующей точкой следует усеченному нормальному распределению

Такие математические удобства применимы не только, когда $x = 0$. Поскольку прогноз в любой заданной точке является нормальным распределением, улучшение в любой точке также соответствует нормальному распределению, которое отсекается, как показано на рис. 4.10.

По сравнению с рис. 4.9, где большая часть нормального распределения при 0 выделена как возможные улучшения, здесь (рис. 4.10) верно следующее:

- Прогноз при –2 (центральная часть рисунка) меньше подходит, поскольку только половина из него показывает возможные улучшения. Это связано с тем, что средний прогноз при –2 примерно равен действующей точке, поэтому существует вероятность лишь 50%, что значение функции при –2 даст значение лучше, чем 1,6.
- Правая часть показывает прогноз в точке 3, где практически невозможно (в соответствии с мнением ГП относительно целевой функции) улучшить положение, поскольку почти все нормальное распределение в точке 3 лежит ниже порога действующей точки.

Это демонстрирует, что разные точки приведут к разным возможным улучшениям, в зависимости от прогнозов ГП.

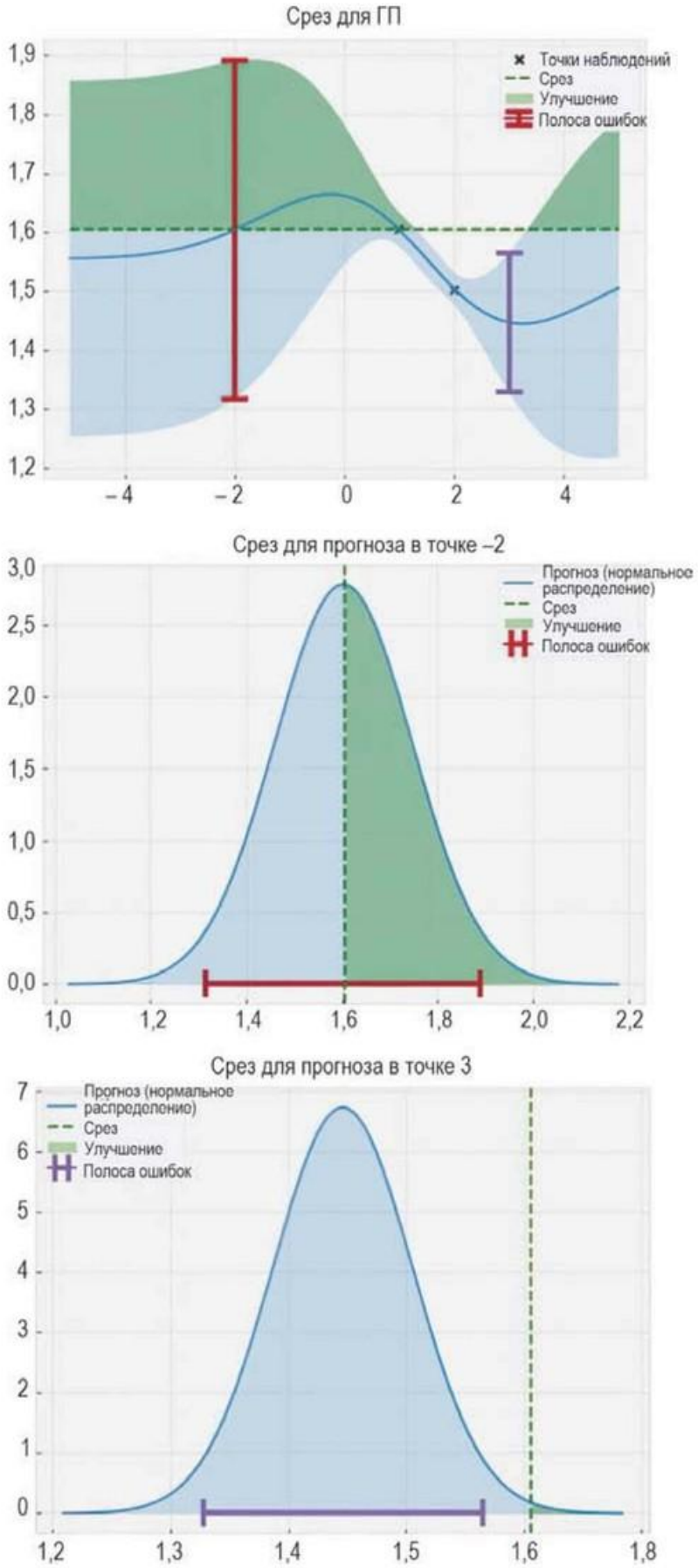


Рис. 4.10. Улучшение по сравнению с действующей точкой при значениях –2 и 3 (выделено более темным оттенком): левая часть показывает весь ГП, центральная — прогноз при –2, а правая — прогноз при 3; выделенные части показывают возможные улучшения, которые зависят от нормального распределения в данной точке

4.2.2. Вычисление вероятности улучшения

Определившись с тем, чего мы стремимся достичь в БО (улучшение действующей точки), можем перейти к обсуждению политики, направленной на достижение цели. В этом разделе мы узнаем о *вероятности улучшения* (**Probability of Improvement**, **PoI**), которая представляет собой политику, измеряющую вероятность улучшения *точки-кандидата*.

Вероятность, что точка-кандидат улучшится по сравнению с действующей, соответствует вероятности, что действующая точка является «хорошей» (рис. 4.7). Об этом также говорят следующие выводы из раздела 4.2.1:

1. В точке 0 (рис. 4.9) большая часть нормального распределения выделена как возможные улучшения (существует высокая вероятность улучшения).
2. Точка –2 (центральная часть рис. 4.10) имеет вероятность улучшения 0,5, потому что только половина ее нормального распределения превышает действующую точку.
3. Наибольшая часть нормального распределения для точки 3 (правая часть рис. 4.10) располагается ниже порога, поэтому вероятность улучшения очень мала.

Это вычисление можно сделать более конкретным. Вероятность, что действующая точка будет улучшена, равна площади выделенной части на рис. 4.9 и 4.10.

> **Определение**
>
> Вся площадь под любой нормальной кривой равна 1, поэтому площадь выделенной части на рис. 4.9 и 4.10 точно определяет вероятность, что рассматриваемая нормальная случайная величина (значение функции в данной точке) превышает действующую.

Величина, соответствующая площади выделенной области, связана с *кумулятивной функцией распределения* (**Cumulative Density Function**, **CDF**), которая определяется как вероятность, что случайная величина примет значение, меньшее или равное целевому. Другими словами, CDF измеряет площадь *неподсвеченной* области, которая равна разности между 1 и площадью *подсвеченной* области.

Благодаря тому, насколько математически удобны нормальные распределения и ГП, можно легко вычислить площадь выделенной области с помощью CDF, для которого требуется среднее значение и стандартное отклонение рассматриваемого нормального распределения. В вычислительных целях можно использовать класс `torch.distributions.Normal` библиотеки PyTorch, который реализует нормальное распределение и предлагает полезный метод `cdf()`. Предположим, нас интересует расчет вероятности, что в 0 точка может дать улучшение по сравнению с действующим значением. Мы будем следовать процедуре, описанной на рис. 4.11:

1. Использовать ГП для вычисления прогнозов среднего значения и стандартного отклонения в точке 0.
2. Вычислить площадь под нормальной кривой, определяемой предыдущими средним значением и стандартным отклонениями. При этом действующее значение является пороговым. Для этого расчета используется CDF.
3. Вычесть значение CDF из 1. Таким образом, получится PoI для точки-кандидата.

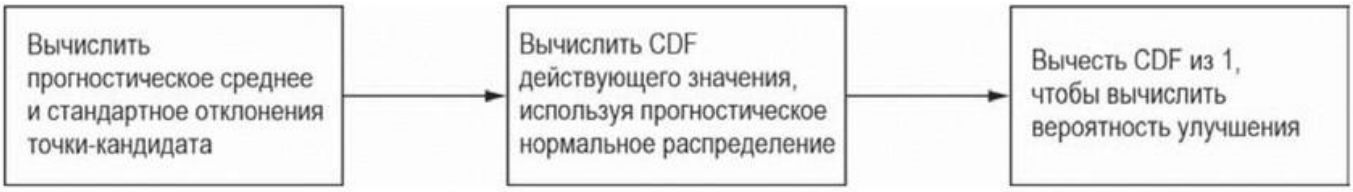


Рис. 4.11. Блок-схема расчета оценки PoI: она помогает вычислить, насколько вероятно, что любая точка-кандидат сможет улучшиться относительно действующей

ПРИМЕЧАНИЕ

Технически CDF вычисляет площадь части нормального распределения, которая расположена слева от порога. Поэтому, чтобы получить площадь части, которая расположена справа и соответствует возможным улучшениям, нужно из 1 вычесть выходные данные CDF.

Сначала сгенерируем прогноз ГП:

```
with torch.no_grad():
    predictive_distribution = likelihood(model(torch.tensor([[0.]])))
    #Прогностическое среднее значение при 0
    predictive_mean = predictive_distribution.mean
    #Прогностическое стандартное отклонение при 0
    predictive_sd = predictive_distribution.stddev
```

Затем инициализируем одномерное нормальное распределение с соответствующим средним значением и стандартным отклонением:

```
normal = torch.distributions.Normal(predictive_mean, predictive_sd)
```

Это нормальное распределение показано на правой части рис. 4.9. Наконец, чтобы вычислить площадь выделенной части, вызываем метод `cdf()`, используя в качестве входных данных действующее значение (максимальное значение обучающих данных — `train_y.max()`), и вычитаем результат из 1:

```
>>> 1 - normal.cdf(train_y.max())
tensor([0.8305])
```

В точке 0 есть более 80% шансов на улучшение по сравнению с действующей точкой. Это согласуется с фактом, что на рис. 4.9 выделена наибольшая часть нормального распределения. Используя те же вычисления, можно определить, что PoI составляет 0,4948 при –2 и 0,0036 при 3. Глядя на рис. 4.10, видно, что эти цифры имеют смысл. Кроме рассмотренных трех точек (0, –2 и 3), используя ту же формулу, можно вычислить, насколько вероятно, что любая точка (а именно ее PoI) сможет улучшиться в пространстве поиска.

ОПРЕДЕЛЕНИЕ

Процедура, показанная на рис. 4.11, помогает оценить политику PoI. Оценка каждой точки в пространстве поиска равна вероятности, что эта точка может улучшиться относительно действующей. Ее также называют *оценкой приобретения*, поскольку мы используем ее как метод оценки данных.

Несмотря на то что политика PoI использует метод `cdf()`, для этой задачи проще использовать BoTorch. Библиотека построена на базе PyTorch и GPyTorch и пре-

красно работает с ними. Как мы видели ранее, нужно изменить всего две строки кода в классе ГП, чтобы сделать модель совместимой с BoTorch. Кроме того, библиотека реализует свои политики в виде *модулей*, что позволяет легко заменять их внутри и снаружи цикла БО.

Модульность политик BoTorch

Под *модульностью* подразумевается, что используемую в данный момент политику можно заменить любой другой прямо в цикле БО. Для этого нужно изменить только ее инициализацию. Остальная часть цикла (обучение модели, визуализация хода оптимизации и обновление данных обучения) не требует изменений. Подробнее модульность рассматривалась в функциях среднего и ядрах GPyTorch в разделах 3.3 и 3.4.2.

Реализовать политику PoI с помощью BoTorch можно следующим образом:

```
#Объявляем политику PoI
policy = botorch.acquisition.analytic.ProbabilityOfImprovement(
    model, best_f=train_y.max()
)


#Вычисляем оценку
with torch.no_grad():
    scores = policy(xs.unsqueeze(1))
```

Класс `ProbabilityOfImprovement` реализует PoI как модуль PyTorch, принимая ГП в качестве первого аргумента и действующую точку в качестве второго. Переменная `scores` теперь хранит оценки политики PoI для точек из переменной `xs`, которые представляют собой плотную сетку в диапазоне от –5 до 5.

ГП предполагает, что оценка приобретения каждой точки равна вероятности, что та улучшится относительно действующей. На рис. 4.12 показана данная оценка в пространстве поиска вместе с ГП.

Можно наблюдать интересное поведение показателей PoI:

- Небольшой регион слева от действующей точки (от 0 до 1) имеет относительно высокие показатели PoI. Это соответствует высоким средним прогнозам в данной области.
- Регион на левой части графика имеет более низкие оценки. Это связано с тем, что средние прогнозы не так высоки. Но здесь достаточно неопределенности, поэтому существует значительная вероятность улучшения ситуации.
- Область возле 2 имеет PoI, близкий к 0. Прогностические нормальные распределения точек здесь в основном лежат ниже действующего порога.

Теперь осталось определить точку между –5 и 5, которая имеет самый высокий показатель PoI. Другими словами, точку, которая максимизирует вероятность улучшения относительно действующей.

Воспользуемся вспомогательной функцией `botorch.optim.optimize.optimize_acqf()`, которая находит точку, максимизирующую оценку любой политики. Для этого используем следующий код, который является частью реализации цикла БО:

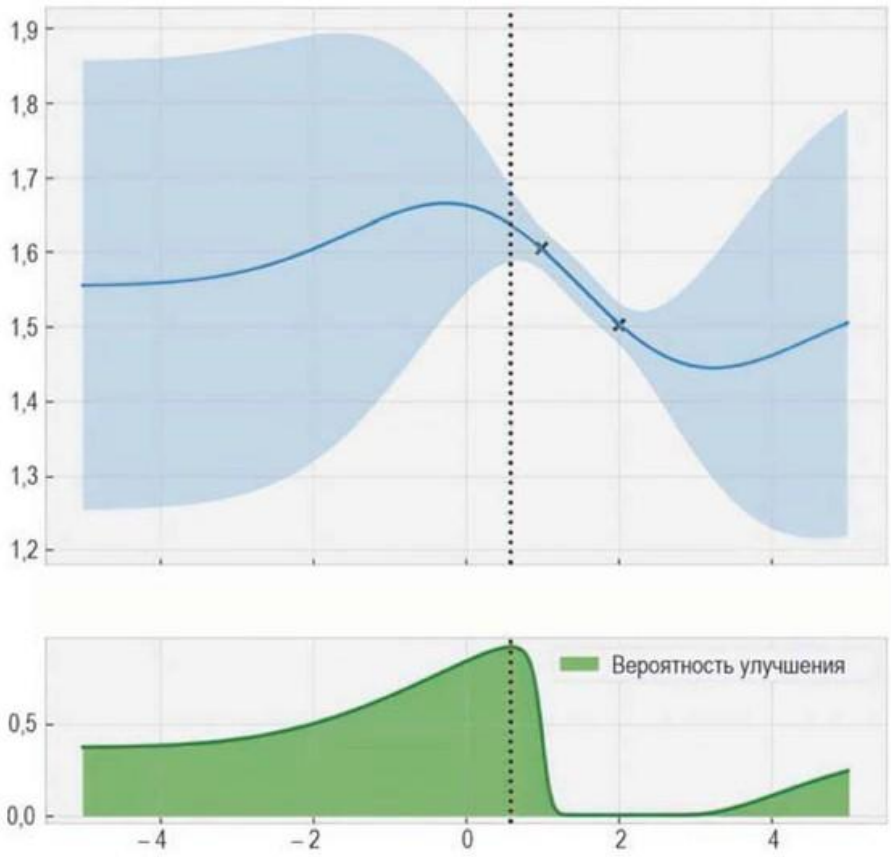


Рис. 4.12. Прогнозы ГП (вверху) и политика PoI (внизу): пунктирная линия указывает на точку, которая максимизирует оценку PoI; в этот момент мы запрашиваем целевую функцию на следующей итерации оптимизации

```
next_x, acq_val = botorch.optim.optimize.optimize_acqf(
    policy,
    bounds=torch.tensor([[-bound], [bound]], dtype=torch.float),
    q=1,
    num_restarts=10,
    raw_samples=20,
)
```

Возвращаемые значения — это местоположение точки с максимальной оценкой приобретения, которую нашла L-BFGS, и максимальная оценка приобретения соответственно. При проверке имеем следующее:

```
>>> next_x, acq_val
(tensor([[0.5985]]), tensor(0.9129))
```

Эти выходные данные показывают, что значение точки-кандидата, максимизирующей оценку PoI при 0,91, составляет примерно 0,6. Это соответствует пунктирной вертикальной линии на рис. 4.12. На данном этапе мы запрашиваем (оцениваем) целевую функцию, чтобы получить следующую точку данных в БО.

Кандидат с самым высоким прогностическим средним значением

Интересно, что выбранная для запроса точка (примерно 0,6) имеет не самое высокое прогностическое среднее значение (около –0,5). Она имеет более низкий PoI, поскольку неопределенность на данный момент высока. Поэтому у нее меньше шансов улучшить-

ся по сравнению с действующей точкой, несмотря на высокое прогностическое среднее значение.

Это все, что нужно для принятия решения, какую точку запрашивать, используя политику PoI за одну итерацию БО. Но важно помнить о цикле на рис. 4.2. Там мы поочередно находим следующую точку данных для запроса, используя политику (шаг 2), и обновляем ГП новыми данными (шаги 1 и 3). Рассмотрим это подробнее в разделе 4.2.3.

4.2.3. Запуск политики PoI

В этом разделе мы наконец запустим политику PoI и проанализируем ее поведение. Повторим весь процесс еще раз. Сначала обучаем модель, затем объявляем политику PoI и используем `optimize_acqf()` для поиска лучшей точки. Повторяем несколько раз, пока не достигнем условия завершения алгоритма. Цикл реализован в файле `CH04/01 – BayesOptloop.ipynb`. Теперь нужно инициализировать политику PoI в соответствующем цикле `for`.

Этот код создает серию графиков, которые генерируются вспомогательной функцией `visualize_gp_belief_and_policy()`. Каждый из них показывает текущее состояние цикла БО на протяжении 10 выполняемых запросов. Графики выглядят аналогично рис. 4.12 с добавлением целевой функции для справки:

```
num_queries = 10
for i in range(num_queries):
    print("iteration", i)
    print("incumbent", train_x[train_y.argmax()], train_y.max())

    model, likelihood = fit_gp_model(train_x, train_y)

    #Политика PoI
    policy = botorch.acquisition.analytic.ProbabilityOfImprovement(
      model, best_f=train_y.max()
    )

    next_x, acq_val = botorch.optim.optimize_acqf(
    ... #Опустим некоторые детали
```

Количество оценок функции в БО

Количество запросов, которые используются в БО, полностью зависит от того, сколько оценок функций мы можем себе позволить. В разделе 1.1 описана проблема дорогостоящей оптимизации «черного ящика». Она предполагает, что количество запросов, которые можно сделать, относительно невелико из-за стоимости вычисления функции.

Существуют и другие критерии, которые можно использовать для определения момента, когда следует завершить цикл БО. Например, можно остановиться, когда достигнуто целевое значение или когда по последним 5 или 10 запросам не произошло значительного улучшения. На протяжении всей книги мы придерживаемся предположения, что у нас есть заранее определенное количество оценок функции, которые можно выполнить.

По умолчанию мы используем 10 запросов при запуске политик для одномерной функции Форрестера и проверке их поведения. Упражнение 2 в этой главе посвящено двумерной функции и использует 20 запросов.

На рис. 4.13 показаны графики на первой, пятой и последней итерациях. На них видно, что политика PoI остается внутри области между 0 и 2. На 10-й (последней) итерации достигнут локальный оптимум. Это означает, что нет возможности адекватно исследовать пространство поиска и, следовательно, глобальный оптимум целевой функции в нужной области (в районе 4) мы упустили.

Предупреждение BoTorch при использовании вспомогательной функции *optimize_acqf()*

При запуске кода для БО совместно с политикой PoI вы можете получить следующее предупреждение от BoTorch:

```
RuntimeWarning: Optimization failed in
`gen_candidates_scipy` with the following warning(s):
[OptimizationWarning('Optimization failed within `scipy.optimize.minimize`
with status 2 and message ABNORMAL_TERMINATION_IN_LNSRCH.')]
Trying again with a new set of initial conditions.
 warnings.warn(first_warn_msg, RuntimeWarning)
```

Оно отображается, когда вспомогательная функция `optimize_acqf()` (в частности, подпрограмма поиска строки) не может успешно оптимизировать оценку приобретения (в данном случае оценку PoI). Этот сбой часто происходит, когда функция оценки приобретения не имеет достаточной «гладкости» (например, на последней части рис. 4.13, где имеется резкий пик около $x = 1{,}5$), что делает численную оптимизацию нестабильной.

Не вдаваясь в подробности процедуры оптимизации, можно прибегнуть к увеличению количества перезапусков (аргумент `num_restarts`) и количества необработанных выборок (аргумент `raw_samples`) при использовании `optimize_acqf()`, что увеличивает шансы найти точку данных с наивысшей оценкой приобретения.

Для простоты изложения с этого момента отключим это предупреждение при запуске `optimize_acqf()` в нашем коде, используя контекстный менеджер с модулем `warnings` следующим образом:

```
with warnings.catch_warnings():
    warnings.filterwarnings('ignore', category=RuntimeWarning)
    next_x, acq_val = botorch.optim.optimize_acqf(...)
```

> **Примечание**
>
> Производительность PoI на рис. 4.13 может не оправдать ваших ожиданий. В конце концов, потрачено много времени на создание данной политики, которая кажется слишком «эксплуататорской». Тем не менее анализ происходящего может дать представление, как усовершенствовать PoI и улучшить производительность.

Несмотря на то что PoI застревает в локальном оптимуме, она выполняет то, что должна. Поскольку политика стремится к улучшению в сравнении с нынешней действующей точкой, можно обнаружить, что медленное движение вправо от 1 приведет к улучшению с высокой вероятностью. Несмотря на то что, перемещаясь

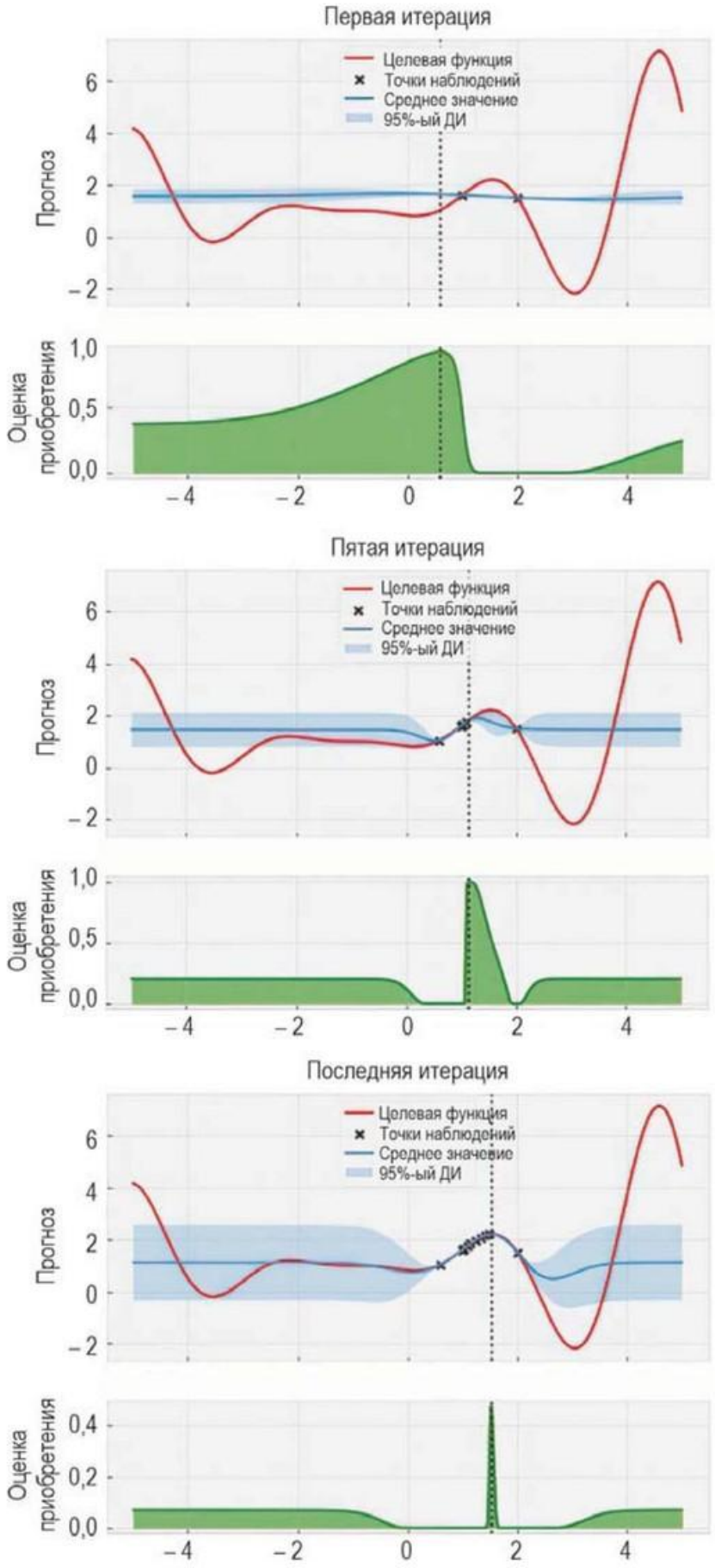


Рис. 4.13. Прогресс, достигнутый политикой PoI: поскольку политика направлена на улучшение любого масштаба, прогресс застревает на локальном оптимуме возле значения 2, и мы не можем исследовать другие области пространства поиска

вправо, PoI постоянно находит все больше улучшений, такое поведение можно считать слишком «эксплуататорским», поскольку политика недостаточно тщательно изучает другие регионы.

Важное примечание

Даже несмотря на то, что действия PoI соответствуют изначальной цели (улучшить действующую точку), полученное в результате поведение — не то, что нам нужно. Это означает, что нас не должно волновать простое стремление к каким-либо улучшениям со стороны действующего значения.

Есть два способа исправить такое чрезмерно эксплуататорское поведение. Первый — ограничить то, что подразумевается под улучшением. Эксперимент с PoI показывает, что на каждой итерации политика получает лишь незначительное улучшение по сравнению с действующим значением, медленно двигаясь в том направлении, в котором, по мнению ГП, функция движется вверх.

Второй способ: если переопределить то, что подразумевается под улучшением, обозначив, что оно *действительно* только в том случае, когда оно, по крайней мере, на ε больше, чем текущее, а также соответствующим образом модифицировать PoI, тогда политика с большей вероятностью будет исследовать пространство более эффективно. Это связано с тем, что ГП будет знать, что сохранение локального оптимума не приведет к значительному улучшению действующей точки. Рис. 4.14 иллюстрирует это предположение.

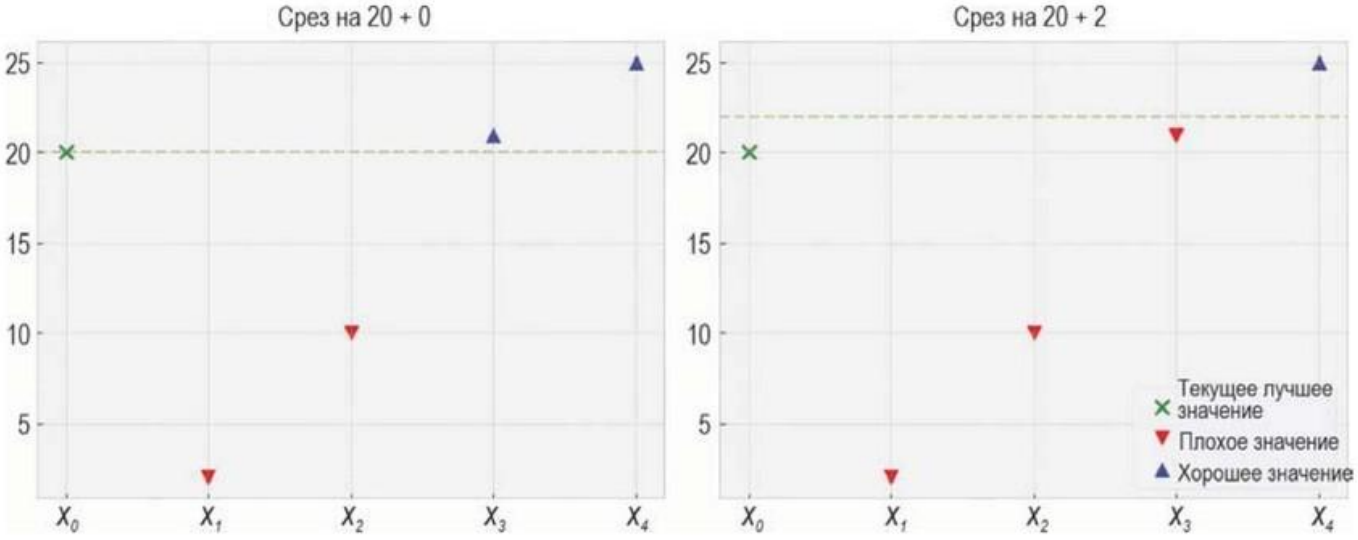


Рис. 4.14. Нахождение более строгого определения для улучшения, которое должно быть больше относительно действующей точки как минимум на $\varepsilon = 0$ (слева) и $\varepsilon = 2$ (справа): чем выше требование, тем больше политика PoI склоняется в сторону исследования (подробности в упражнении 1)

Здесь мы не будем вдаваться в подробности, но в упражнении 1 данный подход присутствует. Интересно, что чем больше улучшений мы требуем от PoI, тем политика становится более «исследовательской».

4.3. Оптимизация ожидаемого значения улучшения

Как мы уже видели, наивное стремление улучшить действующую точку приводит к чрезмерной эксплуатации политикой PoI. Это связано с тем, что простое отклонение от действующей точки на небольшую величину в соответствующем направлении может привести к высокой *вероятности улучшения*. Поэтому ее оптимизация — это не то, чем нужно заниматься. В этом разделе мы научимся дополнительно учитывать *магнитуду* (**Magnitude**) возможных улучшений, которые мы можем наблюдать. Нам важно, *насколько* можно улучшить ситуацию относительно действующей точки. Это приводит нас к одной из самых популярных политик БО: *ожидаемому улучшению* (**Expected Improvement**, **EI**).

Стремление объяснить важность улучшения очевидна. Рассмотрим пример на рис. 4.15.

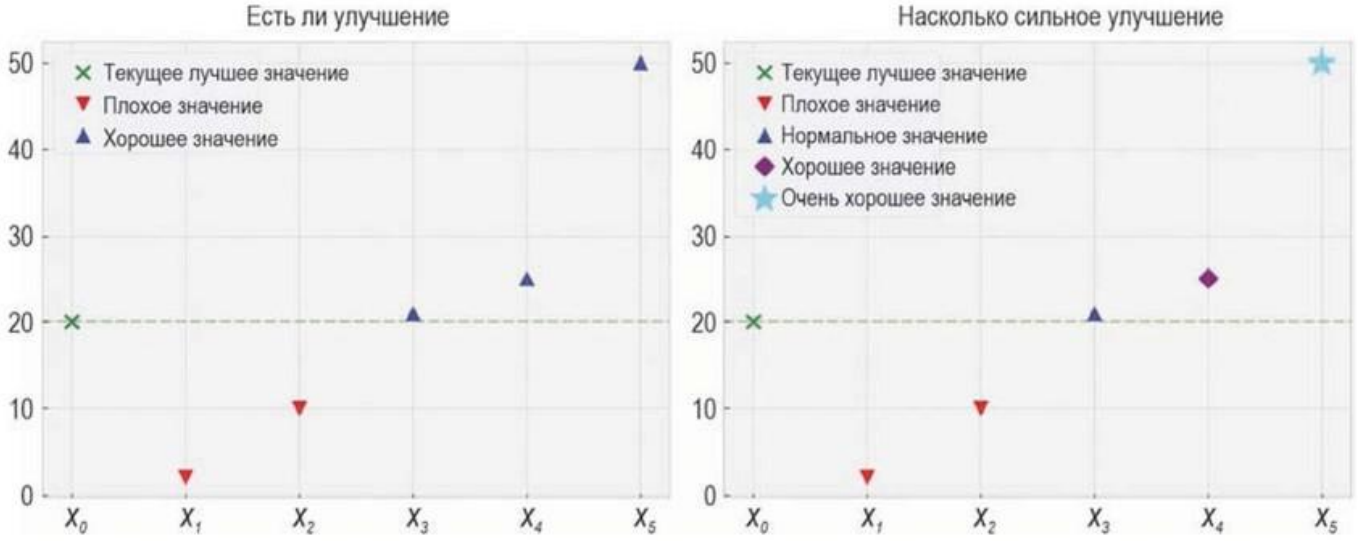


Рис. 4.15. Разница между PoI (слева) и EI (справа): первая показывает лишь наличие улучшения относительно действующего значения, а вторая учитывает, насколько сильно произойдет улучшение

На левой части рисунка показаны вычисления политики PoI, которая учитывает лишь факт, улучшается ли каждая точка данных-кандидатов относительно действующей. Таким образом, точка, которая улучшается на небольшую величину, и точка, которая имеет значительные улучшения, рассматриваются одинаково.

С другой стороны, правая часть показывает, что произойдет, если также принять во внимание магнитуду (величину) возможных улучшений. Точки x_1 и x_2 по-прежнему считаются неподходящими, поскольку они не улучшаются относительно действующей x_0. Точка x_4 считается более подходящей, чем x_3, поскольку она дает большее улучшение. Аналогичным образом x_5 будет лучшим выбором из пяти точек-кандидатов.

Это не означает, что теперь можно просто разработать политику, которая выбирает точку данных, предлагающую наибольшее улучшение относительно действующей. Нам все еще нужно знать, какое улучшение (если оно вообще будет) мы увидим. А обнаружить его можно только после фактического выполнения запроса к целевой функции.

Примечание

Несмотря на то что мы не знаем точного значения улучшения, которое будем наблюдать, мы можем рассуждать о величине улучшения каждой точки-кандидата вероятностным образом.

Стоит напомнить, что на рис. 4.9 и 4.10 изображено усеченное нормальное распределение, отражающее улучшение, которое будет наблюдаться в определенной точке. Вычисляя площадь выделенного региона, мы получим вероятность улучшения точки относительно действующей. Это даст нам политику PoI. Однако можно выполнить и другие вычисления.

Определение

Помимо PoI, мы можем вычислить *ожидаемое значение* (**Expected value**) случайной переменной, соответствующей выделенному региону. Тот факт, что мы имеем дело с усеченными нормальными распределениями, позволяет вычислять ожидаемое значение в закрытой форме. Политика БО, которая оценивает точки данных с использованием такой меры, называется *ожидаемым улучшением* (**Expected Improvement**, **EI**).

Несмотря на то что формула в закрытой форме для оценки EI не так проста, как CDF для PoI, ее по-прежнему легко вычислить. Интуитивно использование ожидаемого значения улучшения может позволить сбалансировать исследование и эксплуатацию лучше, чем использование PoI. В конце концов, точки вокруг действующей могут улучшиться с высокой вероятностью, но их улучшения, скорее всего, будут минимальными (что мы и наблюдали эмпирически в эксперименте).

Точка, находящаяся далеко и о которой мало что известно, может дать меньшую вероятность улучшения. Но если существует вероятность, что она даст большое улучшение, EI может присвоить ей более высокую оценку. Другими словами, PoI можно считать не склонной к риску политикой, которая заботится об улучшении ситуации относительно действующей точки. В случае с EI, каким бы незначительным ни было улучшение, она отлично балансирует между риском и вознаграждением и помогает найти точку, которая наилучшим образом соответствует компромиссу. Это показано на рис. 4.16, где сравниваются PoI и EI с использованием одного и того же набора данных и обученного ГП.

Можно наблюдать, что точка-кандидат, выбранная PoI (около 0,6), отличается от выбранной EI (около –0,7). Первая близка к текущей действующей точке, поэтому вполне вероятно, что обращение к ней поможет получить улучшение. Однако EI видит, что в других регионах, далеких от действующего, существует большая неопределенность. Это может привести к большему улучшению ситуации. Поэтому EI отдает предпочтение точкам, которые обеспечивают лучший баланс между исследованием и эксплуатацией.

Еще одно приятное свойство EI, которое демонстрирует данный баланс, — это то, как она присваивает оценки приобретения точкам данных с одинаковым прогностическим средним значением или стандартным отклонением. Это делается следующими способами:

- Если две точки данных имеют одинаковое прогностическое среднее, но разные стандартные отклонения, более высокую оценку получит та, которая имеет большую неопределенность. Таким образом, политика поощряет исследования.

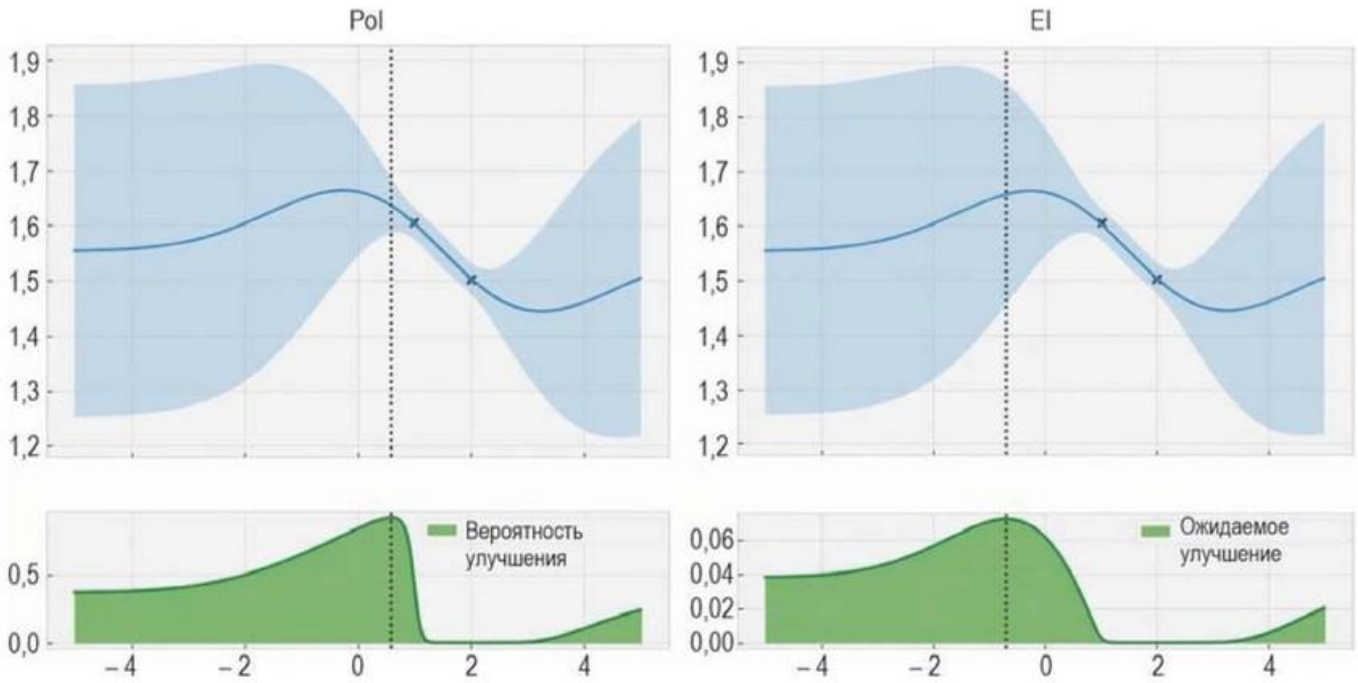


Рис. 4.16. Разница между PoI (слева) и EI (справа): EI лучше балансирует между исследованием и эксплуатацией

- Если две точки данных имеют одинаковые стандартные отклонения, но разные средние значения, более высокую оценку получит та, которая будет иметь более высокое среднее значение. Таким образом, политика вознаграждает эксплуатацию.

Это является необходимым условием политики, поскольку она предпочитает *исследование* при прочих равных условиях, когда средства прогнозирования равны, а также *эксплуатацию*, когда равны неопределенности. Это свойство мы будем более подробно рассматривать в главе 5 с другой политикой БО, которая известна под названием *верхняя доверительная граница* или *верхний доверительный предел* (**Upper Confidence Bound**, **UCB**).

С помощью вычислений мы можем инициализировать EI как объект политики, используя код, который почти идентичен коду для PoI:

```
policy = botorch.acquisition.analytic.ExpectedImprovement(
    model, best_f=train_y.max()
)
```

Теперь повторим весь цикл БО с политикой EI, убедившись, что исходный набор данных является аналогичным. В результате получится рис. 4.17, который стоит сравнить с рис. 4.13.

Несмотря на то что EI по-прежнему первоначально фокусируется на локальном оптимальном регионе возле точки 2, политика быстро исследует другие регионы в пространстве поиска с целью найти более крупные улучшения относительно действующей точки. На пятой итерации видно, что сейчас проверяется область слева. В конце после всех 10 запросов EI успешно определил глобальный оптимум целевой функции, превзойдя PoI (в предыдущем разделе).

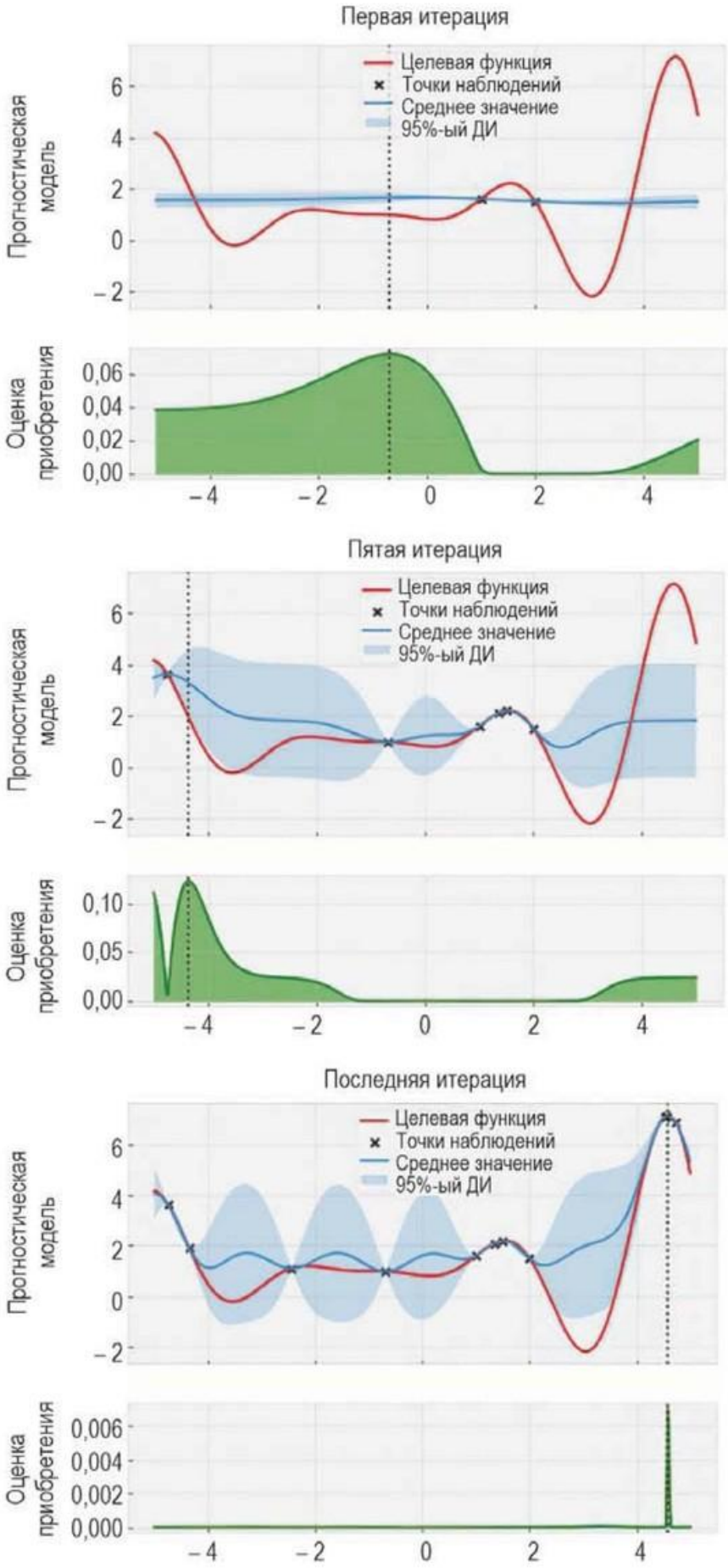


Рис. 4.17. Прогресс политики EI: она уравновешивает исследование и эксплуатацию лучше, чем PoI, и в конце находит глобальный оптимум

Примечание

Благодаря своей простоте и естественному балансу между исследованием и эксплуатацией EI является одной из наиболее (если не самой частой) используемых политик в БО. Она является хорошим вариантом по умолчанию, если нет оснований задействовать другие политики.

4.4. Упражнения

Эта глава содержит два упражнения:

1. Первое затрагивает использование PoI для исследования через изменение определения улучшения.
2. Второе описывает настройку гиперпараметров с использованием БО в целевой функции, которая имитирует поверхность точности для модели SVM.

4.4.1. Упражнение 1: поощрение исследования с помощью PoI

Один из способов справиться с тенденцией PoI к чрезмерной эксплуатации — установить более высокую планку для того, что представляет собой улучшение. В частности, мы увидели, что наивный поиск точек, которые максимизируют вероятность улучшения относительно действующей точки, не позволяет избежать локального оптимума.

В качестве решения этой проблемы можно изменить политику, указав, что нам подходят только те улучшения, которые превосходят действующее значение минимум на ε. Это побудит PoI искать улучшения в других регионах пространства поиска, как только местный регион будет достаточно охвачен. Это упражнение, реализованное в `CH04/02 - Exercise 1.ipynb`, знакомит нас с модификацией данной политики и демонстрирует ее положительное влияние на PoI. Шаги для выполнения упражнения:

1. Воссоздайте цикл БО (`CH04/01 – BayesOptloop.ipynb`), который использует одномерную функцию Форрестера в качестве целевой функции.
2. Перед циклом `for`, реализующим БО, объявите переменную с именем `epsilon`. Она будет действовать как минимальный порог улучшения, стимулирующий исследования. Установите для этой переменной значение 0,1.
3. Внутри цикла `for` инициализируйте политику PoI, как и раньше, но на этот раз укажите, что действующий порог, установленный аргументом `best_f`, представляет собой сумму значения действующей точки и значения, хранящиеся в `epsilon`.
4. Перезапустите программу и посмотрите, приведет ли это изменение к повышению производительности оптимизации по сравнению с исходной политикой PoI путем поощрения дополнительных исследований.
5. Насколько более исследовательской PoI станет, во многом зависит от минимального порога улучшения, хранящегося в `epsilon`. Установите для этой переменной значение 0,001 и отметьте факт, что недостаточно большой порог улучшения не обязательно будет способствовать успешному исследованию. Что произойдет, если для этого значения установить значение 0,5?

6. На предыдущем шаге можно было наблюдать, что установка подходящего значения порога улучшения имеет решающее значение для PoI. Однако неясно, как это сделать в нескольких приложениях и целевых функциях. Разумная эвристика состоит в том, чтобы динамически устанавливать его на некоторый процент α от действующего значения, указывая, что нам необходимо увеличение действующего значения на $1 + \alpha$. Реализуйте это в коде с требованием улучшения на 110%.

4.4.2. Упражнение 2: БО для настройки гиперпараметров

В этом упражнении (`CH04/03 - Exercise 2.ipynb`) БО применяется к целевой функции, которая моделирует поверхность точности модели SVM в задаче по настройке гиперпараметров. Ось X обозначает значение штрафного параметра C, а ось Y — значение параметра γ в ядре RBF. Более подробную информацию можно посмотреть в упражнении к главе 3. Шаги для выполнения:

1. Воссоздайте цикл БО (`CH04/01 – BayesOptloop.ipynb`):
   - Функция Форрестера нам больше не нужна. Вместо этого можно скопировать код двумерной функции, описанной в упражнении к главе 3, и использовать его в качестве целевой функции.
   - Обратите внимание, что областью определения этой функции является $[0, 2]^2$.
2. Объявите соответствующие тестовые данные с помощью переменной `xs` для двумерной сетки, представляющей домен, и переменной `ys` для значений функции `xs`.
3. Модифицируйте вспомогательную функцию, которая визуализирует ход оптимизации. Для одномерных целевых функций легко визуализировать прогнозы ГП вместе с оценками приобретения. Для этой двумерной целевой функции вспомогательная функция сгенерирует график из двух частей: одна должна отображать основную истину, а другая — оценки приобретения. На обеих частях также должны отображаться отмеченные данные. График должен выглядеть примерно так, как показано на рис. 4.18.
4. Скопируйте класс ГП из упражнения к главе 3, которое реализует ядро Матерна 2.5 с ARD. В дальнейшем измените этот класс, чтобы сделать его интегрируемым с BoTorch.
5. Повторно используйте вспомогательную функцию `fit_gp_model()` и цикл `for`, реализующий БО:
   - Исходный набор обучающих данных должен содержать точку в середине области (1, 1).
   - Поскольку пространство поиска является двумерным, сделайте поиск точки, которая максимизирует оценку приобретения, более исчерпывающим, установив `num_restarts = 40` и `raw_samples = 100` в функции `botorch.optim.optimize_acqf()`.
   - Установите количество запросов, которые можно сделать, равным 20.

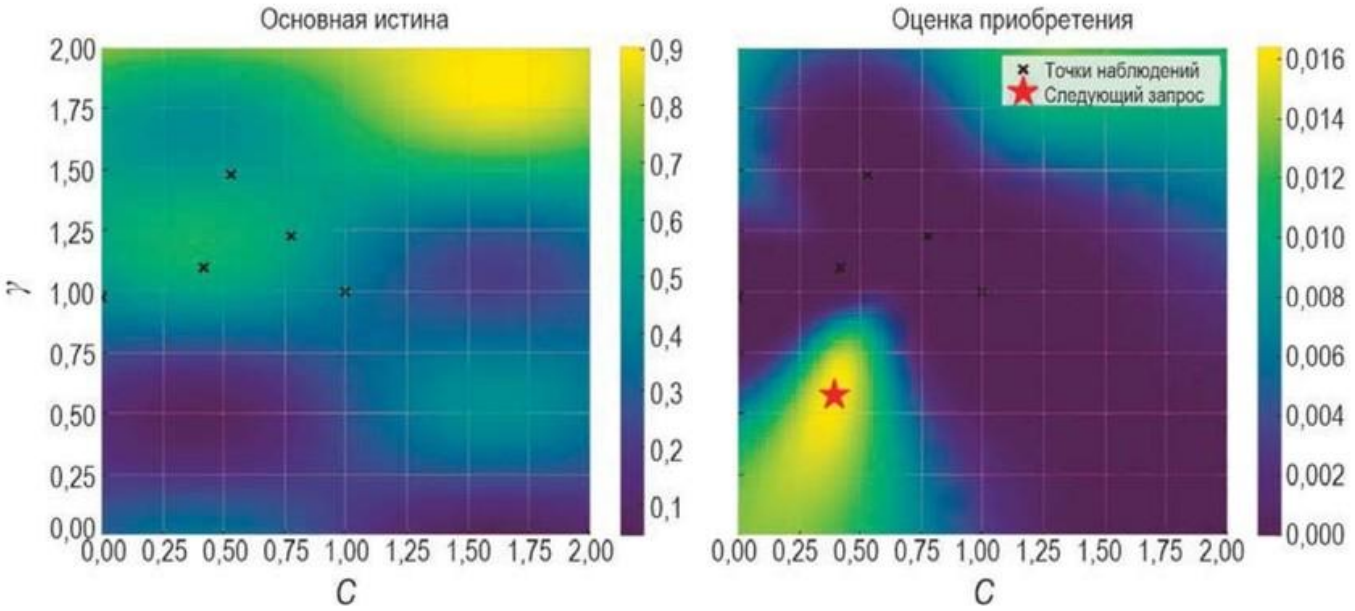


Рис. 4.18. Как должна выглядеть вспомогательная функция, которая визуализирует прогресс в БО: левая часть показывает функцию основной истины, в то время как правая часть демонстрирует оценку приобретения

6. Запустите политику PoI для целевой функции. Обратите внимание, что политика снова застревает на локальном оптимуме.
7. Запустите модифицированную версию PoI, в которой минимальный порог улучшения установлен на уровне 0,1:
   - Более подробную информацию об установке минимального порога улучшения для PoI можно найти в упражнении 1.
   - Обратите внимание, что такая модификация снова приводит к повышению производительности оптимизации.
   - На какой итерации мы достигнем точности не менее 90%? Какие параметры модели обеспечивают такую точность?
8. Запустите политику EI для целевой функции:
   - Обратите внимание, что данная политика превосходит PoI.
   - На какой итерации мы достигнем точности не менее 90%? Какие параметры модели обеспечивают такую точность?
9. Проверка производительности политики на основе одного запуска БО может ввести в заблуждение. Лучше повторить эксперимент несколько раз с разными исходными данными:
   - Реализуйте идею повторных экспериментов и визуализируйте средние действующие значения и полосы ошибок по 10 экспериментам.
   - Каждый эксперимент должен начинаться с одной точки данных, равномерно выбранной из пространства поиска.
   - Запустите перечисленные политики и сравните их производительность.

На этом можно закончить главу о политиках БО, основанных на улучшениях. Важно помнить о коде, который реализует цикл БО для функции Форрестера, исполь-

зованный здесь, поскольку мы будем задействовать его снова для сравнения с другими политиками в будущих главах. В частности, в главе 5 мы узнаем о политике, которая связана с проблемой «многоруких бандитов».

Заключение

- Политика БО использует обученный ГП для оценки, насколько важна каждая точка данных в стремлении найти оптимум целевой функции. Оценка, рассчитанная с помощью политики, называется оценкой приобретения.
- На каждой итерации цикла БО гауссовский процесс обучается на наблюдаемых данных. Политика предлагает новую точку для запроса, и ее метка добавляется в обучающий набор. Это повторяется до тех пор, пока мы не сможем больше выполнять вычисления функций.
- Модель GPyTorch требует лишь минимальных изменений для интеграции в BoTorch.
- BoTorch предоставляет вспомогательную функцию `optimize_acqf()` из модуля `optim.optimize`, которая принимает объект политики и возвращает точку данных, максимизирующую оценку приобретения.
- Хорошая политика призвана сбалансировать исследование (изучение регионов с высокой степенью неопределенности) и эксплуатацию (использование высокопроизводительных регионов).
- Различные политики по-разному решают компромисс между исследованием и эксплуатацией. Важно проверять ход оптимизации, чтобы анализировать и корректировать производительность используемой политики.
- Эвристика, которую можно использовать в БО, заключается в поиске точек, улучшающих наиболее очевидное значение.
- Поиск точки, которая дает наибольшие шансы на улучшение по сравнению с самым очевидным значением, приводит к политике PoI.
- Поиск точки, которая дает наибольшее ожидаемое улучшение по сравнению с самым очевидным значением, приводит к политике EI.
- PoI можно рассматривать как чрезмерно эксплуататорскую и не склонную к риску политику, поскольку она направлена только на улучшение наиболее видимого значения, каким бы большим ни было улучшение. Без каких-либо дальнейших модификаций EI имеет тенденцию лучше сбалансировать исследование и эксплуатацию, чем PoI.
- Благодаря гауссовскому подходу к значениям функции вычисление оценок с помощью PoI и EI может выполняться в закрытой форме. Таким образом, можно легко вычислить и оптимизировать оценки, определенные этими политиками.

5

Исследование пространства поиска с помощью «многорукого бандита»

Темы главы:

- ❒ Проблема «многорукого бандита» и ее связь с БО.
- ❒ Политика верхней доверительной границы.
- ❒ Политика выборки Томпсона.

В какой игровой автомат стоит играть в казино, чтобы максимизировать свой выигрыш? Как разработать стратегию, позволяющую разумно опробовать несколько игровых автоматов и выбрать наиболее прибыльный? Какое отношение эта задача имеет к БО? Данная глава поможет ответить на эти вопросы.

В главе 4 мы познакомились с политиками, которые определяют, как стоит исследовать и проверять пространство поиска. Стратегия должна приводить к оптимуму целевой функции, которую нужно оптимизировать. Две политики, которые были подробно рассмотрены, — это «Вероятность улучшения» (PoI) и «Ожидаемое улучшение» (EI). В их основе лежит идея необходимости улучшить результат по сравнению с наилучшим целевым значением, которое наблюдается к настоящему моменту. Это лишь одна из эвристик, и она не является единственной в БО.

В этой главе речь пойдет еще о двух политиках, напрямую касающихся проблемы принятия решений, которая называется *«многорукий бандит»* (**Multi-Armed Bandit**, **MAB**). Данную задачу можно сформулировать как вопрос: «На каком игровом автомате выгоднее всего играть в казино?». Она создает основу для многих задач принятия решений в условиях неопределенности. MAB имеет долгую историю и обширные исследования. На этой основе было разработано множество хороших решений. Многорукий бандит и БО очень похожи: они обе имеют дело с принятием решений в условиях неопределенности с целью оптимизации. Поэтому ожидается, что решения для MAB будут хорошо работать и для БО.

Сначала мы кратко поговорим о данной теме и о том, как она связана с БО. Это даст некоторую информацию и представление, как байесовская оптимизация связана с другими задачами в области искусственного интеллекта. Затем мы познако-

мимся с двумя самыми популярными политиками в MAB и применим их к БО. Первая — это политика *верхней доверительной границы*, которая использует принцип *оптимизма* перед лицом неопределенности для обоснования своих решений. Вторая — *выборка Томпсона*, она представляет собой «рандомизированное» решение, активно использующее вероятностный характер прогностических моделей. В конце мы реализуем и запустим их на наших примерах и проанализируем эффективность.

К концу этой главы мы получим понимание задачи многорукого бандита и ее связи с БО. Что еще более важно, мы добавим еще два пункта в наш список политик, открывая перед собой дополнительные способы исследования пространства поиска в задачах по оптимизации «черного ящика».

5.1. Введение в задачу многорукого бандита

В этом разделе мы узнаем о MAB на высоком уровне. Начнем с обсуждения и постановки задачи.

> **Важное примечание**
>
> В MAB нам нужно выбрать *действие*, которое необходимо предпринять на каждом этапе в долгосрочной перспективе. Каждое действие приносит вознаграждение в размере определенной неизвестной ставки. Наша цель — максимизировать общее вознаграждение, которое можно получить в конце задачи.

Мы также исследуем связь с БО и другие проблемы в области искусственного интеллекта и машинного обучения. Это даст нам контекст, связывающий MAB с остальной частью книги.

5.1.1. Как найти наилучший игровой автомат в казино

Термин «многорукий бандит» может сбивать с толку, представляясь в воображении чем-то загадочным. На самом деле он относится к задаче «игрока», который делает выбор, в какие игровые автоматы играть, чтобы получить наибольшую сумму вознаграждения. Представьте, что вы находитесь в казино и видите игровой автомат с ручкой, которую можно потянуть.

Нажав на рычаг, можно получить в награду монеты. Однако в этом процессе присутствует элемент случайности. Во внутренней работе автомата запрограммирована вероятность вознаграждения (p), в соответствии с которой при каждом нажатии машина возвращает деньги. Такой игровой автомат изображен на рис. 5.1. Если $p = 0{,}5$, то он вознаграждает своих игроков в половине случаев. Если $p = 0{,}01$, только 1 из 100 попыток приведет к выдаче награды. Поскольку вероятность запрограммирована внутри машины и, следовательно, скрыта от нас, нельзя точно узнать ее значение.

В этой гипотетической ситуации автоматы запрограммированы так, чтобы скорость выдачи монет была ниже, чем скорость игры и получения монет от игроков. Поэтому даже если есть случайные победители, в среднем прибыль получает именно казино.



Рис. 5.1. Игровой автомат с рычагом, который может возвращать монеты в зависимости от вероятности вознаграждения

ОПРЕДЕЛЕНИЕ

Особенно неудачливый игрок, который теряет монеты быстрее, чем выигрывает их, может назвать игровой автомат «бандитом», думая, что тот крадет его деньги. Поскольку у машины есть «рука», которую можно тянуть, ее можно назвать «однорукий бандитом».

Теперь представьте, что есть не один игровой автомат, а целый ряд. Каждый из них имеет свою вероятность вознаграждения p, как показано на рис. 5.2.



Рис. 5.2. Ряд игровых автоматов, каждый из которых имеет свою вероятность вознаграждения

С помощью этого ряда игровых автоматов стратегический игрок может превратить данную ситуацию в задачу принятия решений и попытаться определить машину, которая имеет наибольшую вероятность получения вознаграждения. Цель здесь в том, чтобы максимизировать сумму вознаграждения, учитывая, что потянуть за рычаги этих машин можно только определенное количество раз.

ОПРЕДЕЛЕНИЕ

Эта задача принятия решений называется *многоруким бандитом* (**Multi-Armed Bandit**, **MAB**), поскольку можно тянуть за несколько рук. Цель состоит в том, чтобы разработать политику принятия решения, какую руку следует потянуть дальше, чтобы максимизировать общее вознаграждение, которое игрок получит в конце.

MAB обладает многими характеристиками, которые соответствуют задаче оптимизации в условиях неопределенности (такой как БО):

- *Возможность предпринять конкретные действия*. Например, потянуть за рычаг определенного игрового автомата.

- *Ограниченное количество действий.* Тянуть за ручки можно только определенное количество раз.
- *Существует неопределенность в результатах действий, которые можно предпринять.* Вероятность вознаграждения каждого автомата неизвестна. Ее даже нельзя оценить, пока игрок не потянет за рычаг несколько раз. Сделав несколько попыток выиграть, даже тогда нельзя точно знать, выдаст ли машина монеты, т. к. в награде присутствует случайность.
- *Цель — оптимизация целевого объекта.* Мы стремимся максимизировать совокупное вознаграждение, которое получим в конце (потянув определенное количество раз за рычаги).

Вероятно, самое важное в MAB — это то, что мы сталкиваемся с тем же компромиссом между исследованием и эксплуатацией, который обсуждался в разделе 4.1.2. Каждый раз, решая потянуть за рычаг, мы должны выбирать между автоматом, который до этого момента обеспечивал хороший уровень успеха (эксплуатация), и другими машинами, вероятности вознаграждения которых мало изучены (исследование).

Проблема, с которой мы сталкиваемся, — это баланс, потому что, склоняясь к исследованию, мы рискуем потратить силы на машины с низкой вероятностью вознаграждения, но с другой стороны, чрезмерная эксплуатация способствует тому, что мы можем полностью упустить автомат, уровень вознаграждения которого выше, чем тот, что мы наблюдаем в настоящее время. На рис. 5.3 показан пример, в котором верно следующее:

1. Мы собрали 70 монет за 100 попыток на первой машине. Другими словами, первый игровой автомат на данный момент обеспечивает самый высокий эмпирический показатель успеха — 70%.
2. Вторая машина дает нам больше данных. Наименьшая неопределенность относительно ее ставки вознаграждения составляет 50%.
3. Несмотря на то что эмпирический уровень успеха третьей машины самый низкий (0%), возможно, придется попробовать сыграть на ней большее количество раз, чтобы иметь больше уверенности в ее ставке вознаграждения.



Рис. 5.3. Пример набора данных, показывающий дилемму исследования-эксплуатации: политика MAB должна выбирать между машиной с постоянно высоким уровнем успеха и машиной с неопределенной ставкой вознаграждения

Задача политики MAB (аналогична задаче политики БО) состоит в том, чтобы просмотреть данные о прошлых вознаграждениях и решить, какую ручку следует тянуть дальше, балансируя между исследованием и эксплуатацией. Многорукий бандит моделирует широкий спектр приложений, которые можно наблюдать в реальном мире:

- При рекомендации продуктов в онлайн-магазине алгоритму необходимо выбрать из нескольких один, чтобы предложить пользователю. Каждый товар можно рассматривать как рычаг игрового автомата. А нажатие на него означает, что механизм выбирает этот продукт и показывает его пользователю. Если покупатель нажимает на рекламу с товаром, это можно рассматривать как вознаграждение, поскольку клик пользователя — это именно то, чего мы хотим достичь.
- Многие задачи управления ресурсами можно сформулировать как MAB. Например, необходимо рассмотреть, как лучше всего распределить различные ресурсы между организациями, чтобы наилучшим образом оптимизировать их стратегические цели на высоком уровне (например, прибыль или производительность), не зная заранее, насколько хорошо будет работать каждая организация. Или управление портфелем акций также можно сформулировать как подобную задачу.
- MAB также нашел применение при планировании клинических испытаний, когда каждому пациенту необходимо назначить определенное лечение. Задача — оптимизировать результат лечения для всех пациентов, имея дело с ограниченными ресурсами, и определить вероятность пользы для пациентов от данного лечения.

В каждом из подобных приложений можно предпринять ряд действий (набор рычагов, которые можно использовать) для оптимизации целевой функции в условиях неопределенности.

5.1.2. От MAB к БО

MAB и БО имеют много общих характеристик. В обеих задачах присутствует необходимость обдумывать, какое решение следует принять, чтобы максимизировать интересующий нас результат. Кроме того, итог каждого действия не является детерминированным. Это означает, что нам неизвестно, принесет ли действие хороший результат, пока не предпримем его.

Однако эти две задачи не эквивалентны. В MAB мы стремимся максимизировать совокупное вознаграждение с течением времени (общее количество монет, которые в итоге получим). В БО наша цель — просто найти такие входные данные для функции, которые приведут к высокому искомому значению (пока в наборе данных есть хорошее целевое значение, успех оптимизации гарантирован). Такую цель иногда называют *простой наградой*. Подобная разница политик означает, что в MAB мы часто получаем вознаграждения, чтобы поддерживать хорошее совокупное значение, в то время как в БО мы можем позволить себе сделать больший

упор на исследовании, чтобы найти потенциально хорошее целевое значение (вознаграждение).

> **Определение**
>
> Термин «простое вознаграждение» (**simple reward**) не означает, что цель легче или проще оптимизировать. Он подразумевает, что цель представляет собой одно число, а не сумму, которая является совокупным вознаграждением.

Кроме того, в MAB можно выполнить только конечное число действий. В БО, когда мы пытаемся оптимизировать целевую функцию в непрерывной области, существует бесконечное количество действий. Поскольку с помощью ГП мы предполагаем, что значения функций в соседних точках похожи друг на друга, можно думать об этом как о действиях, близких друг к другу (дающих одинаковые вознаграждения). Это показано на рис. 5.4, где каждая бесконечно малая точка представляет собой игровой автомат, за руку которого мы можем потянуть (запросить значение целевой функции), а машины, расположенные близко друг к другу, имеют одинаковые цвета (коррелируют).

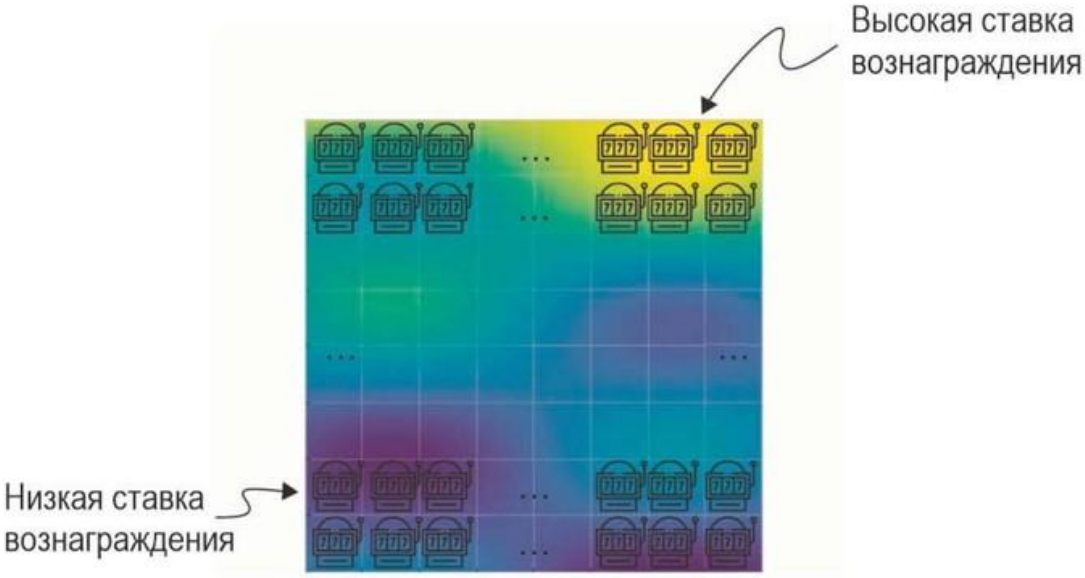


Рис. 5.4. БО аналогична MAB, но с бесконечным количеством действий: каждая бесконечно малая точка — это игровой автомат, за руку которого мы можем потянуть; кроме того, машины, расположенные близко друг к другу, коррелируют

Большинство формализаций задачи MAB подразумевают двоичную ситуацию, когда автомат либо возвращает монету, либо нет. С другой стороны, значение функции, которое наблюдается в БО, может принимать любое реальное значение.

Основные различия между MAB и БО суммированы в табл. 5.1. Несмотря на то что они являются фундаментальными, компромисс между исследованием и эксплуатацией при принятии решений присутствует в обеих политиках, поэтому разумно попытаться перепрофилировать MAB на БО.

Далее поговорим о похожих двух политиках, их мотивах и о том, как реализовать их с помощью BoTorch. В следующем разделе мы начнем с политики верхней доверительной границы.

Таблица 5.1. Различия между MAB и БО

Критерий	MAB	БО
Цель, которую нужно максимизировать	Совокупная награда	Простая награда
Тип наблюдения/награды	Бинарный	Оценочный
Количество действий	Конечное	Бесконечное
Корреляция между действиями	Нет	Есть, для схожих действий

5.2. Как быть оптимистом в условиях неопределенности с помощью политики верхней доверительной границы

Как нужно учитывать бесконечное множество возможностей для значения, которое мы наблюдаем, когда оцениваем целевую функцию в определенном месте? Кроме того, как следует рассуждать об этих возможностях простым и эффективным методом, способствующим принятию решений? В этом разделе мы исследуем и реализуем политику *верхней доверительной границы* (**Upper Confidence Bound**, **UCB**), вытекающую из MAB.

UCB действует в духе оптимизма перед лицом неопределенности. Идея состоит в том, чтобы вместо истинной (неизвестной) оценки вознаграждения использовать ее верхнюю границу. Другими словами, мы *оптимистично* оцениваем ставку вознаграждения каждого автомата, используя верхнюю границу ее предполагаемого диапазона. И выбираем ту ставку, у которой самая высокая верхняя граница.

Сначала подробнее рассмотрим сам принцип и то, как он помогает при принятии решений. После этого реализуем политику UCB с помощью BoTorch и проанализируем ее поведение.

5.2.1. Оптимизм в условиях неопределенности

Рассмотрим простой пример, чтобы конкретизировать эту идею. Предположим, вы однажды просыпаетесь и замечаете, что, несмотря на солнечную погоду, на горизонте видны темные облака. Вы проверяете погодное приложение на своем телефоне, чтобы узнать, будет ли в течение дня дождь и стоит ли брать с собой на работу зонт. К сожалению, приложение не может с абсолютной уверенностью сказать наверняка. Вместо этого оно показывает только вероятность солнечной погоды от 30 до 60%.

Вы думаете про себя, что если вероятность солнечной погоды в течение дня меньше 50%, то вы возьмете с собой зонтик. Однако однозначной оценки здесь нет, а есть диапазон от 30 до 60%. Как же тогда решить, нужен ли зонт?

Пессимист мог бы сказать, что, учитывая вероятность солнечной погоды всего в 30%, следует перестраховаться и подготовиться к худшему. Другой человек, который колеблется посередине, может дополнительно заглянуть в телефон, чтобы

узнать среднее значение вероятности и принять соответствующее решение. А с точки зрения оптимиста, вероятность в 60% является достаточной причиной полагать, что будет солнечно, поэтому он не станет брать с собой зонтик.

Эти способы мышления продемонстрированы на рис. 5.5. На нем рассуждения оптимиста соответствуют идее, лежащей в основе политики UCB: быть оптимистом при неизвестном исходе события и принимать решения, основанные на этом убеждении. В задаче MAB с помощью политики UCB мы строим верхнюю границу для возможной ставки вознаграждения каждого игрового автомата и переходим к выбору машины с самой высокой оценкой.

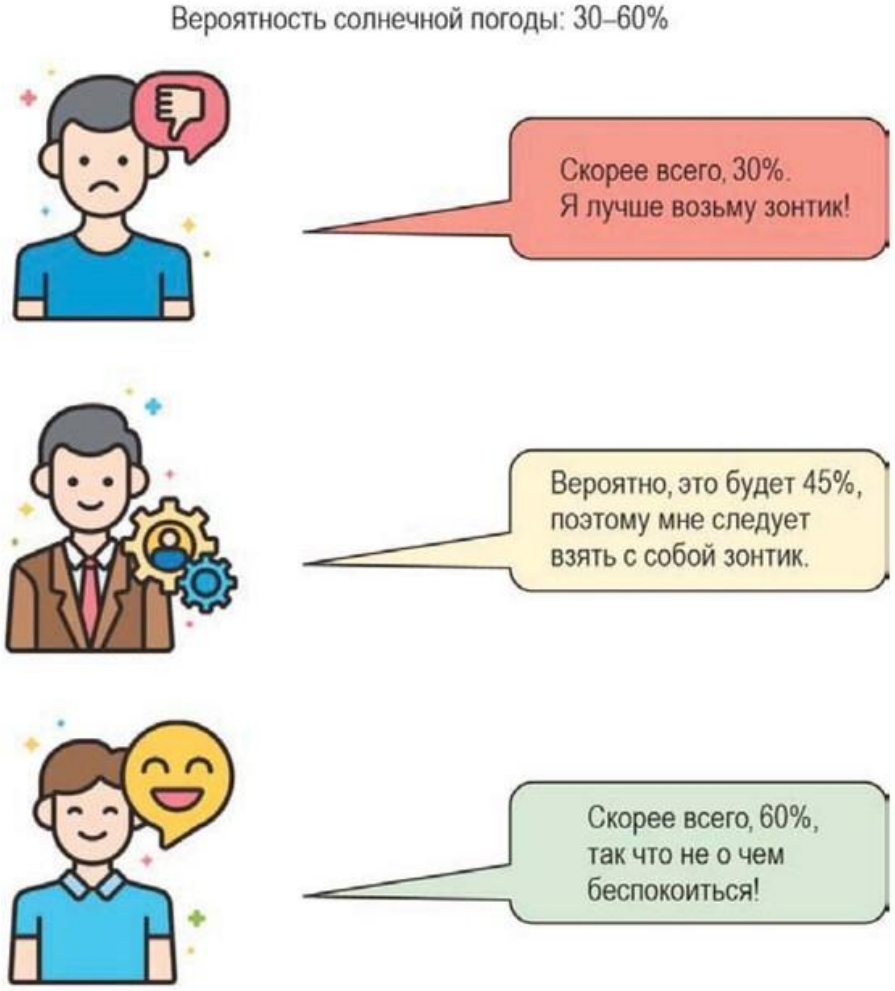


Рис. 5.5. Разные способы мышления при принятии решений: последний человек соответствует политике UCB, которая оптимистично оценивает неизвестную величину

Примечание

Способ реализации политики UCB в БО особенно прост. Используя ГП в качестве прогностической модели для целевой функции, у нас уже есть верхняя граница нормы вознаграждения за каждое действие. То есть у нас есть верхняя граница целевого значения в любом заданном месте ввода.

Мы знаем, что целевое значение в данном месте соответствует нормальному распределению, а стандартной мерой для количественной оценки неопределенности нормального распределения является 95%-ный ДИ, который содержит 95% вероят-

ностной массы распределения. С помощью ГП мы визуализируем этот 95%-ный ДИ во входном пространстве в виде толстой линии на рис. 5.6. Верхняя граница заштрихованной области — это именно то, что политика UCB использует в качестве оценок приобретения для точек данных в пространстве поиска. А в точке, которая дает наивысшую оценку, мы будем исследовать целевую функцию дальше. В данном случае это местоположение, обозначенное пунктирной линией около –1,3.

Определение

Оценки приобретения количественно определяют, насколько ценна точка данных, помогающая достичь оптимума целевой функции (раздел 4.1.1).

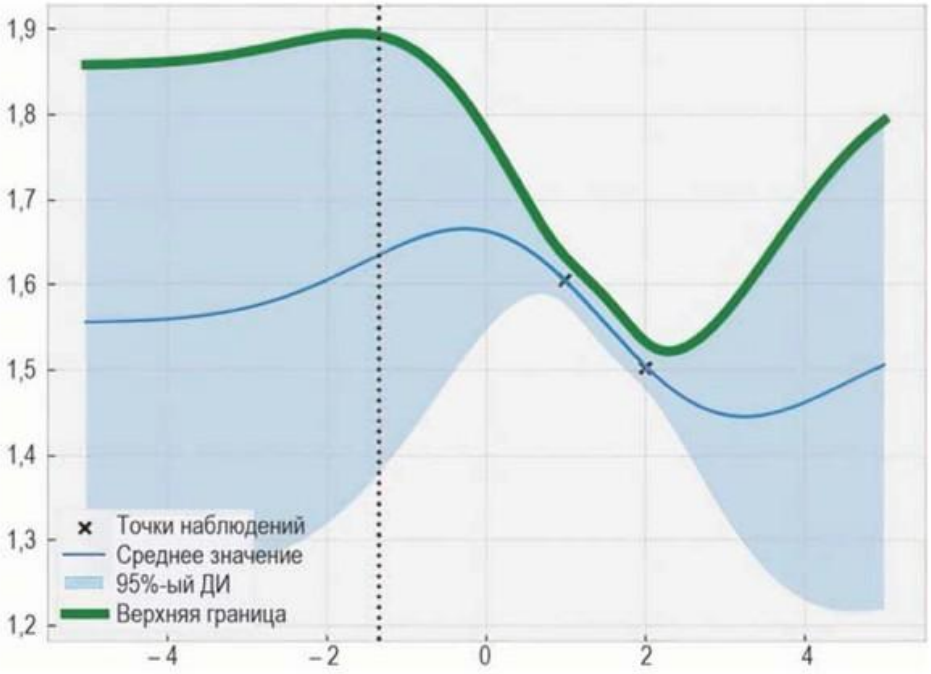


Рис. 5.6. Верхняя доверительная граница ГП, соответствующая 95%-ному ДИ, может быть использована в качестве оценки приобретения политики UCB

Можно подумать, что такая форма принятия решений может оказаться неприемлемой, особенно в ситуациях с большими ставками, когда цена неправильного решения высока. В примере с зонтиком, оптимистично недооценивая вероятность дождя, вы рискуете попасть под дождь.

Однако это особенно эффективный в вычислительном отношении способ рассуждения, поскольку все, что нам нужно, — это получить верхнюю границу оценки для интересующей нас величины и использовать ее при принятии решений. Более того, в следующем разделе мы увидим, как можно иметь полный контроль над оптимистичностью UCB, выбирая, какой ДИ мы будем использовать (отличный от 95%). Это также позволит политике сбалансировать исследование и эксплуатацию, что является основным вопросом, который должен быть решен в любой политике БО.

5.2.2. Баланс между исследованием и эксплуатацией

В этом разделе мы поговорим о том, как можно скорректировать поведение UCB, чтобы обеспечить баланс между регионами с высокой неопределенностью (исследование) и регионами с высоким прогностическим средним значением (эксплуатация). Это обсуждение направлено на более глубокое понимание политики, прежде чем мы перейдем к ее реализации с помощью BoTorch в следующем разделе.

Следует помнить, что при нормальном распределении отклонение от среднего значения на величину в два стандартных отклонения (т. е. $\mu \pm 2\sigma$) дает 95%-ный ДИ. Верхняя граница интервала ($\mu + 2\sigma$) — это оценка приобретения UCB, которую мы обсудили в предыдущем разделе.

Однако 95%-ный ДИ — это не единственный ДИ нормального распределения. Задавая множитель (β) для стандартного отклонения σ в формуле $\mu + \beta\sigma$, мы получаем другие ДИ. Например, для одномерного нормального распределения верно следующее (как показано на рис. 5.7):

- Увеличение на одно стандартное отклонение выше среднего значения ($\mu + \sigma$), т. е. установка $\beta = 1$, дает 68%-ный ДИ. Иначе говоря, 68% массы вероятности нормального распределения лежит между $\mu - \sigma$ и $\mu + \sigma$.
- Аналогичным образом три стандартных отклонения от среднего значения ($\beta = 3$) дают 99,7%-ный ДИ.

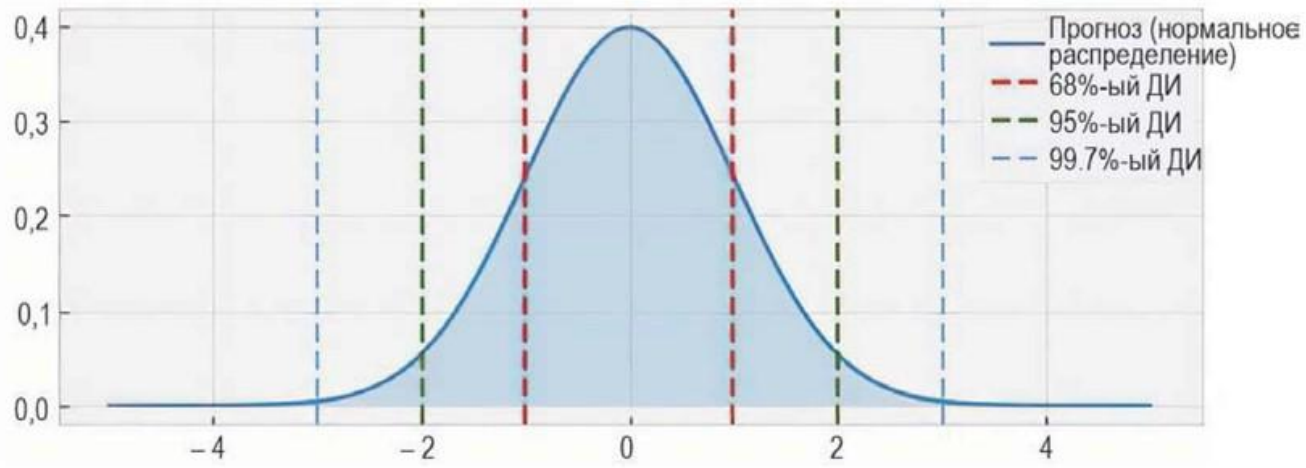


Рис. 5.7. Различные ДИ стандартных нормальных распределений: если отойти от среднего значения на одно, два и три стандартных отклонения, мы получим ДИ в 68, 95 и 99,7%; верхние границы этих интервалов используются политикой UCB

Фактически любое значение β дает уникальный ДИ для нормального распределения. Поскольку UCB предписывает использовать только верхнюю границу прогностической модели для принятия решений, им может быть любое значение $\mu + \beta\sigma$. Задавая параметр β, можно контролировать поведение политики.

На рис. 5.8 показаны три разные верхние границы (β = 1, 2, 3). Однако функция среднего, по сути, соответствует положению $\beta = 0$. Можно увидеть, что верхние границы имеют примерно одинаковую форму, но они идут вверх и вниз с разной скоростью. Кроме того, поскольку UCB для запроса оптимизации выбирает точку

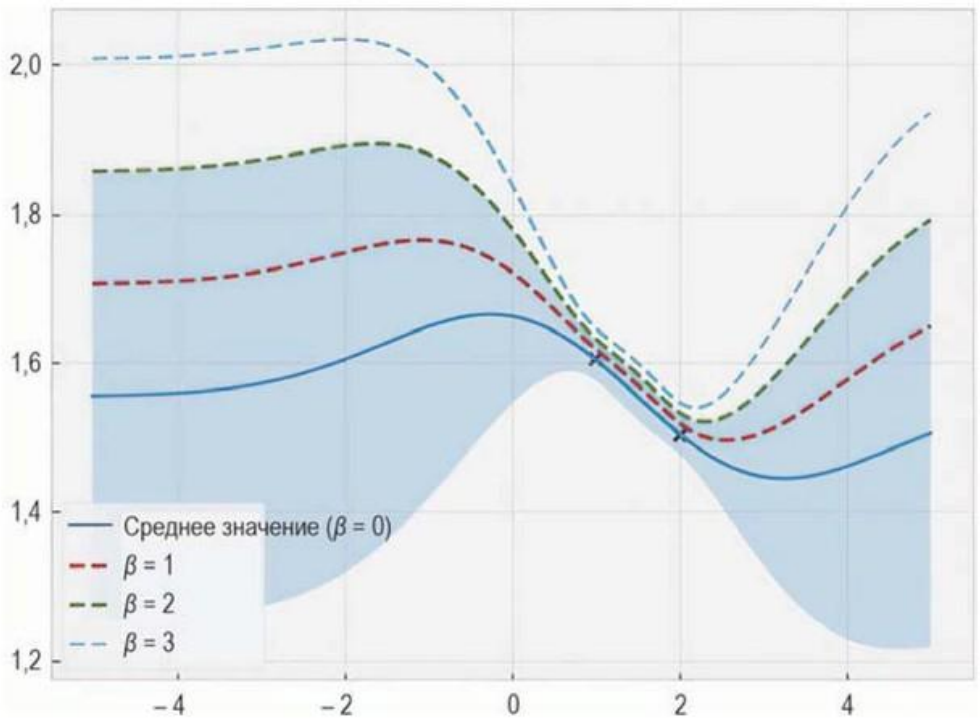


Рис. 5.8. Различные верхние границы ГП, соответствующие разным значениям ДИ и β: чем больше β, тем более исследовательской становится политика UCB

данных, максимизирующую границу, то разные границы (разные значения β) будут вызывать различное поведение оптимизации.

> **Важное примечание**
>
> Чем меньше β, тем более эксплуататорской становится UCB. И наоборот, чем больше β, тем UCB будет более исследовательской.

Мы можем наблюдать, что значение β контролирует поведение UCB, обратив внимание на формулу $\mu + \beta\sigma$. Когда β достаточно мало, μ вносит наибольший вклад в оценку приобретения. Таким образом, точка данных с самым высоким прогностическим средним значением максимизирует этот показатель. Такой выбор соответствует чистой эксплуатации, поскольку мы просто выбираем точку с наибольшим прогнозируемым значением. С другой стороны, когда β велико, стандартное отклонение σ, которое количественно определяет нашу неопределенность, приобретает большую важность в оценке UCB, подчеркивая необходимость исследований.

Это различие можно увидеть на рис. 5.8, где область около 0 — это место, где мы достигаем самого высокого прогностического среднего. Это указывает на то, что регион эксплуататорский. По мере увеличения β точка, в которой достигаются пики различных верхних границ, постепенно перемещается влево, где в прогнозе появляется больше неопределенности.

> **Примечание**
>
> Интересно, что в пределе, когда $\beta \to \infty$, точка, которая максимизирует показатель приобретения UCB, — это точка, где стандартное отклонение σ имеет максимальное значение (т. е. там, где неопределенность максимальна). Такое поведение соответствует чистому исследованию, поскольку выбираются точки данных с высокой неопределенностью.

Наконец, UCB правильно присваивает более высокие оценки точкам данных, которые обеспечивают лучший баланс между исследованием и эксплуатацией:

- Если две точки данных имеют одинаковое прогностическое среднее, но разные стандартные отклонения, то точка с более высокой неопределенностью получит оценку выше. Таким образом, политика поощряет исследования.
- Если две точки данных имеют одинаковые стандартные отклонения, но разные прогностические средние значения, то точка с более высоким средним значением будет иметь более высокую оценку. Таким образом, политика вознаграждает эксплуатацию.

Следует помнить, что политика EI также обладает этим свойством, которое является обязательным для любой политики БО.

В целом параметр β контролирует, как политика исследует и использует пространство поиска. Это означает, что, задавая значение параметра, мы имеем прямое влияние над ее поведением. К сожалению, помимо того, что значение β соответствует уровню исследования, не существует простого и принципиального способа установки этого параметра, и некоторые значения могут работать для одних задач, но не работать для других. Упражнение 1 в этой главе сфокусировано на более сложном методе установки β, который в целом может работать достаточно хорошо.

> **Примечание**
>
> В документации BoTorch политика по умолчанию имеет параметр β = 0,1, в то время как во многих исследовательских работах, в которых UCB используется для БО, β = 3. Таким образом, 0,1 может быть оптимальным значением при использовании этой политики, если есть предпочтение к эксплуатации. В свою очередь, значение 3 позволит склонить поведение политики к исследованию.

5.2.3. Реализация с помощью BoTorch

Теперь можно перейти к обсуждению, как реализовать эту политику с помощью BoTorch. Код, который здесь рассматривается, включен в `CH05/01 – BayesOptloop.ipynb`. Вспомним раздел 4.2.2. Несмотря на то что можно вручную реализовать политику PoI, объявление объекта BoTorch, использование его с моделью ГП и оптимизация оценки приобретения с помощью вспомогательной функции `BoTorchOptimize_acqf()` упрощают реализацию нашего цикла БО.

По этой причине мы сделаем здесь то же самое и используем встроенный класс политики UCB, хотя могли бы просто вычислить величину $\mu + \beta\sigma$ самостоятельно.

```
policy = botorch.acquisition.analytic.UpperConfidenceBound(
    model, beta=1
)
```

Здесь реализация класса UCB принимает модель ГП в качестве первого входного аргумента и положительное значение `beta` в качестве второго. Последнее соответствует параметру β в формуле $\mu + \beta\sigma$, которая обеспечивает компромисс между исследованием и эксплуатацией. Пока что `beta` присвоено значение 1.

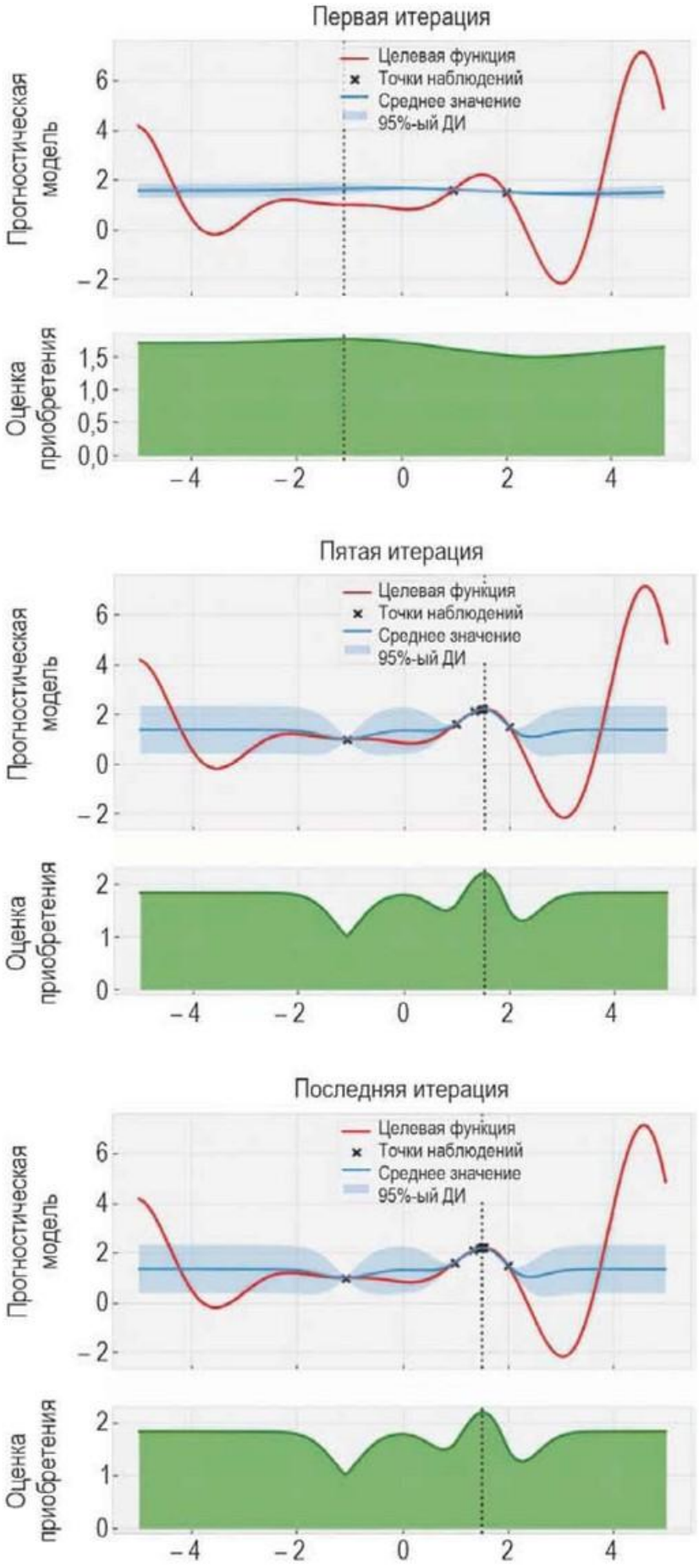


Рис. 5.9. Прогресс политики UCB с параметром $\beta = 1$: значение параметра недостаточно велико, чтобы стимулировать эксплуатацию, в результате чего прогресс застревает в локальном оптимуме

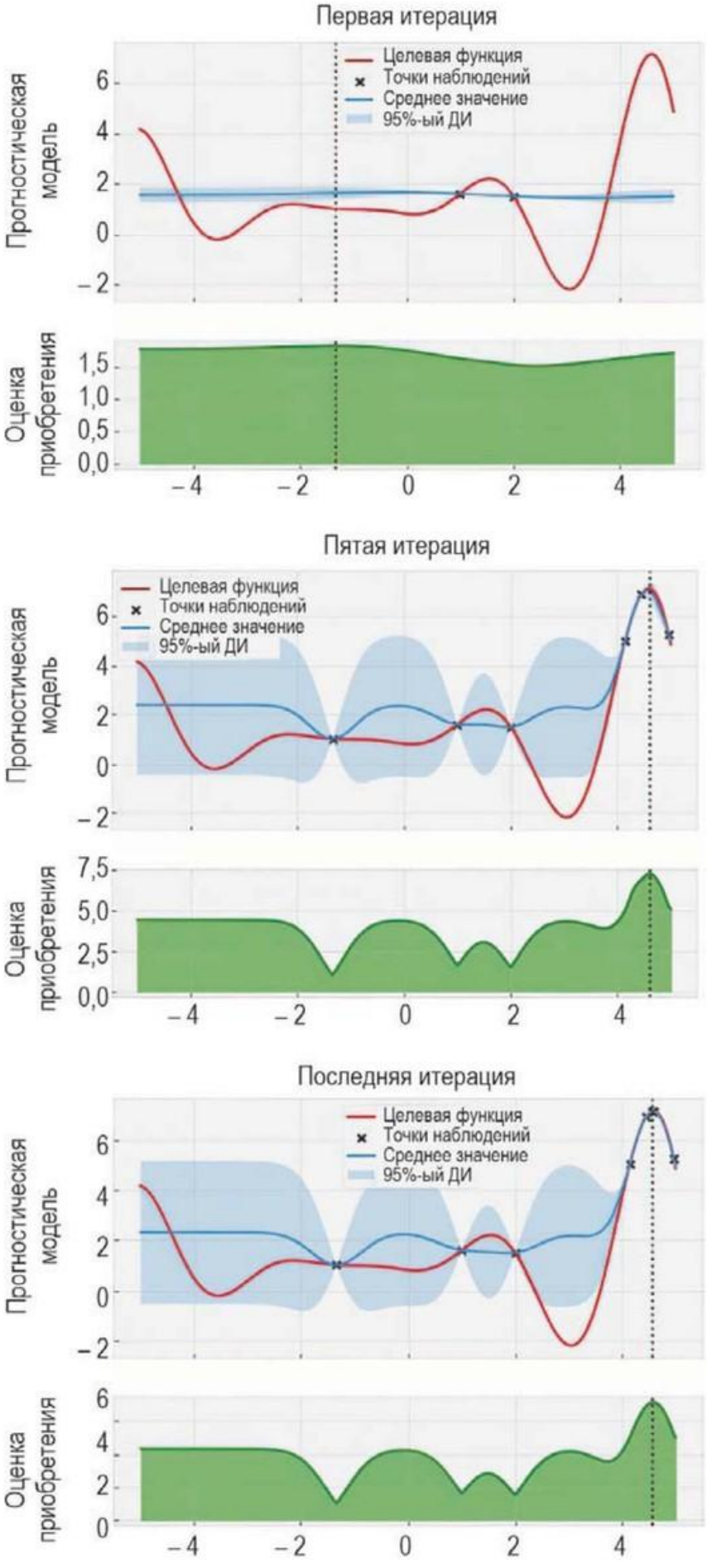


Рис. 5.10. Прогресс политики UCB с параметром $\beta = 2$: глобальный оптимум найден успешно

Примечание

Вы можете сомневаться, но это все, что нужно изменить в коде из предыдущей главы, чтобы применить политику UCB к имеющимся целевым функциям. Это демонстрирует преимущество модульности BoTorch. Она позволяет подключать и использовать любую политику в конвейере БО.

Запуск UCB с параметром $\beta = 1$ для нашей целевой функции Форрестера даст результат, изображенный на рис. 5.9. Он показывает прогресс оптимизации на протяжении 10 оценок функции. Можно убедиться, что, как и в случае с PoI, UCB не может в достаточной степени исследовать пространство поиска и застревает в локальном оптимуме. Это означает, что значение β слишком мало.

Сделаем еще одну попытку, на этот раз установив для β большее значение:

```
policy = botorch.acquisition.analytic.UpperConfidenceBound(
    model, beta=2
)
```

Реализация показана на рис. 5.10. Сейчас UCB смогла найти глобальный оптимум благодаря более тщательному исследованию, вызванному новым значением β. Однако если этот параметр будет велик настолько, что UCB потратит все ресурсы только на изучение пространства поиска, производительность оптимизации может пострадать.

Важность параметра β в политике UCB очевидна. Однако трудно сказать, какое значение будет хорошо работать для каждой конкретной целевой функции. В упражнении 1 к этой главе рассматривается определенная стратегия, которая позволяет корректировать значения этого параметра по мере продвижения поиска.

На этом можно завершить обсуждение политики UCB. Мы узнали, что, используя *оптимизм в условиях неопределенности* для задачи многорукого бандита, мы получим политику БО, исследовательское поведение которой можно контролировать и настраивать с помощью специального компромиссного параметра. Далее мы рассмотрим вторую политику, относящуюся к MAB, которая имеет совершенно другую мотивацию и стратегию.

5.3. Интеллектуальная выборка с помощью политики выборки Томпсона

В этом разделе мы узнаем о другой эвристике, которая называется *выборкой Томпсона* (**Thompson Sampling**, **TS**). Она использует совершенно другую мотивацию, нежели UCB, и, следовательно, вызывает другое поведение оптимизации. Как и в разделе 5.2, мы сначала изучим общую концепцию данной политики, а затем перейдем к ее реализации в коде.

5.3.1. Одна выборка для представления неизвестного

Используя UCB, мы принимаем решения на основе оптимистической оценки неизвестной величины, которая нас волнует. Это предполагает простой способ рассуж-

дения о последствиях действий, которые мы предпринимаем, и о вознаграждениях, которые мы получаем, в результате компромисса между исследованием и эксплуатацией. А что насчет TS?

Определение

Идея выборки Томпсона состоит в том, чтобы сначала сохранить вероятностное убеждение в отношении интересующей нас величины, а затем на его основе сделать выборку и рассмотреть ее как замену этой самой рассматриваемой величины. Такая выборочная замена затем используется для выбора лучшего решения.

Вернемся к нашему примеру с прогнозом погоды и посмотрим, как это работает. Нас интересует, следует ли нам брать с собой на работу зонтик. Нам дана оценка вероятности, что погода будет солнечной в течение дня. В данной ситуации UCB полагается на верхнюю границу данной оценки при принятии решения, но что будет делать политика TS?

Сначала она извлекает выборку из распределения вероятностей, которое используется для моделирования неизвестной величины (в данном случае вероятность солнечной погоды), и принимает решения на ее основе. Допустим, вместо диапазона, который мы наблюдали в предыдущем разделе, приложение теперь сообщает, что с вероятностью 66% (примерно две трети) погода будет солнечной. Это соответствует тому, что, следуя политике TS, мы будто подбрасываем монету с уклоном в сторону орла на две трети (из 3 подбрасываний орел выпадет 2 раза):

- Если монета упадет орлом вверх (вероятность 66%), мы рассматриваем это как вариант, что погода будет ясной в течение дня, и делаем вывод, что зонтик не понадобится.
- Если монета упадет решкой вверх (вероятность 34%), мы рассматриваем это как вариант, что пойдет дождь, и приходим к выводу, что следует взять с собой зонт.

Данный процесс визуализирован на рис. 5.11 в виде дерева решений, где вначале мы подбрасываем «смещенную» монету, чтобы получить выборку из распределения вероятностей солнечной погоды. В зависимости от результата, решаем, следует ли нам брать с собой на работу зонтик.

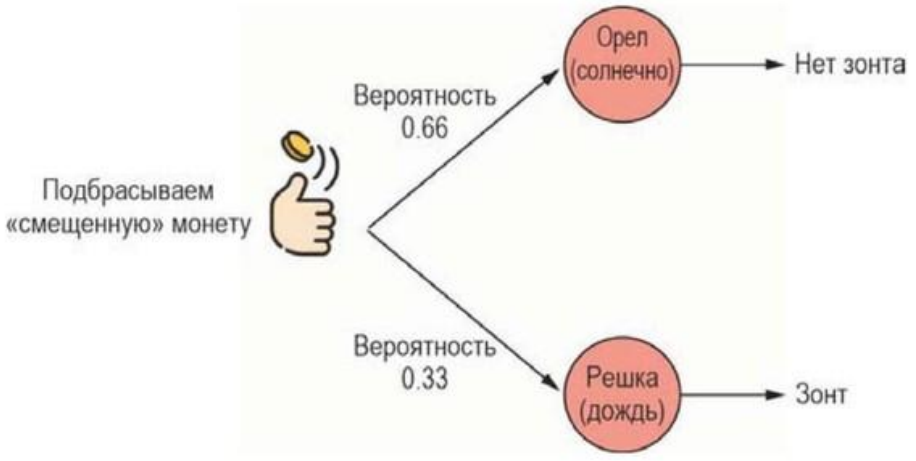


Рис. 5.11. Политика TS в виде дерева решений: мы подбрасываем монету, чтобы получить выборку распределения вероятностей солнечной погоды, и на ее основе решаем, брать ли с собой зонтик

На первый взгляд это может показаться произвольным методом принятия решений. TS особенно эффективна как в MAB, так и в БО. Прежде всего, выборка учитывает распределение вероятностей, представляющее интересующую величину, и является возможной ее реализацией. Таким образом, эту выборку можно использовать как представление распределения. При оптимизации в условиях неопределенности TS предлагает те же преимущества, что и UCB, поскольку зачастую выполнить выборку из распределения вероятностей достаточно легко. Точно так же, как оптимистичные оценки ставок вознаграждения UCB могут быть получены эффективным способом, выборка может быть сделана столь же эффективно.

Рассмотрим, как работает TS в БО. Для этого из ГП, обученного на основе текущих наблюдаемых данных, мы возьмем одну выборку. Как уже обсуждалось в главе 3, выборка из гауссовского процесса — это функция, представляющая конкретную реализацию целевого объекта в соответствии с предположениями нашего ГП. Однако, в отличие от истинной целевой функции, которая неизвестна в регионах, где нет наблюдаемых данных, выборка, полученная из ГП, полностью известна. Это означает, что мы можем найти место, в котором выборка максимальна.

На рис. 5.12 показан ГП, обученный на функции Форрестера, и три выборки, взятые из него (пунктирные линии). Далее вдоль каждой из выборок «ромбиком» указано место, в котором они максимизируются. Как можно наблюдать, одна выборка максимизируется возле значения −3,2, другая — возле −1,2, а третья — в точке 5.

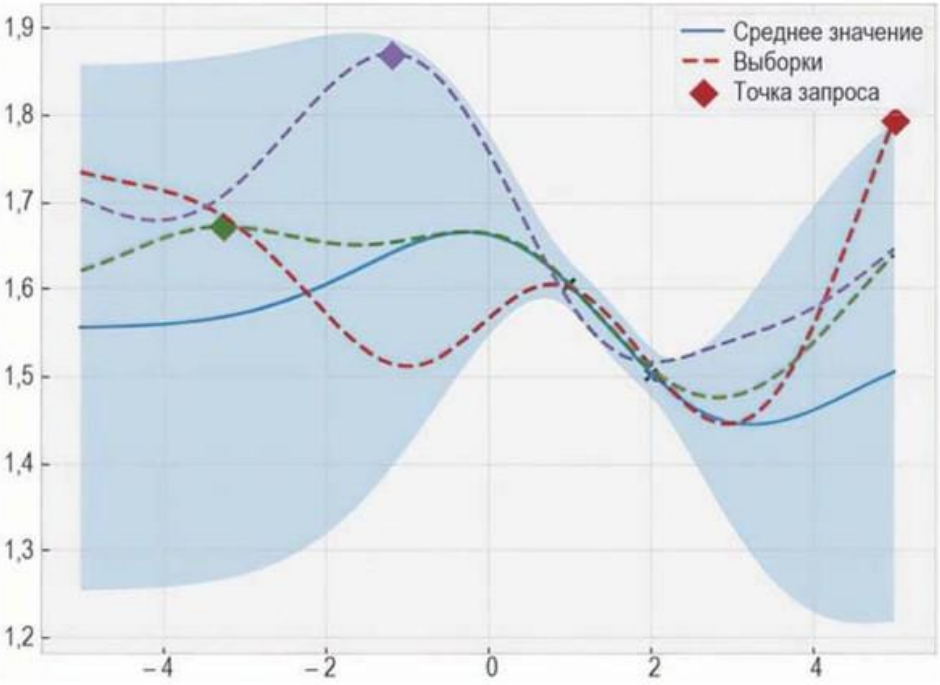


Рис. 5.12. Выборки из ГП и точки данных, максимизирующие их: какая бы выборка ни была взята, TS запросит точку, которая ее максимизирует, в качестве следующей для оценки целевой функции

При использовании политики TS определить, какую выборку мы извлекаем из ГП, можно лишь с помощью случайности. Однако какую бы из них мы ни выбрали, следующим запросом будет та точка данных, которая максимизирует выборку. То есть если нарисовать выборку с максимальным значением –3,2, то в следующий раз

целевая функция будет оцениваться в точке –3,2. А если нарисовать выборку с максимальным значением 5, то следующим запросом будет точка $x = 5$.

Определение

Оценки приобретения, которые вычисляет TS, представляют собой значения случайной выборки, взятой из ГП. Точка данных, которая максимизирует эту выборку, будет запрашиваться следующей.

В отличие от других политик, TS является «рандомизированной». Это означает, что при работе с одними и теми же данными обучения и ГП нет гарантии, что она примет одно и то же решение (если мы не установим случайное начальное число вычислительным путем). Однако такая случайность не является недостатком. Получить выборку из ГП легко, поэтому вычисление оценки приобретения в TS может быть выполнено эффективно. Кроме того, максимизация случайной выборки по своей сути балансирует между исследованием и эксплуатацией, что, как мы знаем, является основной задачей БО:

- Если точка данных имеет высокое прогностическое среднее, то значение случайной выборки в ней, скорее всего, будет высоким. Это повышает вероятность, что она максимизирует выборку.
- Если точка данных имеет высокое прогностическое стандартное отклонение (неопределенность), то случайная выборка в ней также будет иметь более высокую вариабельность и, следовательно, более высокую вероятность получить большое значение. Таким образом, это повышает вероятность выбора точки данных в качестве следующего запроса.

TS, скорее всего, будет эксплуатировать случайную выборку, которая максимальна в регионе с высоким прогностическим средним, но она с той же вероятностью будет эксплуатировать и другую выборку в такой же ситуации. Это видно на рис. 5.12, где одна выборка максимизируется около точки −1,2, которая имеет относительно высокое прогностическое среднее. Если мы возьмем эту выборку, то, оценивая целевую функцию в точке −1,2, политика уходит в сторону эксплуатации. Однако если мы возьмем любую из двух других выборок, политика уйдет в сторону исследования, поскольку регионы, где выборки максимальны, имеют высокую неопределенность.

Это довольно элегантная схема баланса. Используя случайность выборок, TS напрямую задействует вероятностную природу нашей прогнозирующей модели для исследования и эксплуатации пространства поиска. Рандомизированный характер TS означает, что на любой итерации цикла БО есть вариант, что политика не примет лучшее решение о компромиссе между исследованием и эксплуатацией. Но со временем, агрегируя свои решения, политика сможет исследовать достаточно пространства и сузить высокоэффективные регионы.

5.3.2. Реализация с помощью BoTorch

Перейдем к реализации политики TS в Python. Код включен в `CH05/01 – Bayes Optloop.ipynb`. Обратите внимание, что для политик БО реализация сводится к объяв-

лению объекта политики BoTorch и указанию соответствующей информации. А чтобы найти точку данных, которую нужно запросить следующей, нужно оптимизировать оценку приобретения, рассчитанную политикой. Однако в случае с TS нужно делать по-другому.

Основная проблема реализации TS в качестве универсального модуля PyTorch, который принимает точку данных и оценивает ее по некоторому критерию, заключается в том, что выборку из ГП можно выполнить только с конечным числом точек. Раньше мы представляли выборку из ГП, используя многомерное распределение (MVN) на плотной сетке по пространству поиска. Когда выборки из этой гауссианы отображены на графике, они кажутся реальными функциями с плавными кривыми, но на самом деле они определяются на сетке.

Все это означает, что с вычислительной точки зрения невозможно получить выборку как функцию от ГП, поскольку для этого потребуется бесконечное количество битов. Типичное решение — нарисовать соответствующее MVN-распределение по большому количеству точек, охватывающих входное пространство, чтобы в пространстве поиска были представлены все регионы.

Важное примечание

Это именно то, что нужно для реализации TS: мы генерируем большое количество точек по всему пространству поиска и извлекаем выборку из распределения MVN, соответствующую предсказаниям ГП по этим точкам.

Процедура реализации для TS обобщена на рис. 5.13, где используется *последовательность Соболя* (**Sobol sequence**) в качестве точек, охватывающих пространство поиска. Позже мы обсудим, почему она предпочтительнее других стратегий выборки. Затем мы берем выборку из ГП в этих точках и выбираем ту, которая дает наибольшее значение. Далее мы оцениваем целевую функцию в том месте, где в следующий раз значение выборки максимизируется.

Определение

Последовательность Соболя — это бесконечный список точек в области евклидова пространства, целью которого является равномерное покрытие этой области.

Сначала обсудим, почему нам нужно использовать последовательность Соболя для создания точек, охватывающих пространство поиска. Более простым решением было бы использовать плотную сетку. Однако ее создание быстро становится невозможным по мере роста числа измерений пространства, поэтому такая стратегия бесполезна. Другим потенциальным решением является равномерная выборка из этого пространства, но статистическая теория показывает, что такой подход на самом деле не лучший способ генерировать точки, равномерно покрывающие пространство, и последовательность Соболя справляется с этой задачей лучше.

На рис. 5.14 показано сравнение последовательности Соболя из 100 точек и того же количества точек, равномерно отобранных в пределах двумерного единичного квадрата. Видно, что последовательность Соболя покрывает квадрат более равномерно, чего мы и хотели добиться с помощью TS. Данный контраст еще заметнее

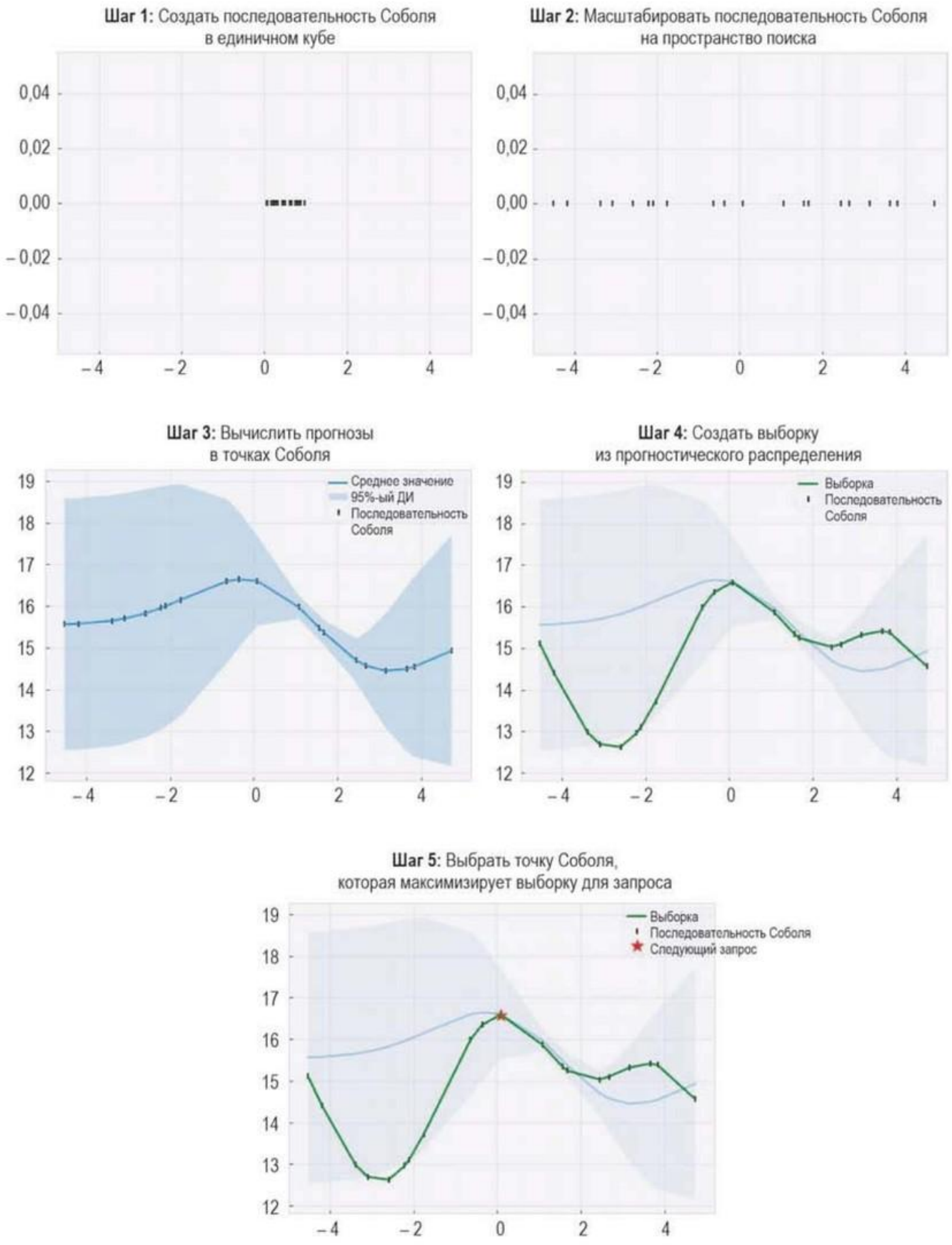


Рис. 5.13. Блок-схема реализации TS в BoTorch: мы используем последовательность Соболя для заполнения пространства поиска, извлекаем выборку из ГП и выбираем точку в последовательности, которая максимизирует выборку для оценки целевой функции

в более высоких измерениях, что дает больше оснований предпочесть именно эту последовательность, чем равномерно выбранные точки. Нет необходимости рассматривать последовательность Соболя подробнее. Достаточно знать, что это наиболее подходящий метод генерации выборки, если наша цель — равномерно покрыть пространство.

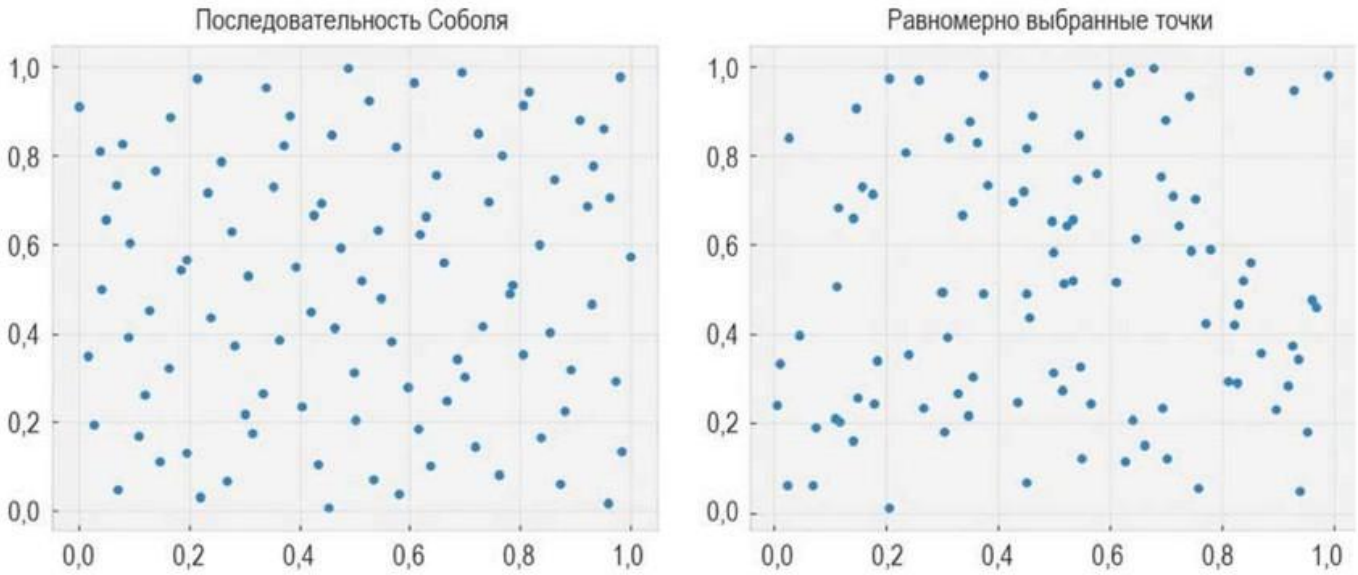


Рис. 5.14. Точки из последовательности Соболя и равномерно выбранные точки в двумерном единичном квадрате: последовательность Соболя покрывает квадрат более равномерно

PyTorch предоставляет реализацию последовательностей Соболя, которую можно использовать следующим образом:

```
dim = 1 #Количество измерений пространства
num_candidates = 1000 #Количество точек, которое будет сгенерировано

sobol = torch.quasirandom.SobolEngine(dim, scramble=True)
candidate_x = sobol.draw(num_candidates)
```

Здесь `sobol` является экземпляром класса `SobolEngine`, реализующим логику выборки последовательностей Соболя в единичном кубе, а `candidate_x` — это тензор PyTorch (`num_candidates`, `dim`), содержащий сгенерированные точки с правильной размерностью.

> **Примечание**
>
> Важно помнить, что `SobolEngine` генерирует точки, покрывающие единичный куб. Чтобы `candidate_x` покрывал нужное пространство, следует соответствующим образом изменить размер этого тензора.

Сколько точек должна содержать последовательность Соболя (`num_candidates`), решать нам (пользователям). Предыдущий пример показывает, что их задействовано 1000. В данном случае понадобится достаточно большое значение, чтобы можно было в полной мере охватить пространство поиска. Однако слишком большое значение сделает выборку из апостериорного ГП численно нестабильной.

Численная нестабильность при построении выборок ГП

Численная нестабильность при построении выборок ГП иногда может привести к следующему предупреждению при запуске TS:

```
NumericalWarning: A not p.d., added jitter of 1.0e-06 to the diagonal
 warnings.warn()
```

Это предупреждение указывает, что в коде возникла числовая проблема из-за того, что ковариационная матрица ГП не является положительно определенной (`p.d.`), когда она должна быть таковой. Однако этот код также применяет автоматическое исправление, когда мы добавляем «колебание» величины `1e-6` к ковариационной матрице, делая ее положительно определенной, поэтому нам (пользователям) не нужно ничего делать дальше.

Как и в разделе 4.2.3, мы используем модуль `warnings`, чтобы отключить это предупреждение и сделать вывод более чистым следующим образом:

```
with warnings.catch_warnings():
    warnings.filterwarnings('ignore', category=RuntimeWarning)
    ... #код для TS
```

Можно попробовать поиграться с несколькими тысячами точек, чтобы найти число, которое лучше всего подходит для конкретного варианта использования, целевой функции и обученного ГП. Однако всегда следует использовать не менее 1000 точек.

Далее перейдем ко второму компоненту TS, который представляет собой процесс выборки MVN из апостериорного ГП и ее дальнейшей максимизации. Во-первых, объект, реализующий выборку, может быть объявлен так:

```
ts = botorch.generation.MaxPosteriorSampling(model, replacement=False)
```

Класс `MaxPosteriorSampling` реализует логику TS: выборку из апостериорного ГП и ее дальнейшую максимизацию. Здесь модель ссылается на ГП, обученный на основе наблюдаемых данных. Важно установить `replacement` в значение `False`, убедившись, что мы выполняем выборку без замены (выборка с заменой не подходит для TS). Наконец, чтобы получить точку данных, которая дает наибольшее выборочное значение из `candidate_x`, мы передаем ее объекту, выполняющему выборку:

```
next_x = ts(candidate_x, num_samples=1)
```

Возвращаемое значение действительно является точкой, которая максимизирует выборку и будет запрошена следующей. На этом можно завершить реализацию политики TS и подключить ее к коду, который мы использовали в цикле БО для функции Форрестера, следующим образом:

```
for i in range(num_queries):
    print("iteration", i)
    print("incumbent", train_x[train_y.argmax()], train_y.max())

    #Генерируем точки с помощью движка Соболя
    sobol = torch.quasirandom.SobolEngine(1, scramble=True)
    candidate_x = sobol.draw(num_candidates)
```

```
#Изменяем размер сгенерированных точек в диапазоне -5 и 5, что является
#нашим пространством поиска
candidate_x = 10 * candidate_x - 5

model, likelihood = fit_gp_model(train_x, train_y)

#Генерируем кандидата TS, который будет запрошен следующим
ts = botorch.generation.MaxPosteriorSampling(model, replacement=False)
next_x = ts(candidate_x, num_samples=1)

#Визуализируем текущий прогресс без функции оценки приобретения
visualize_gp_belief_and_policy(model, likelihood, next_x=next_x)

next_y = forrester_1d(next_x)

train_x = torch.cat([train_x, next_x])
train_y = torch.cat([train_y, next_y])
```

Обратите внимание, что общая структура цикла БО остается прежней. Отличие заключается в том, что вместо объекта политики BoTorch теперь есть последовательность Соболя для генерации набора точек, охватывающих пространство поиска, которые затем передаются в объект `MaxPosteriorSampling`, реализующий политику TS. Переменная `next_x`, как и раньше, содержит точку данных, которая будет запрашиваться следующей.

> **Примечание**
>
> Поскольку у нас нет объекта при визуализации прогресса с помощью вспомогательной функции `visualize_gp_belief_and_policy()`, мы больше не указываем аргумент политики. Таким образом, эта функция будет отображать только обученный ГП на каждой итерации без оценок приобретения.

Ход оптимизации TS показан на рис. 5.15, где можно наблюдать, что политика успешно достигает глобального оптимума на протяжении всей процедуры (не без затрат на исследование пространства). Это демонстрирует возможность сочетать исследование и эксплуатацию TS в байесовской оптимизации.

На этом обсуждение политик БО в рамках MAB можно завершить. Мы увидели, что каждая из двух изученных политик (UCB и TS) использует естественные эвристики для эффективного рассуждения о неизвестных величинах и принятия соответствующих решений. Одна из задач БО (баланс между исследованием и эксплуатацией) также может быть успешно решена обеими политиками, что позволяет им иметь хорошие показатели оптимизации. В следующей и последней главе второй части книги мы узнаем о еще одной часто используемой эвристике принятия решений в БО, на этот раз с использованием теории информации.

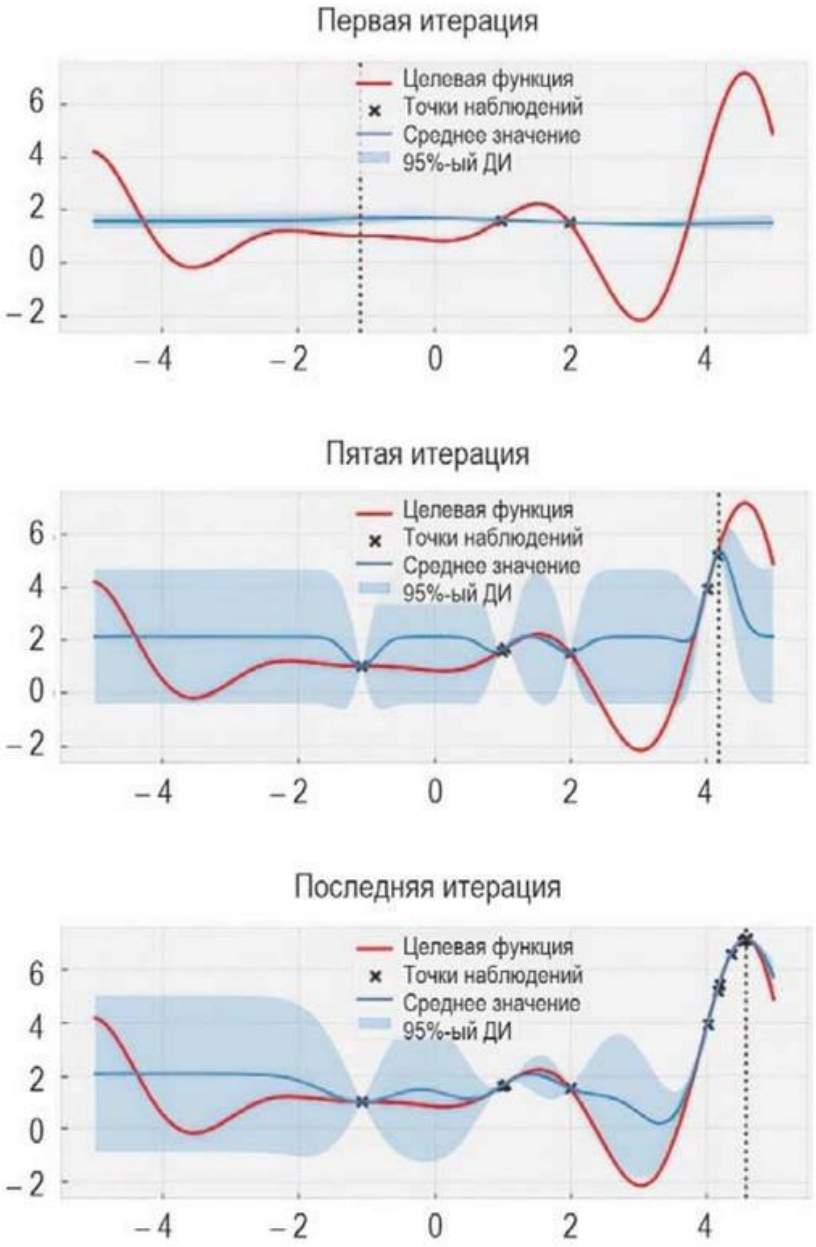


Рис. 5.15. Прогресс политики TS, которая исследует пространство поиска в течение нескольких итераций, а затем постепенно приближается к глобальному оптимуму

5.4. Упражнения

В этой главе есть два упражнения:

1. В первом упражнении исследуется потенциальный метод установки компромиссного параметра для политики UCB, который учитывает, насколько далеко мы продвинулись в оптимизации.

2. Во втором упражнении две политики, которые были изучены в этой главе, применяются к задаче настройки гиперпараметров, рассмотренной в предыдущих главах.

5.4.1. Упражнение 1: Настройка плана эксплуатации для UCB

В этом упражнении (`CH05/02 - Exercise 1.ipynb`) обсуждается стратегия адаптивной установки значения для компромиссного параметра β. Как упоминалось ранее, производительность политики сильно зависит от этого параметра, но не очевидно, как нам следует устанавливать его значение. С некоторыми целевыми функциями он работает хорошо, а с другими плохо.

Специалисты байесовской оптимизации замечают, что по мере сбора все больших данных UCB может стать слишком эксплуататорской. Это связано с тем, что по мере увеличения размера обучающих данных мы получаем больше знаний о целевой функции, а неопределенность в отношении прогнозов ГП уменьшается. Это означает, что ДИ, создаваемый ГП, будет более точным при перемещении верхней границы (которая составляет оценку приобретения, используемую UCB) ближе к среднему прогнозу.

Однако если оценка приобретения UCB совпадает со средним прогнозом, то политика является эксплуататорской, поскольку она будет запрашивать только точки данных с высокими прогностическими средними значениями. Этот феномен показывает, что чем больше данных наблюдается, тем более исследовательской нам надо сделать политику. Здесь естественным способом, с помощью которого мы постепенно поощряем дополнительные исследования со стороны UCB, является медленное увеличение значения компромиссного параметра β, что мы и научимся делать в этом упражнении, выполнив следующие шаги:

1. Воссоздайте цикл БО из `CH04/02 - Exercise 1.ipynb`, который использует одномерную функцию Форрестера в качестве цели для оптимизации.
2. Медленно увеличьте значение компромиссного параметра β, умножая его на константу на каждой итерации цикла. Другими словами, в конце каждой итерации нужно обновить параметр с помощью `beta *= multiplier`.

   Например, мы хотим, чтобы β начинался со значения 1 и к концу поиска имел значение 10 (последняя десятая итерация). Какое значение должен иметь множитель для β?
3. Реализуйте эту логику планирования и понаблюдайте за полученной производительностью оптимизации.

   В частности, даже несмотря на то, что эта версия UCB начинается с $\beta = 1$, застревает ли она на локальном оптимуме, как, например, версия, для которой параметр фиксирован на 1?

5.4.2. Упражнение 2: БО для настройки гиперпараметров

В этом упражнении (`CH05/03 - Exercise 2.ipynb`) БО применяется к целевой функции, которая имитирует поверхность точности для модели машины опорных векторов в задаче по настройке гиперпараметров. Ось X обозначает штрафной параметр C, а ось Y — параметр γ ядра RBF. Более подробную информацию можно посмотреть в упражнениях к главам 3 и 4.

Шаги упражнения:

1. Воссоздайте цикл БО из `CH05/03 - Exercise 2.ipynb`, включая внешний цикл, реализующий повторяющиеся эксперименты.
2. Запустите политику UCB, установив значение компромиссного параметра равным $\beta \in \{ 1, 3, 10, 30 \}$, и понаблюдайте за совокупной производительностью значений:

   Какая ценность ведет к чрезмерной эксплуатации, а какая — к чрезмерному исследованию? Какое значение работает лучше всего?
3. Запустите адаптивную версию UCB (см. упражнение 1):
   - Параметр β должен начинаться с 3 и заканчиваться 10.
   - Обратите внимание, что изменение конечного значения с 10 на 30 не слишком сильно влияет на производительность оптимизации. Поэтому такая стратегия устойчива к значению конечной величины, которая является желательной.
   - Сравните производительность этой адаптивной версии и других версий с фиксированным β.
4. Запустите политику TS и понаблюдайте за ее совокупной производительностью.

Заключение

- Задача MAB состоит из набора действий, которые можно выполнить (рычаги игровых автоматов, которые можно потянуть). Каждая машина возвращает вознаграждение в соответствии со своей ставкой. Цель состоит в том, чтобы максимизировать сумму вознаграждения (в совокупности), которое мы получим за определенное количество итераций.
- Политика MAB выбирает, какое действие предпринять дальше на основе предыдущих данных. Хорошая политика должна балансировать между малоизученными и высокоэффективными действиями.
- В отличие от MAB, где имеется конечное множество итераций, в БО можно выполнять бесконечное количество действий.
- Цель БО — максимизировать наибольшее наблюдаемое вознаграждение, которое часто называют *простым вознаграждением*.
- Награды в БО коррелируют: аналогичные действия приносят одинаковые вознаграждения. Это не обязательно верно для MAB.
- Политика UCB использует оптимистичную оценку для принятия решений. Такой оптимизм в условиях неопределенности может сбалансировать исследование и эксплуатацию с помощью компромиссного параметра, который можно настраивать.
- Чем меньше компромиссный параметр, тем более эксплуататорской становится политика, стремясь оставаться в пределах регионов, дающих высокие вознагра-

ждения. Чем больше компромиссный параметр, тем более исследовательской становится политика, стремясь запрашивать регионы, находящиеся далеко от наблюдаемых данных.

- Политика TS извлекает выборку из вероятностной модели и использует ее для принятия решений.
- Рандомизированный характер TS позволяет ей соответствующим образом исследовать и использовать пространство поиска. Политика, скорее всего, будет выбирать как области с высокой неопределенностью, так и области с высоким прогностическим средним значением.
- По вычислительным причинам при реализации TS требуется больше внимания. Сначала нужно сгенерировать набор точек для равномерного покрытия пространства поиска, а затем для них построить выборку из апостериорного ГП.
- Чтобы равномерно покрыть пространство поиска, можно использовать последовательность Соболя для генерации точек внутри единичного куба и масштабирования их до целевого пространства.

6 Использование теории информации с политиками на основе энтропии

Темы главы:

- Энтропия как теоретико-информационная мера неопределенности.
- Информация как метод снижения энтропии.
- Политики БО, использующие теорию информации для поиска.

В этой главе речь пойдет еще об одной эвристике принятия решений, использующей теорию информации для разработки политик БО, которые можно использовать для оптимизации. В отличие от тех, что мы уже видели (поиск улучшения, оптимизм в условиях неопределенности и случайная выборка), теория информации является основной областью математики, которая используется в большом количестве приложений на самые разные тематики. Обращаясь к данному методу, а именно к энтропии (величине, измеряющей неопределенность с точки зрения информации), можно разработать политику БО, стремящуюся уменьшить неопределенность в отношении целевой функции, которая должна быть оптимизирована принципиальным и математически элегантным способом.

Идея поиска, основанного на энтропии, довольно проста: мы наблюдаем места, где информация об интересующей величине возрастает больше всего. В этой главе мы рассмотрим, что метод похож на поиск потерянного пульта дистанционного управления в квартире.

Первая часть главы представляет собой общее изложение теории информации, энтропии и способов максимизировать количество информации, которую мы получаем при выполнении *действия*. Для этого мы переосмыслим идею двоичного поиска. Вооружившись основами теории, далее перейдем к обсуждению политик БО, которые максимизируют информацию о глобальном оптимуме целевой функции. Они являются результатом применения теории информации к задаче БО. Также мы узнаем, как реализовать эти политики в Python.

К концу главы вы получите практическое понимание, что такое теория информации, как количественно определяется энтропия в качестве меры неопределенности

и как она может быть переведена в БО. Эта глава добавит еще одну политику в наш набор инструментов, и на этом можно будет завершить вторую часть книги о политиках БО.

6.1. Измерение знаний с помощью теории информации

Теория информации (**Information theory**) — это раздел математики, который изучает, как лучше всего представлять, количественно оценивать и рассуждать об *информации* принципиальным математическим способом. В этом разделе мы познакомимся с ней на высоком уровне и обсудим, как она связана с принятием решений в условиях неопределенности. Для этого мы пересмотрим идею двоичного поиска (популярного алгоритма в информатике) с точки зрения теории информации. Это обсуждение впоследствии позволит связать данную эвристику с БО, а также покажет, что мотивирует теоретико-информационную политику оптимизации.

6.1.1. Измерение неопределенности с помощью энтропии

Теория информации особенно распространена в информатике, где цифровые данные представлены в виде битов (0 и 1). Возможно, вы помните задачи, когда надо вычислить, сколько битов требуется, чтобы представить заданное целое число. На рис. 6.1 показано, что одного бита достаточно для представления двух чисел (0 и 1), а для представления 32 чисел (2 в пятой степени) необходимо уже 5 битов. Такие расчеты являются примерами применения теории информации на практике.



Рис. 6.1. Пример применения теории информации на практике: определение количества битов для представления определенного количества целых чисел

Энтропия является теоретико-информационной концепцией в контексте принятия решений в условиях неопределенности. Она измеряет уровень неопределенности, которую мы имеем в неизвестной величине. Если эта неизвестная величина моделируется как случайная, то энтропия измеряет изменчивость возможных значений случайной величины.

Примечание

Эта мера неопределенности (энтропия) похожа, но не совсем совпадает с тем, что мы до сих пор называли неопределенностью в прогнозах ГП (просто стандартное отклонение прогнозного распределения).

В этом разделе мы узнаем больше об энтропии как концепции, а также о том, как она вычисляется для простого распределения Бернулли в бинарных событиях. Узнаем, как энтропия успешно определяет неопределенность неизвестной величины.

Рассмотрим классический пример теории вероятностей — подбрасывание монеты (рис. 6.2). Допустим, вы собираетесь подбросить «смещенную» монету, которая выпадет орлом с некоторой вероятностью p от 0 до 1, и вы хотите порассуждать о том, что она действительно упадет орлом вверх. Обозначим X как двоичную случайную величину, которая указывает, произойдет ли событие ($X = 1$, если монета упадет орлом, и $X = 0$ в противном случае). Тогда говорят, что X соответствует распределению Бернулли с параметром p, и вероятность того, что $X = 1$, равна p.

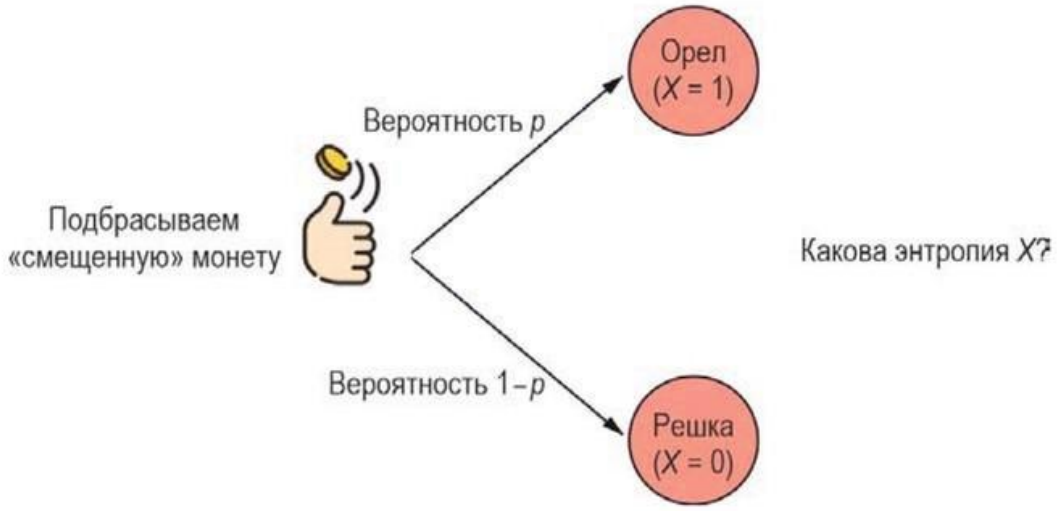


Рис. 6.2. Еще один пример применения теории информации на практике: определение вероятности выпадения «орла» или «решки» при подбрасывании монеты

Здесь энтропия X определяется как

$$-p \log p - (1 - p) \log(1 - p),$$

где log — логарифмическая функция по основанию 2. Можно сразу заметить, что это уравнение является функцией вероятности выпадения орла (p). На рис. 6.3 показана функция энтропии для p в точке (0, 1), из которой можно понять следующее:

- Энтропия всегда неотрицательна.
- Энтропия увеличивается, когда p увеличивается до значения 0,5, затем достигает максимума при $p = 0{,}5$, а после уменьшается.

Оба вывода ценны, когда дело касается изучения неуверенности, упадет ли монета орлом. Во-первых, не должно быть отрицательной неопределенности в отношении чего-либо, поэтому логично, что энтропия никогда не бывает отрицательной. Что еще более важно, энтропия максимизируется ровно посередине, когда $p = 0{,}5$. Это

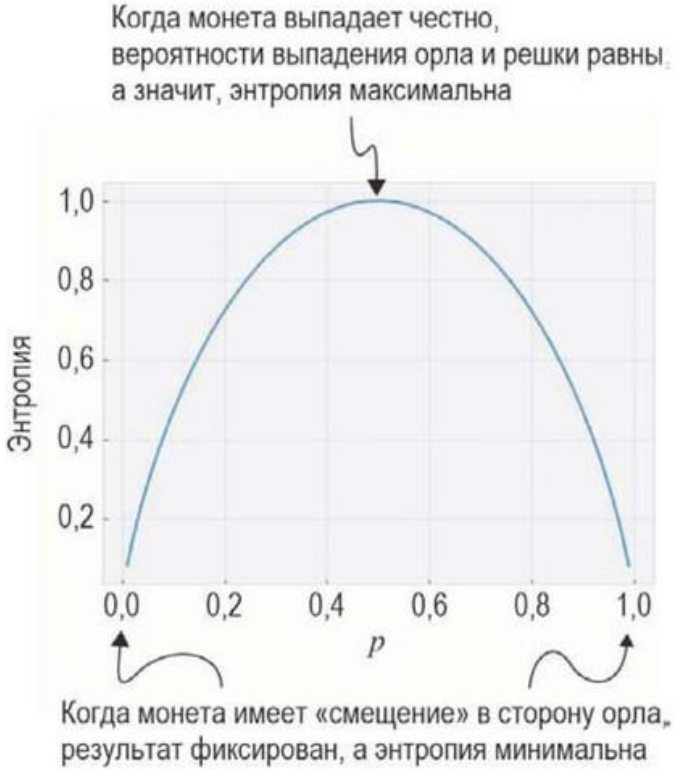


Рис. 6.3. Энтропия случайной величины Бернулли как функция вероятности успеха: энтропия максимальна (неопределённость максимальна), когда вероятность успеха равна 0,5

вполне разумно: по мере того как p становится все дальше и дальше от 0,5, мы становимся более уверенными в исходе события — упадет ли монета орлом.

Например, если $p = 0{,}7$, мы более уверены, что выпадет орел, а энтропия здесь имеет значение около 0,9. Если $p = 0{,}1$, то мы еще больше уверены в результате (на этот раз в том, что выпадет решка), а энтропия здесь примерно равна 0,5. Несмотря на то что энтропия не определена в конечных точках (из-за логарифмической функции), она стремится к нулю по мере приближения к любой из них, что указывает на нулевую неопределенность. С другой стороны, когда $p = 0{,}5$, неопределенность максимальна, поскольку мы максимально не уверены в том, какой стороной упадет монета. Эти расчеты показывают, что энтропия является подходящей мерой для оценки неопределенности.

Определение

Для данного распределения вероятностей энтропия определяется как $-\Sigma_i\, p_i \log p_i$, где мы суммируем различные возможные события, индексированные с помощью i.

Энтропия против стандартного отклонения

Когда мы использовали термин «неопределенность» в предыдущих главах, мы имели в виду стандартные отклонения прогнозируемых нормальных распределений, полученных с помощью ГП. Стандартное отклонение распределения, как следует из названия, измеряет, насколько значения в распределении отклоняются от среднего значения, и, следовательно, является действительной мерой неопределенности.

Энтропия, с другой стороны, мотивируется концепциями теории информации и также является действенной мерой неопределенности. Фактически это более элегантный и общий подход к количественной оценке неопределенности, который позволяет более точно моделировать ее в крайних случаях для многих ситуаций.

То, что мы использовали для предыдущего распределения Бернулли, является частным случаем нашей формулы. Мы также задействуем ее позже в этой главе, когда будем работать с равномерными распределениями.

6.1.2. Поиск пульта дистанционного управления с помощью энтропии

Поскольку энтропия измеряет степень неопределенности об интересующих нас величине или событии, она может влиять на решения, помогая наиболее эффективно уменьшить эту самую неопределенность. Рассмотрим на примере, где лучше искать потерянный пульт. Несмотря на простоту, задача представляет собой теоретико-информационные размышления, которые будут использоваться в последующих обсуждениях, где энтропия используется для более сложных задач принятия решений.

Представьте, что однажды, пытаясь включить телевизор в гостиной, вы понимаете, что не можете найти пульт дистанционного управления на столе, где он обычно находится. Таким образом, вы решаете его поискать. Во-первых, есть предположение, что он должен быть где-то в гостиной, но, где именно, неизвестно, поэтому все места одинаково вероятны. На языке вероятностного вывода можно сказать, что пульт равномерно распределен по всей гостиной.

Рис. 6.4 визуализирует это представление (место обозначено темной областью, согласно убеждению). Теперь нужно задать себе вопрос: где в этом доме искать пульт? Разумно предположить, что следует смотреть в гостиной, а не в ванной комнате, поскольку именно там находится телевизор. Но как можно количественно оправдать этот выбор?

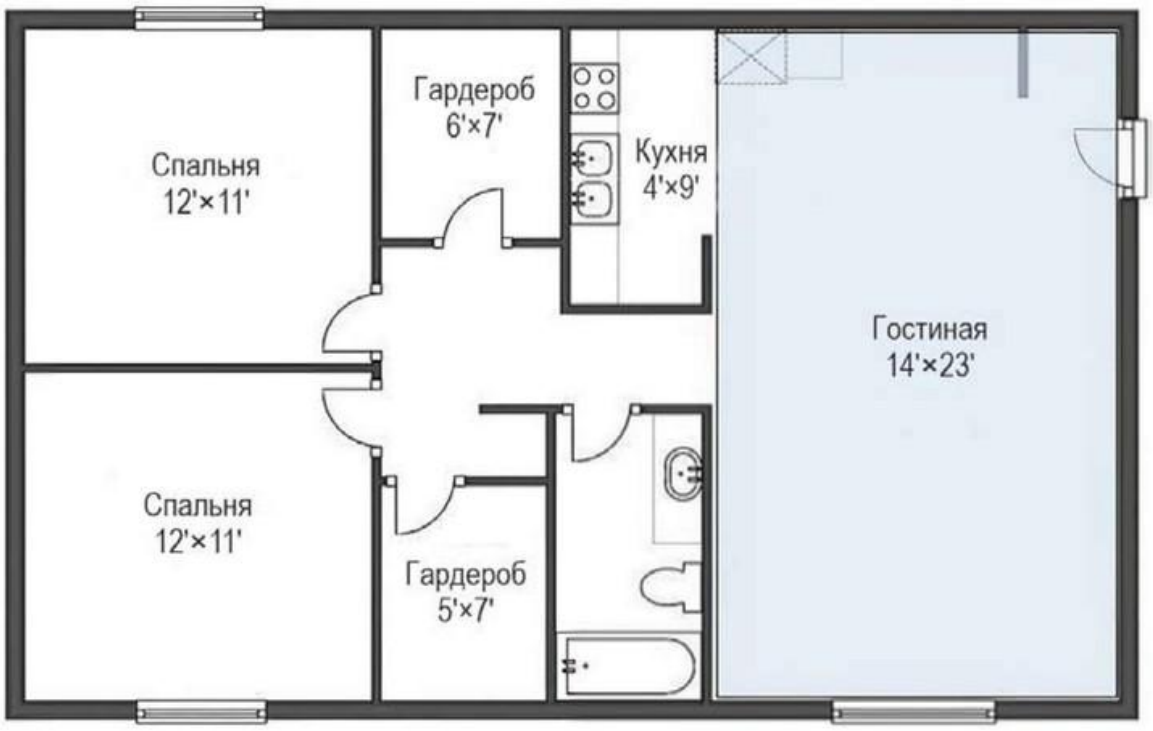


Рис. 6.4. План помещения в примере с поиском пульта ДУ: гостиная заштрихована равномерно, что указывает на равномерное распределение возможного нахождения пульта по всей площади

Теория информации (в частности, энтропия) предлагает способ сделать это с помощью рассуждений о том, сколько неопределенности остается после поиска пульта в гостиной или в ванной. Иначе говоря, у нас есть возможность определить, сколько неопределенности относительно местоположения пульта осталось после того, как мы посмотрели именно в гостиной, а не в ванной.

На рис. 6.5 показано, как энтропия местоположения пульта уменьшается при обыске верхней части гостиной. Строить рассуждения можно следующим образом:

- Если пульт найден в пределах искомого региона, то у нас больше не будет неуверенности в его местонахождении. Другими словами, энтропия будет равна нулю.
- Если пульт не найден, то апостериорное убеждение о его местоположении обновляется до заштрихованной области в правом нижнем углу. Такое распределение охватывает меньшую область, чем на рис. 6.4, поэтому неопределенность (энтропия) меньше.

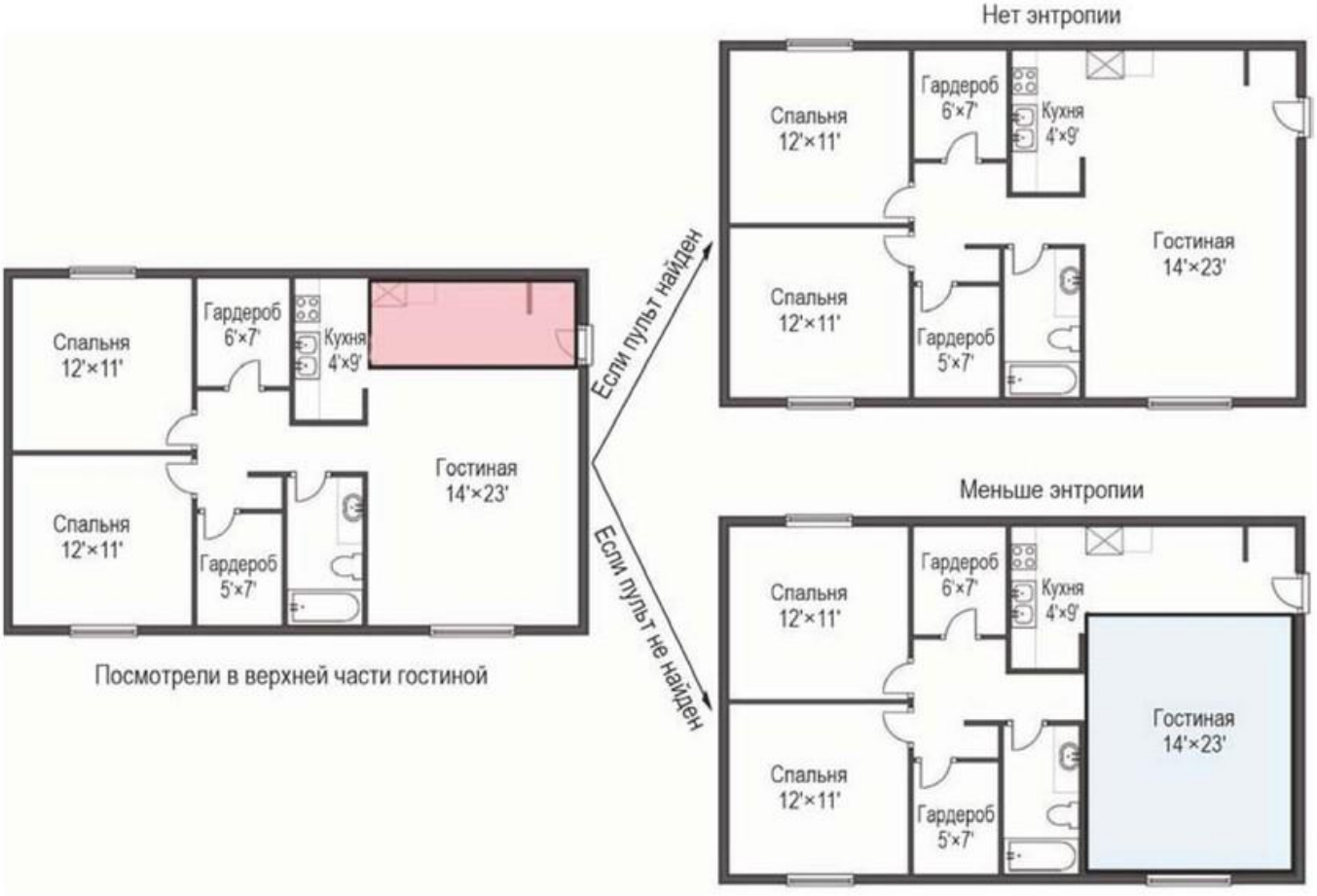


Рис. 6.5. Энтропия местоположения пульта после обыска части гостиной: если пульт найден (вверху справа), то никакой неопределенности не остается; в противном случае энтропия все равно уменьшится (внизу справа), поскольку распределение возможного местоположения пульта стало меньше

В любом случае энтропия снизится, если поискать в указанной части гостиной. Но что произойдет, если поискать пульт, например, в ванной? На рис. 6.6 показаны соответствующие рассуждения:

- Если пульт найден в ванной, то энтропия все равно упадет до нуля. Однако, судя по нашим представлениям, это вряд ли произойдет.
- Если пульт не найден в ванной, то апостериорное убеждение о местонахождении пульта не изменится (по сравнению с рис. 6.4), и результирующая энтропия останется прежней.

Отсутствие пульта в ванной не уменьшает энтропию его местоположения. Другими словами, осмотр ванной не дает никаких дополнительных данных, поэтому согласно теории информации это является неоптимальным решением.



Рис. 6.6. Энтропия местоположения пульта после обыска ванной комнаты: поскольку маловероятно, что пульт будет в ванной, энтропия его апостериорного распределения не меняется

Это сравнение не было бы таким категоричным, если предварительное распределение местоположения пульта (первоначальное предположение о том, где он находится) было бы по всему дому, а не только там, где мы предположили. В конце концов, всегда существует небольшая вероятность, что он потерялся за пределами гостиной. Однако процедура определения места, где стоит поискать (часть дома, которая даст максимальную информацию), остается той же:

1. Рассмотреть апостериорное распределение местоположения пульта, если он был найден. Вычислить энтропию этого распределения.
2. Вычислить ту же энтропию, если пульт не найден.
3. Вычислить среднюю энтропию для двух этих случаев.

4. Повторить вычисления для всех мест, которые вы собираетесь исследовать. Выбрать то, которое даст наименьшую энтропию.

Энтропия дает возможность количественно оценить неопределенность в отношении интересующей величины, используя ее вероятностное распределение теоретико-информационным способом. Данный процесс использует энтропию для определения *действия*, которое максимально ее же и снижает.

> **Примечание**
>
> Это математически элегантная процедура, применимая ко многим ситуациям принятия решений в условиях неопределенности. Мы можем рассуждать о ней как о своего рода поиске истины, в котором мы стремимся предпринять действие, которое приведет нас как можно ближе к цели, максимально уменьшая неопределенность.

6.1.3. Бинарный поиск с применением энтропии

Чтобы лучше понять поиск, основанный на энтропии, рассмотрим, как эта процедура проявляется в одном из классических алгоритмов информатики — бинарном поиске. Скорее всего, вы уже знакомы с ним, поэтому не будем здесь вдаваться в подробности. Понять бинарный поиск начинающим поможет глава 1 книги «Aditya Bhargava's Grokking Algorithms» автора Адитья Бхаргава («Алгоритмы Гроккинга Адитьи Бхаргавы», издательство Manning, 2016). Двоичный поиск используется (на высоком уровне), когда, например, нужно найти позицию определенного числа в отсортированном списке, в котором элементы увеличиваются от первого элемента к последнему.

> **Совет**
>
> Идея двоичного поиска состоит в том, чтобы сравнить целевой элемент с тем, который находится в середине списка. Если он меньше среднего, становится понятно, что дальше нужно искать в первой половине списка. В противном случае — во второй. Процесс сокращения списка пополам повторяется, пока не будет найден целевой элемент.

Рассмотрим конкретный пример, в котором имеется отсортированный список из 100 элементов $[x_1, x_2, ..., x_{100}]$, и необходимо найти местоположение заданного целевого элемента z, предполагая, что тот действительно находится в отсортированном списке.

Как показано на рис. 6.7, двоичный поиск работает путем деления списка на две половины: первые 50 элементов и последние 50 элементов. Поскольку известно, что список отсортирован, можно сделать следующие выводы:

- Если z меньше элемента x_{50}, то нужно учитывать только первые 50 элементов, поскольку последние 50 больше z.
- Если z больше x_{50}, то нужно учитывать только вторую половину списка.

Прекращение поиска

Для каждой итерации (рис. 6.7) опускается ситуация, когда z равно элементу, с которым оно сравнивается. В этом случае можно просто прекратить поиск.

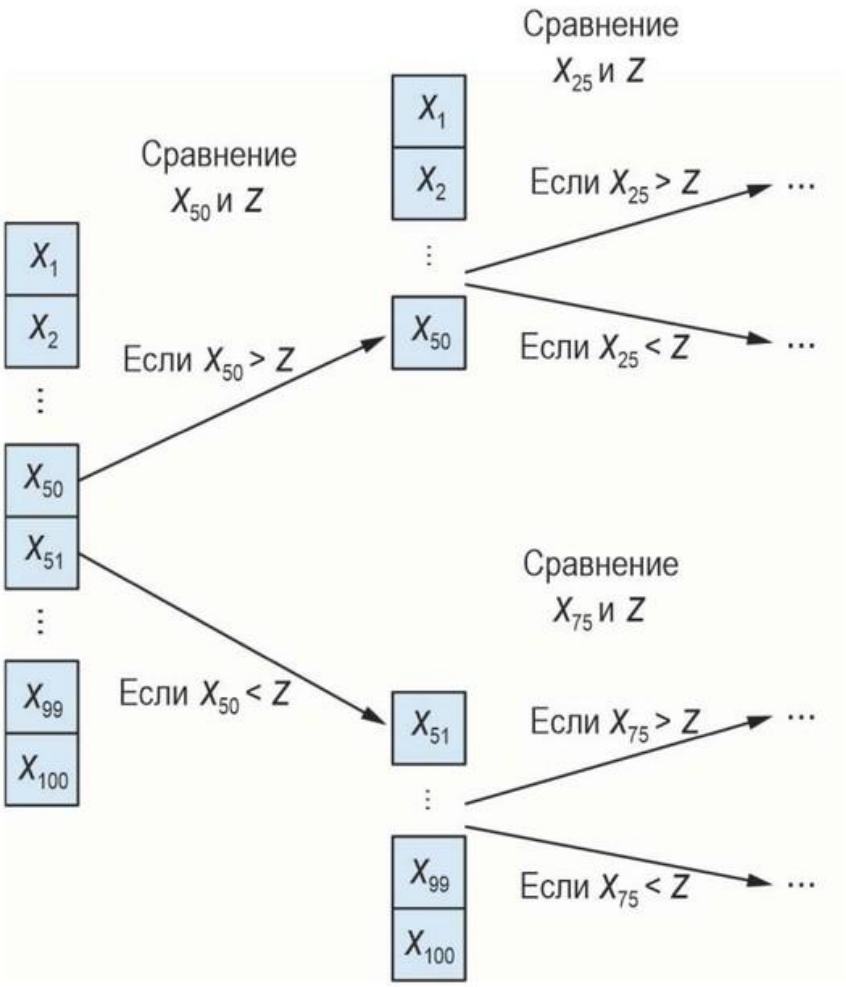


Рис. 6.7. Иллюстрация двоичного поиска в списке из 100 элементов: на каждой итерации целевое значение сравнивается со средним элементом текущего списка; в зависимости от результата из пространства поиска удаляется либо первая, либо вторая половина списка

В среднем процедура помогает найти местоположение z гораздо быстрее, чем последовательный поиск по списку от начала до конца. Если подходить к решению с вероятностной точки зрения, бинарный поиск — это реализация принятия оптимального решения на основе теории информации в задаче по поиску местоположения числа в отсортированном списке.

> **Примечание**
>
> Стратегия двоичного поиска является оптимальным решением для поиска z, позволяя нам найти его в среднем быстрее, чем любая другая стратегия.

Во-первых, используем случайную величину L для обозначения местоположения z в отсортированном списке. Здесь следовало бы задействовать распределение, чтобы описать наше предположение (убеждение) о переменной. Поскольку любое место в списке с одинаковой вероятностью может содержать z, используем для моделирования равномерное распределение.

Рис. 6.8 визуализирует распределение, которое отражает предположение о местоположении z. Поскольку каждый вариант столь же вероятен, как и любой другой, вероятность, что определенное местоположение содержит z, везде составляет $1 \div 100$ (или 1%).

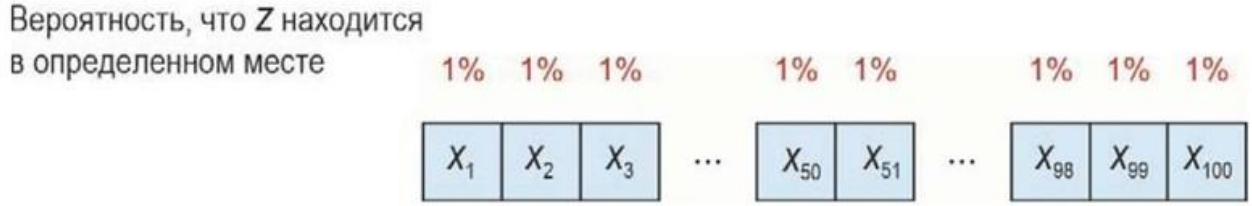


Рис. 6.8. Предварительное распределение z в списке из 100 элементов: поскольку каждое местоположение одинаково вероятно, как и любое другое, вероятность, что данное местоположение содержит z, составляет 1%

ПРИМЕЧАНИЕ

Попробуем вычислить энтропию равномерного распределения. Как нам уже известно, формула энтропии такова: $-\Sigma_i p_i \log p_i$. Мы суммируем различные возможные события, индексированные i. Подставив в формулу значения, имеем следующее:

$$-\sum_{i=2}^{100} 0,0101(\log 0,0101) = 6,63.$$

Таким образом, степень неопределенности в нашем априорном распределении L составляет примерно 6,64.

Далее решим похожий вопрос: как следует выполнять поиск в списке, чтобы найти z как можно быстрее? Сделаем это, следуя процедуре нахождения энтропии, описанной в разделе 6.1.2, где мы стремимся минимизировать энтропию в апостериорном распределении для интересующей величины (в данном случае местоположение L).

Как можно вычислить энтропию апостериорного распределения L после сравнения с конкретным местоположением? Этот расчет требует размышления о том, какой вывод можно сделать о L после поиска в определенном месте, что довольно легко сделать. Допустим, мы решили проверить первым элемент x_1. Согласно нашему убеждению относительно L, существует вероятность 1%, что он находится в этом месте, и вероятность 99%, что он находится в одном из оставшихся элементов:

- Если L действительно находится в этом месте, апостериорная энтропия падает до 0, поскольку в отношении этой величины больше нет неопределенности.
- В противном случае распределение L обновляется, чтобы отразить наблюдение о том, что z не является первым элементом списка.

На рис. 6.9 этот процесс показан в виде диаграммы, на которой нам нужно обновить распределение L так, чтобы каждое из 99 местоположений имело вероятность наличия z, равную $1 \div 99$ (или примерно 1,01%). Вероятность каждого из 99 мест по-прежнему одинакова, но она немного выросла, поскольку мы исключили один элемент из нашего гипотетического сценария поиска.

ПРИМЕЧАНИЕ

Мы рассматриваем только случай, когда z присутствует в списке, поэтому наименьший элемент (x_1) либо равен z, либо меньше него.

Следуя тем же расчетам, можно получить энтропию для нового распределения как

$$-\sum_{i=2}^{100} 0,0101(\log 0,0101) = 6,63.$$

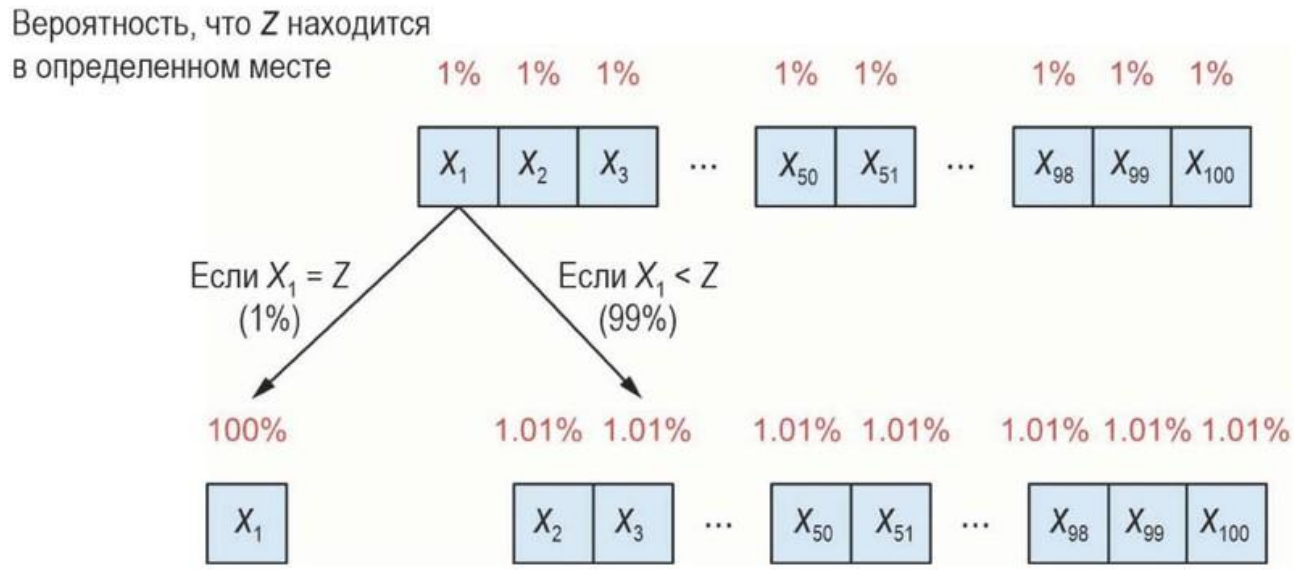


Рис. 6.9. Апостериорные распределения местоположения z в списке после проверки первого элемента: в каждом сценарии вероятность, что z находится в определенном месте, соответствующим образом обновляется

Это апостериорная энтропия L в случае, когда z отсутствует в первом элементе. Последний шаг, который нужно предпринять для вычисления общей апостериорной энтропии после проверки первого места, — это взять среднее значение двух случаев:

- Если z находится в первом элементе (вероятность 1%), то апостериорная энтропия равна 0.
- Если z не находится в первом элементе (вероятность 99%), то апостериорная энтропия равна 6,63.

Взяв среднее, имеем 0,01 (0) + 0,99 (6,63) = 6,56. Таким образом, в среднем мы ожидаем увидеть апостериорную энтропию 6,56, когда решим изучить первый элемент массива. Теперь, чтобы определить, является ли просмотр первого элемента оптимальным решением или есть лучшее место для получения дополнительной информации о L, нужно повторить эту процедуру для других элементов списка. В частности, для любого местоположения нам нужно:

1. Перебрать каждый из потенциальных сценариев, просматривая его.
2. Вычислить апостериорную энтропию распределения L для каждого сценария.
3. Рассчитать среднюю апостериорную энтропию для всех сценариев, исходя из того, насколько вероятен каждый из них.

Сделаем это еще раз, например, для 10-го элемента x_{10}. Соответствующая схема показана на рис. 6.10. Несмотря на то что данный сценарий немного отличается от рассмотренного выше, основная идея остается той же. Во-первых, существуют различные сценарии, которые могут произойти, если наблюдать x_{10}:

1. Элемент x_{10} может быть больше z. В этом случае можно исключить оставшийся 91 элемент из списка и сосредоточить поиск на первых 9, где каждый имеет вероятность содержания z — 11%, а апостериорная энтропия составляет примерно 3,17.

2. Элемент x_{10} может быть равен z. В этом случае апостериорная энтропия снова равна 0.
3. Элемент x_{10} может быть меньше z. В этом случае поиск сужается до последних 90 элементов. Апостериорная энтропия составляет около 6,49.

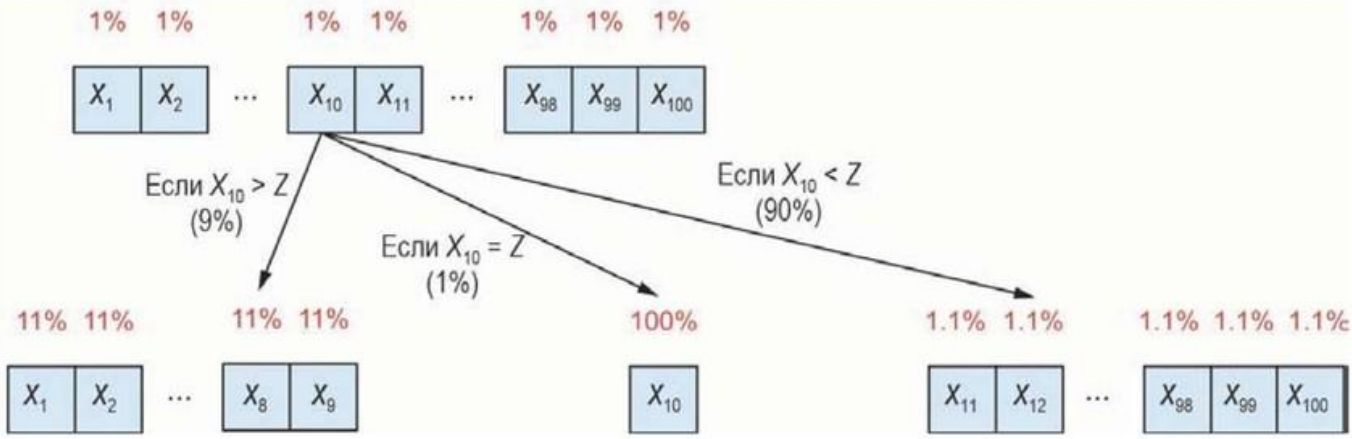


Рис. 6.10. Апостериорные распределения положения z в списке при проверке 10-го элемента: в каждом сценарии вероятность, что z находится в определенном месте, соответствующим образом обновляется

ПРИМЕЧАНИЕ

Обязательно попробуйте вычислить энтропию самостоятельно, чтобы понять, как получаются эти числа.

Далее вычислим средневзвешенное значение этих энтропий, используя соответствующие вероятности: 0,09 (3,17) + 0,01 (0) + 0,9 (6,49) = 6,13. Это число представляет собой ожидаемую апостериорную энтропию (неопределенность), которую мы имеем относительно L после проверки 10-го элемента x_{10}.

Сравнивая результат с тем же числом для первого элемента x_1 (6,56), можно прийти к выводу, что просмотр x_{10} в среднем дает больше информации о L, чем просмотр x_1. Другими словами, проверка x_{10} — лучшее решение с точки зрения теории информации.

Но какое решение будет самым оптимальным, которое даст больше всего информации об L? Чтобы определить это, просто повторим такие же вычисления для других элементов в списке и выберем то, где ожидаемая апостериорная энтропия будет наименьшей.

На рис. 6.11 показана ожидаемая апостериорная энтропия для местоположения z, в зависимости от элемента, который выбран для проверки. Первое, что можно заметить, — это симметрия кривой: взгляд на последний элемент дает такую же ожидаемую апостериорную энтропию (неопределенность), как и на первый. Аналогично 10-й и 90-й элементы дают одинаковое количество информации и т. д.

Еще более важно, что проверка мест посередине (50-й или 51-й элементы в списке) дает нам максимальное количество информации. Так происходит потому, что как только мы сделаем проверку, мы гарантированно исключим половину списка, независимо от того, больше или меньше наше целевое число, чем число в середине.

В других местах такого не наблюдается. Как мы видели ранее, можно исключить и 90 элементов из списка, взглянув на 10-е число, но это происходит только с вероятностью 0,1. Соответственно, глядя на первый элемент, с вероятностью 99% мы сможем исключить только одно число.

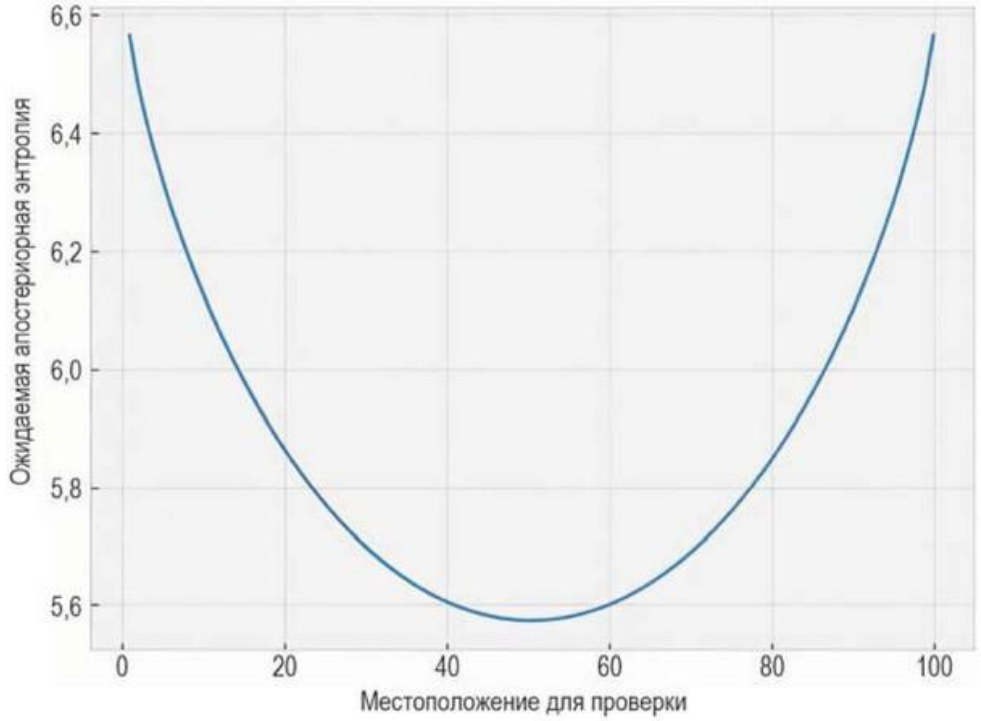


Рис. 6.11. Ожидаемая апостериорная энтропия для *z* в зависимости от расположения в списке для проверки: место посередине является оптимальным и минимизирует ожидаемую энтропию

ПРИМЕЧАНИЕ

В целом проверка элементов посередине максимизирует объем информации, которую мы получаем.

Остальная часть поиска следует той же процедуре: вычисление ожидаемой апостериорной энтропии, которая будет результатом каждого решения, а затем выбор решения, которое минимизирует эту энтропию. Поскольку распределение вероятностей, с которым мы работаем, всегда является равномерным после каждого обновления, оптимальным числом для проверки всегда является элемент в середине массива, который еще не был исключен.

Это и есть стратегия бинарного поиска. С точки зрения теории информации он является оптимальным решением задачи по поиску числа в отсортированном списке.

Обоснование бинарного поиска с помощью теории информации

Помню, когда я впервые узнал об этом алгоритме, то подумал, что стратегия поиска в середине массива казалась хоть и логичной, но вполне необыкновенной и словно «из ниоткуда». Однако мы только что научились получать то же самое решение с точки зрения теории информации, которая количественным способом вполне конкретно определяет идею исключения половины пространства поиска ради получения как можно большего количества данных или снижения как можно большей энтропии.

Применение теории информации и энтропии не ограничивается бинарным поиском. Изученную процедуру можно обобщить и применить к другим задачам принятия решений. Если мы можем смоделировать интересующую задачу с точки зрения распределения вероятностей для неизвестных величин (для предпринимаемых действий), а также для того, как их распределения могут быть обновлены (когда действие принято), тогда мы снова можем выбрать оптимальное действие с точки зрения теории информации (т. е. действие, которое больше всего уменьшает неопределенность в отношении интересующей величины). В оставшейся части этой главы мы применим эту идею к БО и попробуем реализовать полученную политику энтропийного поиска с помощью BoTorch.

6.2. Энтропийный поиск в БО

Используя тот же подход, что и в предыдущем разделе, мы получим политики энтропийного поиска в БО. Основная идея состоит в выборе действий (величин) таким образом, чтобы можно было уменьшить максимально много энтропии в апостериорном распределении. В этом разделе мы сначала обсудим, как это сделать на высоком уровне, а затем перейдем к реализации в BoTorch.

6.2.1. Поиск оптимума с помощью теории информации

В примере с пультом мы стремимся найти предмет в квартире и, таким образом, хотим уменьшить энтропию распределения его местоположения. В бинарном поиске процесс аналогичен: целью является положение определенного числа, которое нужно найти в списке, и нам нужно уменьшить энтропию его распределения. Теперь, чтобы разработать политику энтропийного поиска, мы должны определить, что использовать в качестве цели в БО и как использовать теорию информации для облегчения процесса поиска.

Стоит напомнить, конечная цель использования БО — поиск функции «черного ящика» в области D там, где она максимизируется. Это означает, что естественной целью поиска для нас является местоположение x^*, которое максимизирует целевое значение функции f. То есть $f^* = f(x^*) \geq f(x)$ для всех x в D.

Определение

Оптимальное место x^* часто называют *оптимизатором* целевой функции f.

Учитывая предположение ГП о целевой функции f, существует соответствующее вероятностное убеждение об оптимизаторе x^*, который рассматривается как случайная величина. На рис. 6.12 показан пример обученного ГП, а также вызванное из него распределение оптимизатора x^*. Важно помнить о нескольких интересных характеристиках этого распределения:

- Оно сложное и мультимодальное (имеет несколько локальных оптимумов).
- Наиболее вероятное расположение x^* — немного левее нуля. Именно здесь прогностическое среднее значение ГП максимизируется.

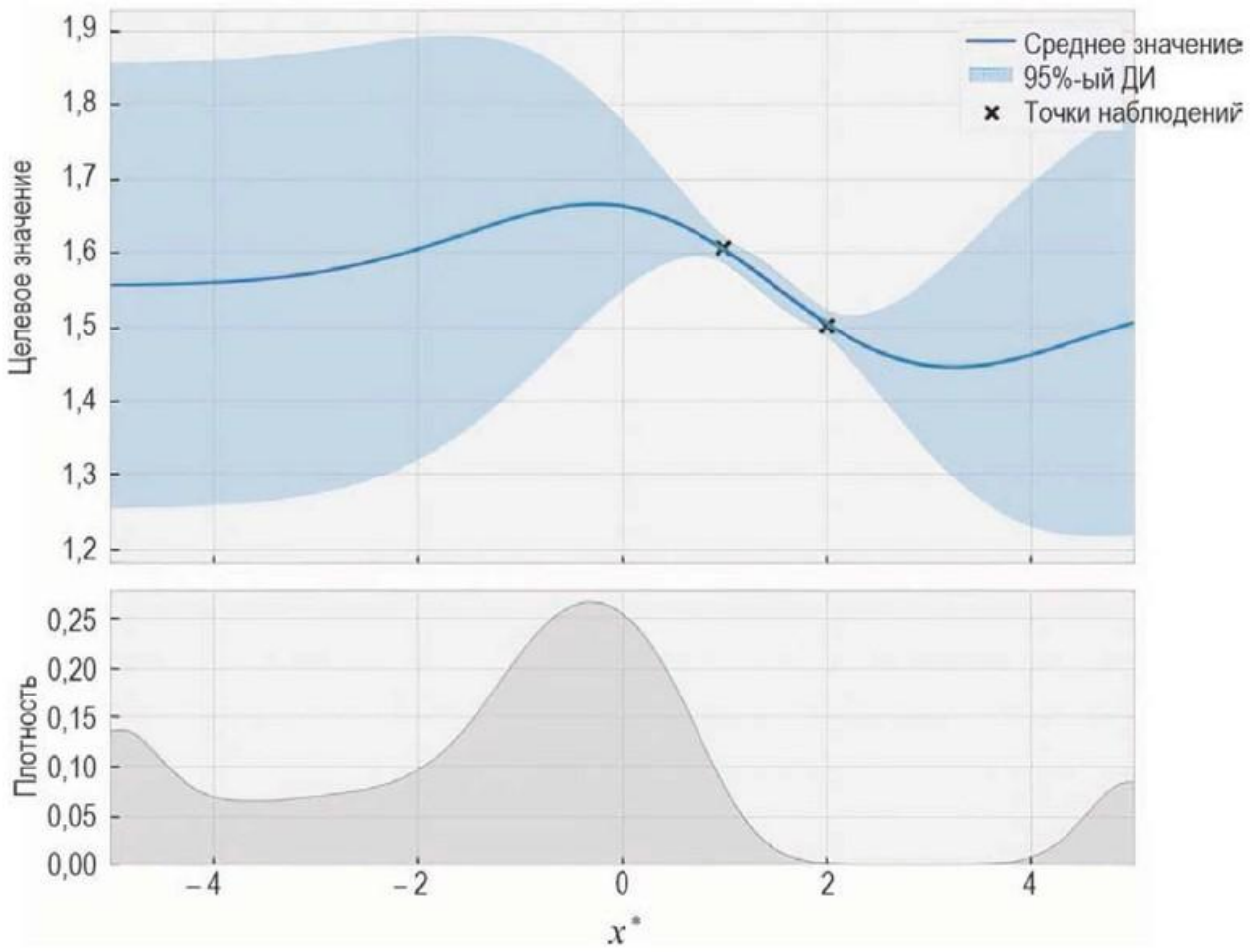


Рис. 6.12. Предположение ГП (вверху) и распределение оптимизатора функции x^* (внизу): распределение оптимизатора является негауссовым и довольно сложным, что затрудняет моделирование и принятие решений

- Существует немалая вероятность, что x^* находится в конечной точке –5 или 5.
- Вероятность, что x^* находится возле 2, почти равна нулю. Это соответствует факту, что мы уже наблюдали более высокое объективное значение, чем $f(2)$.

Эти характеристики делают моделирование распределения x^* очень сложным. Самый простой метод измерения, как распределяется эта величина, — просто взять множество выборок из ГП и записать место, в котором каждая из них максимальна. Фактически именно так создается рис. 6.12.

Ситуация усугубляется тем, что нам нужно намного больше (экспоненциально) выборок для оценки распределения x^*, когда размерность целевой функции (длина входного вектора x или количество признаков, которые имеет каждый x) увеличивается.

> **Определение**
>
> Это пример *проклятия размерности* (**curse of dimensionality**), которое в ML часто используется для обозначения экспоненциальной стоимости многих процедур по отношению к размерности интересующего объекта.

Чтобы использовать энтропийный поиск в БО, нужно смоделировать убеждение о местоположении оптимального оптимизатора x^*, используя распределение веро-

ятностей. Однако невозможно точно смоделировать данное распределение. Вместо этого мы должны аппроксимировать его, используя выборки из ГП. К сожалению, такой процесс быстро становится очень затратным в вычислительном плане по мере увеличения числа измерений целевой функции (длины x).

Примечание

На самом деле в БО есть исследовательские работы, в которых делается попытка найти местоположение оптимизатора x^* с использованием энтропии. Однако полученные в результате политики часто оказываются слишком дорогими для выполнения и не реализуются в BoTorch.

Но это не значит, что нужно вообще отказаться от использования теории информации в БО. Нам просто следует изменить процедуру поиска, чтобы сделать ее более подходящей для вычислительных методологий. Один из простых способов добиться этого — ориентироваться на величину, отличную от x^*, которая, с одной стороны, связана с поиском x^*, а с другой — легче поддается рассуждениям.

Величиной, представляющей интерес для оптимизации, отличной от x^*, является оптимальное значение $f^* = f(x^*)$, достигнутое в оптимизаторе. Оно также является случайной величиной согласно убеждению ГП о целевой функции f. Знание оптимального значения f^* может многое рассказать об x^*. Эти две величины связаны с точки зрения теории информации. Однако с f^* работать гораздо проще, чем с x^*, поскольку первая представляет собой просто вещественное число, а второй — вектор длины, равной числу измерений целевой функции.

Пример распределения оптимального значения f^*, индуцированного ГП, показан на рис. 6.13 (правая часть). На нем видно, что это распределение примерно усечено около значения 1,6, что в точности соответствует значению действующей точки в наборе обучающих данных. Это имеет смысл, поскольку f^* должно быть не ниже действующего значения 1,6.

Примечание

Основным преимуществом сосредоточения усилий на распределении f^* является тот факт, что оно всегда одномерно, независимо от размерности целевой функции.

В частности, мы можем извлечь выборки из этого одномерного распределения, чтобы аппроксимировать ожидаемую апостериорную энтропию при выполнении запроса. С этого момента мы будем следовать той же идее, что и при поиске по энтропии: выбираем запрос, который (приблизительно) минимизирует ожидаемую апостериорную энтропию или, другими словами, максимизирует ожидаемое снижение энтропии.

Определение

Используя величину ожидаемого снижения энтропии в качестве оценки приобретения, мы получаем политику *поиска энтропии с максимальным значением* (**Max-value Entropy Search**, **MES**). Термин «максимальное значение» обозначает, что мы используем теорию информации для поиска максимального значения или оптимального значения f^* целевой функции.

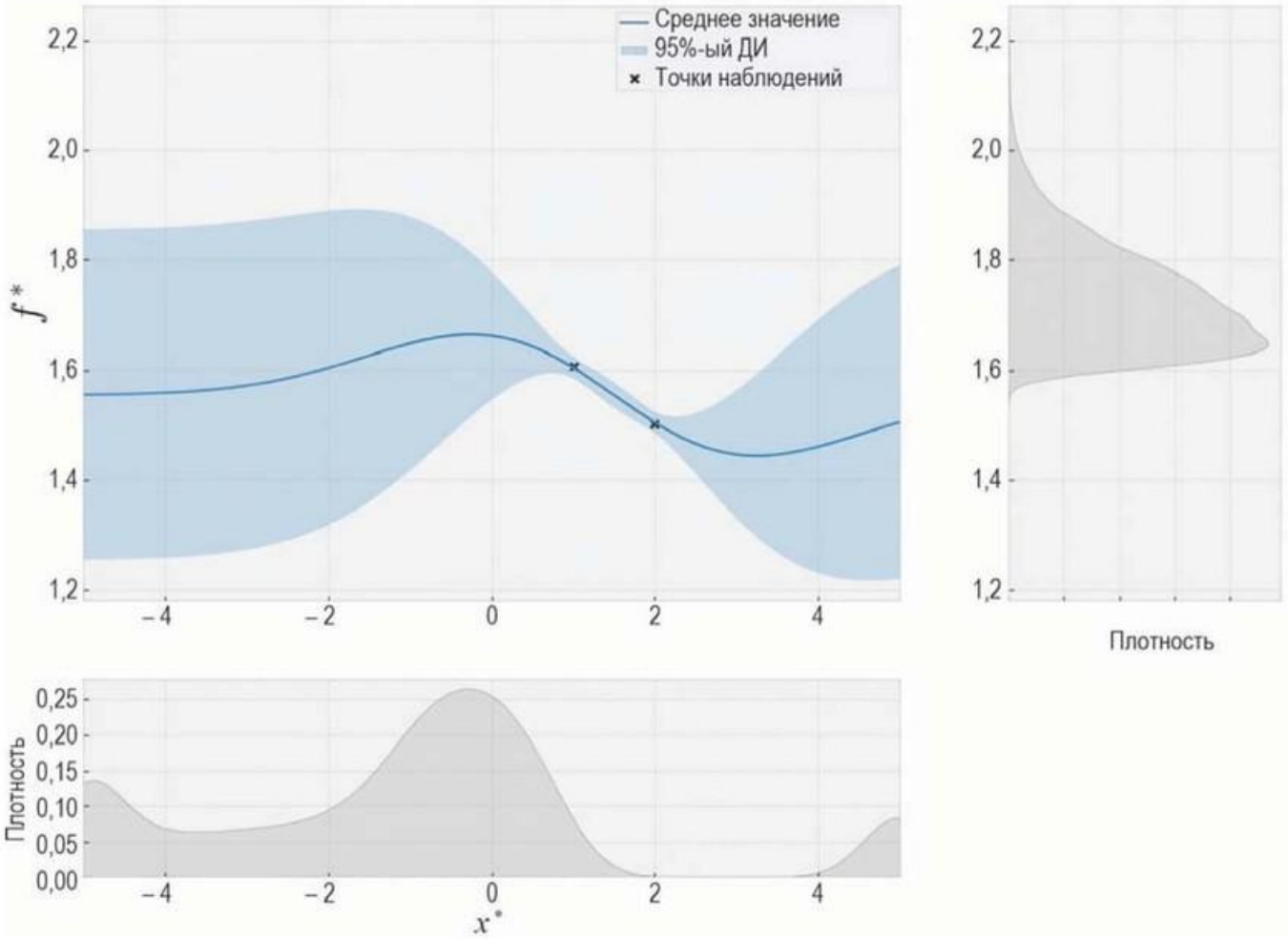


Рис. 6.13. Убеждение ГП (вверху слева), распределение оптимизатора x^* (внизу) и распределение оптимального значения f^* (справа): распределение f^* всегда одномерно и, следовательно, с ним проще работать, чем с оптимизатором

На рис. 6.14 изображена оценка приобретения политики MES для текущего примера (внизу). Точка возле –2 указывает, где следует выполнять следующий запрос в соответствии с критерием, основанным на теории информации. Политика MES отдает предпочтение этому местоположению, поскольку оно имеет как относительно высокое среднее значение, так и высокий ДИ, что обеспечивает баланс между исследованием и эксплуатацией.

Интересно, что оценка приобретения здесь чем-то похожа на распределение оптимизатора x^*, которое показано на рис. 6.13. Мы видим, что кривая:

1. Достигает своего зенита где-то посередине.
2. Достигает существенного значения в конечных точках.
3. Доходит до 0 возле значения 2, когда мы точно знаем, что местоположение не оптимально.

Это указывает на то, что f^* тесно связано с x^*. И, несмотря на то, что мы теряем некоторую информацию, изменяя нашу цель, поиск f^* является хорошим показателем для поиска x^* и гораздо более удобен в вычислительном отношении.

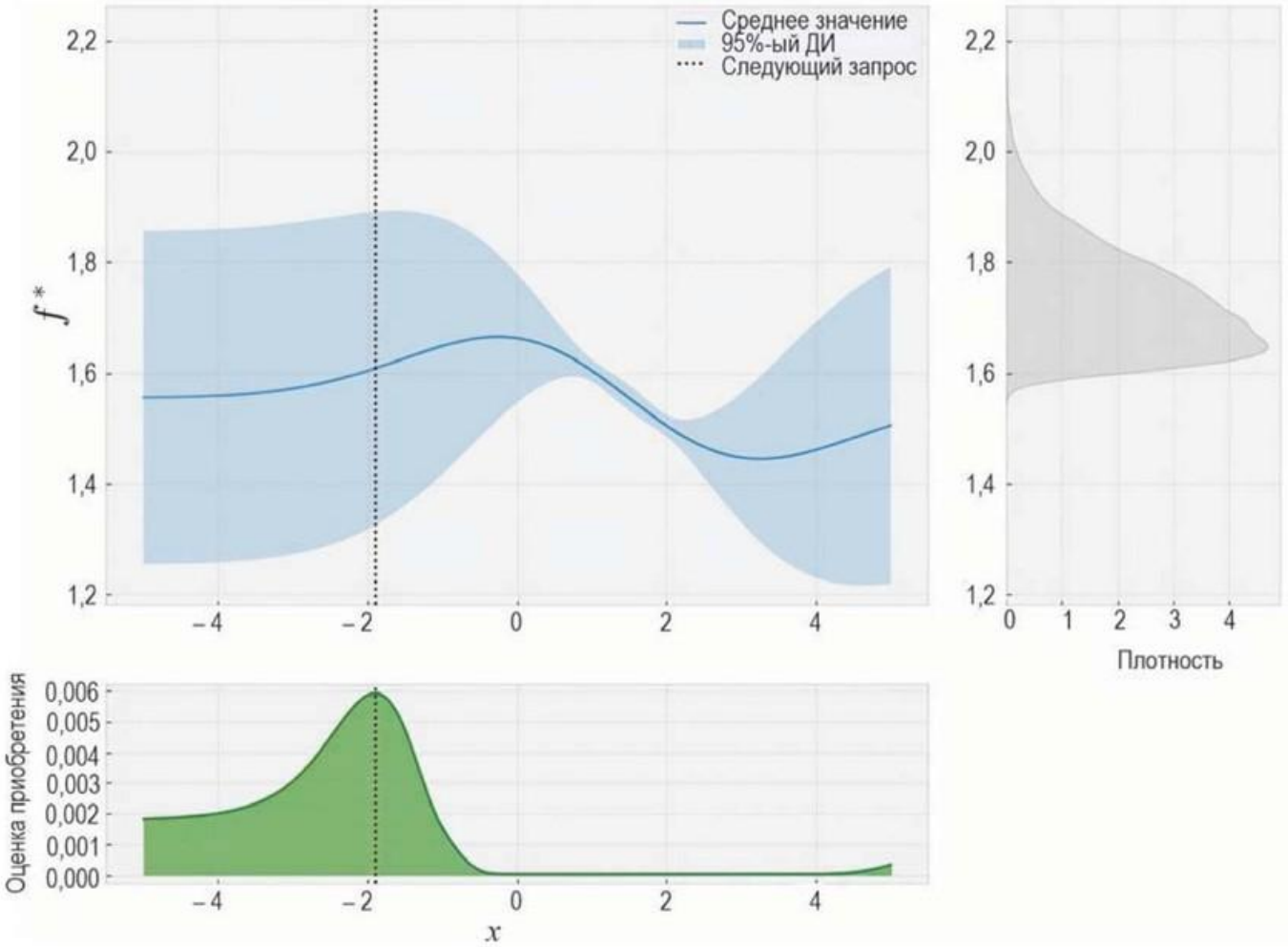


Рис. 6.14. Убеждение ГП (вверху слева), распределение оптимального значения *f** (справа) и приблизительное ожидаемое снижение энтропии, которое используется в качестве оценки приобретения (внизу): *f** всегда одномерно и, следовательно, с ним проще работать, чем с оптимизатором

6.2.2. Реализация энтропийного поиска с помощью BoTorch

Теперь мы готовы реализовать MES с помощью BoTorch. Данная политика реализуется классом `qMaxValueEntropy` в `botorch.acquisition.max_value_entropy_search` как модуль PyTorch (аналогично большинству политик, которые мы видели). При инициализации этот класс принимает два аргумента: модель ГП и набор точек, которые будут использоваться в качестве выборок в процедуре аппроксимации (приближения), описанной в предыдущем разделе.

Несмотря на то что существует множество способов создания этих точек, один из них (который мы узнали в разделе 5.3.2) — это последовательность Соболя, которая лучше справляется с охватом целевого пространства. В целом политика MES реализуется следующим образом:

```
#Генерируем выборки от 0 до 1, используя последовательность Соболя
num_candidates = 1000
sobol = torch.quasirandom.SobolEngine(1, scramble=True)
candidate_x = sobol.draw(num_candidates)
```

```
#Изменяем масштаб выборок так, чтобы они находились внутри домена
candidate_x = 10 * candidate_x - 5

#Объявляем объект политики MES
policy = botorch.acquisition.max_value_entropy_search.qMaxValueEntropy(
    model, candidate_x
)

#Вычисляем оценки приобретения
with torch.no_grad():
    acquisition_score = policy(xs.unsqueeze(1))
```

Здесь `num_candidates` — это настраиваемый параметр, устанавливающий количество выборок, которые мы хотим использовать в вычислениях MES. Большее значение будет означать аппроксимацию (приближение) с более высокой точностью, но потребует более высоких вычислительных затрат.

Далее применим этот код к нашей текущей задаче оптимизации одномерной функции Форрестера, реализованной в `CH06/01 – BayesOptloop.ipynb`. Мы уже знакомы с большей частью этого кода, поэтому не будем вдаваться в подробности.

На рис. 6.15 показан прогресс политики по 10 запросам, при этом MES быстро находит глобальный оптимум функции Форрестера после пяти запросов. Интересно, что по мере продвижения оптимизации растет уверенность, что просмотр других областей в пространстве поиска не приведет к существенному уменьшению энтропии. Это помогает нам оставаться ближе к оптимальному местоположению.

Можно убедиться, что теория информации дает нам принципиальную, математически элегантную основу для принятия решений, которая вращается вокруг попыток узнать как можно больше об интересующей величине. Это сводится к уменьшению ожидаемой апостериорной энтропии в распределении, которое моделирует интересующую целевую функцию.

В рамках БО мы увидели, что прямое применение этой процедуры создает вычислительные проблемы при моделировании местоположения оптимального значения для нашей цели поиска. Вместо этого мы переключаем внимание на оптимальное значение самой целевой функции, делая вычисления более удобными. К счастью, BoTorch прекрасно абстрагирует всю математику, оставляя нам удобный модульный интерфейс, который можно подключить к любой задаче оптимизации.

На этом можно закончить главу 6 и вторую часть книги о политике БО. В последних трех главах были рассмотрены некоторые из наиболее часто используемых эвристик для принятия решений в БО и соответствующие политики, начиная от поиска улучшений для действующей точки до заимствования методов многорукого бандита и использования теории информации.

Оставшаяся часть книги выводит наше обсуждение на новый уровень, добавляя специальные настройки оптимизации, которые отличаются от тех, что мы видели до этого момента (где мы последовательно наблюдаем одну точку данных на каждом этапе оптимизации). В последующих главах поднимается вопрос, как изу-

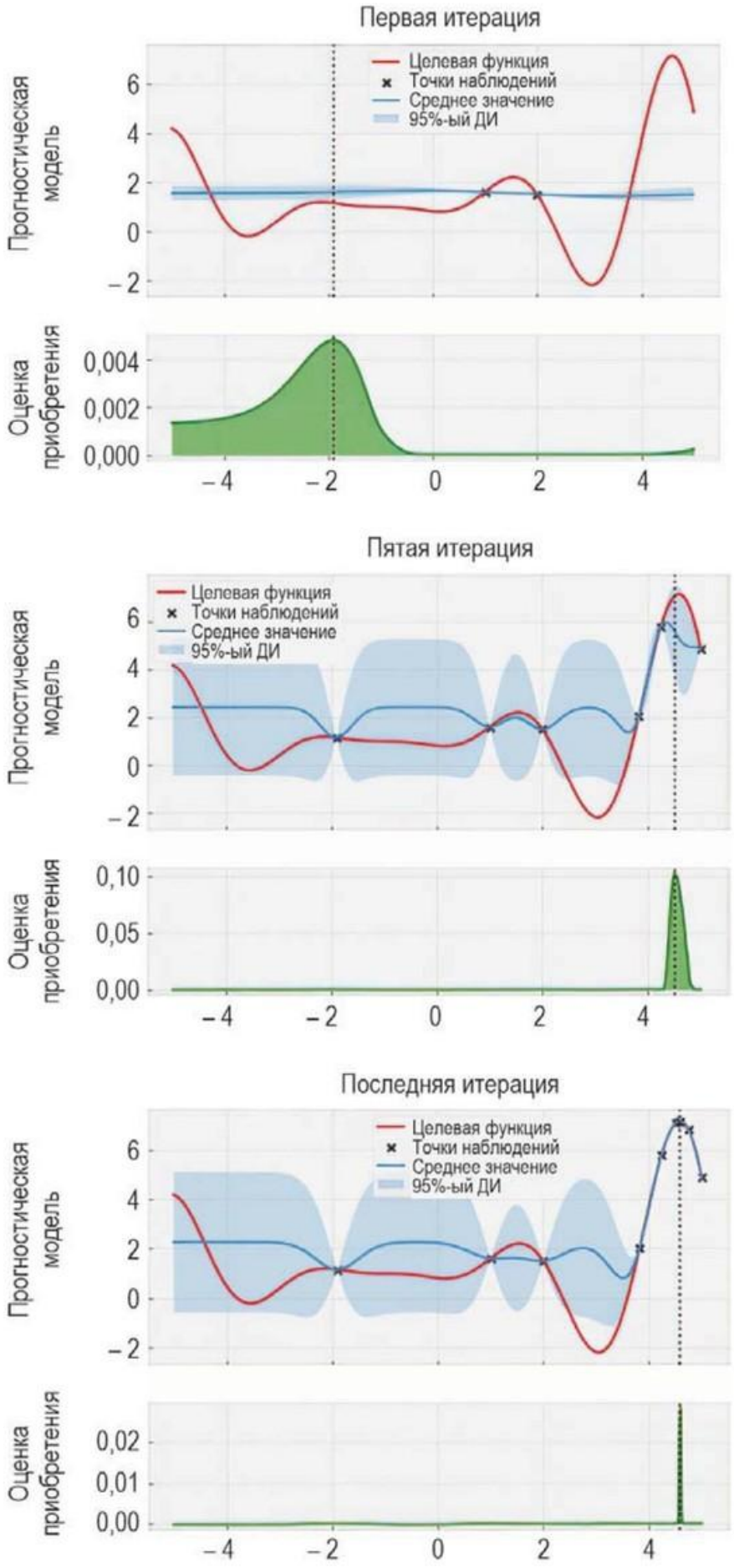


Рис. 6.15. Прогресс политики MES: она быстро находит глобальный оптимум после пяти запросов

ченные методы можно применить к практическим условиям реального мира для ускорения оптимизации.

6.3. Упражнения

Эта глава содержит два упражнения:

1. Бинарный поиск, в котором априорная информация может быть учтена при принятии решений.
2. Процесс реализации MES в задаче по настройке гиперпараметров, рассмотренной в предыдущих главах.

6.3.1. Упражнение 1: Включение предварительных (априорных) знаний в энтропийный поиск

В разделе 6.1.3 мы видели, что при равномерном априорном распределении оптимальным теоретико-информационным решением поиска является разделение массива пополам. Что произойдет, если равномерное распределение не будет верно отражать априорное предположение и нужно будет использовать другое распределение? Это упражнение (`CH06/02 - Exercise 1.ipynb`) показывает подобный пример, а также учит, как в итоге получить оптимальный результат. Решение данной задачи должно помочь еще больше оценить элегантность и гибкость энтропийного поиска в качестве общей процедуры принятия решений в условиях неопределенности.

Представьте себе следующий сценарий: вы работаете в отделе контроля качества компании-производителя телефонов, и ваш текущий проект — провести стресс-тестирование прочности для корпуса нового продукта. В частности, вашей команде необходимо выяснить, с какого этажа 10-этажного дома можно уронить телефон на землю и не разбить его. Применяются несколько правил:

- Инженеры, создавшие телефон, уверены, что он не сломается, если упадет с первого этажа.
- Если телефон сломался при падении с определенного этажа, то он сломается и при падении с этажей повыше.

Вам предстоит с помощью испытаний найти самый высокий этаж, с которого можно уронить телефон, не разбив его (обозначим этот неизвестный этаж как X). Другими словами, вы должны сбросить реальные телефоны с определенных этажей, чтобы определить X. Вопрос заключается в следующем: как следует выбирать, с каких этажей сбрасывать телефоны, чтобы найти X? Поскольку устройства имеют большую стоимость, необходимо провести как можно меньше испытаний и использовать теорию информации для облегчения поиска. Выполните следующие шаги:

1. Предположите, что, принимая во внимание физику, материалы и конструкцию телефона, инженеры имеют первоначальное предположение о том, с какой высоты устройство может упасть без последствий.

   В частности, априорное распределение X является экспоненциальным, а именно вероятность, что X равен числу, обратно экспоненциальна по отношению к это-

му числу: $Pr(X = n) = 1/2n$, для $n = 1, 2, ..., 9$. Вероятность, соответствующая самому высокому (десятому) этажу, равна $Pr(X = 10) = 1/29$. Таким образом, вероятность, что $X = 1$, равна 50%, и она уменьшается вдвое по мере увеличения числа. Это распределение вероятностей показано на рис. 6.16.

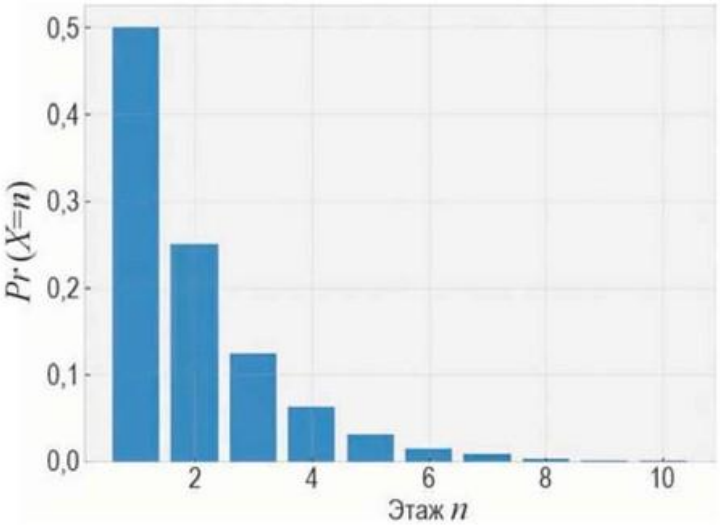


Рис. 6.16. Вероятность, что X равен числу от 1 до 10 (вероятность, что этаж является самым высоким, при падении с которого телефон не сломается)

Убедитесь, что это распределение вероятностей верно, доказав, что сумма вероятностей равна единице. Иначе говоря, докажите, что $Pr(X = 1) + Pr(X = 2) + ... + Pr(X = 10) = 1$.

2. Рассчитайте энтропию данного априорного распределения, используя формулу, приведенную в конце раздела 6.1.1.
3. Учитывая априорное распределение, определенное между 1 и 10, определите какова вероятность, что телефон сломается при падении со второго этажа? Какова эта вероятность для пятого этажа? Какова эта вероятность для первого этажа?
4. Предположите, что после наблюдения за результатом любого испытания апостериорное распределение X снова становится экспоненциальным и определяется между самым низким и самым высоким возможными этажами.

   Например, если телефон не сломался при падении с пятого этажа, то мы знаем, что X как минимум равен значению 5, а апостериорное распределение X таково: $Pr(X = 5) = 1/2$, $Pr(X = 6) = 1/4$, ..., $Pr(X = 9) = 1/32$, $Pr(X = 10) = 1/32$. Если же телефон сломался при падении с пятого этажа, то мы знаем, что X не превышает значения 4, а его апостериорное распределение таково: $Pr(X = 1) = 1/2$, $Pr(X = 2) = 1/4$, $Pr(X = 3) = 1/8$, $Pr(X = 4) = 1/8$. На рис. 6.17 показаны оба этих сценария.

   Вычислите энтропию этого предполагаемого апостериорного распределения в обоих случаях.
5. Учитывая априорное распределение, вычислите ожидаемую апостериорную энтропию после того, как испытание на пятом этаже проведено (т. е. после того, как телефон упадет с пятого этажа, и мы понаблюдаем, не сломался ли он).

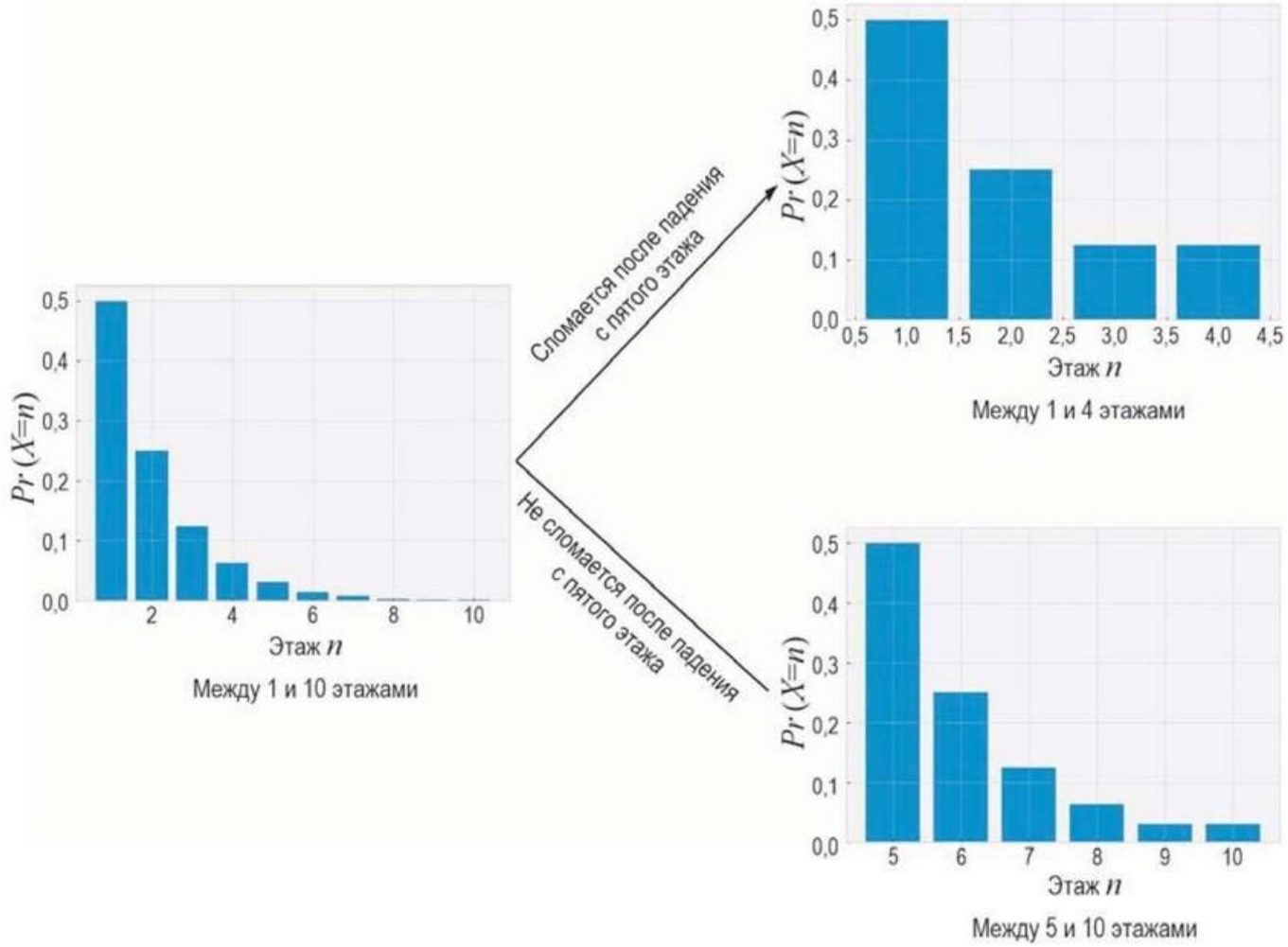


Рис. 6.17. Апостериорные распределения вероятностей для X в двух разных сценариях, когда телефон упал с пятого этажа: каждое апостериорное распределение по-прежнему является экспоненциальным

6. Вычислите ожидаемую апостериорную энтропию для других этажей. Какой этаж дает наибольшее снижение энтропии? Остается ли результат прежним, как при бинарном поиске? Если нет, то что изменилось?

6.3.2. Упражнение 2: Байесовская оптимизация для настройки гиперпараметров

В этом упражнении (`CH06/03 - Exercise 2.ipynb`) мы применяем БО к целевой функции, которая моделирует поверхность точности для машины опорных векторов в задаче по настройке гиперпараметров. Ось X обозначает значение штрафного параметра C, а ось Y — значение параметра γ ядра RBF. Более подробную информацию можно найти в упражнениях к главам 3 и 4. Выполните упражнение, используя следующие действия:

1. Воссоздайте цикл БО из `CH06/03 - Exercise 2.ipynb`, включая внешний цикл, реализующий повторяющиеся эксперименты.
2. Запустите политику MES. Поскольку целевая функция является двумерной, следует увеличить размер последовательности Соболя, который использует поли-

тика. Например, можно установить его на уровне 2000. Понаблюдайте за его совокупной производительностью.

Повторные эксперименты в БО

Обратитесь к шагу 9 упражнения 2 из главы 4, чтобы узнать, как можно проводить повторные эксперименты в БО.

Заключение

- Теория информации изучает представление, количественную оценку и передачу информации. Одной из основных концепций в этой области является энтропия, которая количественно определяет неопределенность в отношении случайной величины на основе распределения ее вероятностей.
- Процедура поиска энтропии учитывает ожидаемое ее снижение (и, следовательно, неопределенности) в отношении интересующей величины с помощью выбора действия, которое максимизирует это снижение. Эту общую процедуру можно применять ко многим проблемам принятия решений в условиях неопределенности.
- Бинарный поиск может быть получен как результат энтропийного поиска, примененного к задаче нахождения определенного числа в отсортированном массиве.
- Проклятие размерности относится к экспоненциальной стоимости многих процедур в ML относительно размерности интересующего объекта. По мере увеличения количества измерений для завершения процедуры требуется экспоненциально больше времени.
- В БО, несмотря на то, что энтропийный поиск может быть применен к проблеме определения местоположения оптимизатора функции, он требует больших вычислительных затрат из-за проклятия размерности.
- Чтобы преодолеть проклятие размерности, нужно модифицировать целевую функцию нахождения оптимизированного значения, превратив ее в одномерную задачу поиска. Полученная в результате политика называется поиском максимального значения энтропии.
- Из-за сложного поведения глобального оптимума функции, которую моделирует ГП, невозможно вычислить оценки приобретения MES в закрытой форме. Однако можно извлечь выборки из распределений вероятностей, чтобы аппроксимировать оценки приобретения.
- Реализация MES в BoTorch осуществляется по той же процедуре, что и реализация других политик БО. Чтобы облегчить процедуру выборки при аппроксимации (приближении) оценки приобретения, используется последовательность Соболя при инициализации объекта политики.

Часть III

Расширение байесовской оптимизации на специализированные сценарии

Изученный нами цикл БО покрывает широкий спектр задач по оптимизации. Однако реальные случаи часто не соответствуют этой идеализированной модели. Что если нам нужно одновременно выполнять несколько оценок функций? Такое часто встречается в приложениях с настройкой гиперпараметров, где доступно несколько графических процессоров. Что делать, если у нас есть несколько конкурирующих целей, которые мы хотим оптимизировать? В этой части книги представлены некоторые из наиболее распространенных сценариев оптимизации, с которыми можно столкнуться в реальном мире, а также рассуждения на тему, как распространить БО на подобные случаи.

Для увеличения пропускной способности многие настройки позволяют проводить эксперименты параллельно. В главе 7 представлена структура БО, в которой вычисления функций выполняются *пакетами*. Мы узнаем, как расширить политику принятия решений (изученную в части 2) на данный сценарий, гарантируя при этом преимущества параллелизма в полной мере.

В критически важных для безопасности случаях нет возможности свободно исследовать пространство поиска, поскольку некоторые запросы к функциям могут иметь пагубные последствия. Это создает сценарии, при которых вводятся ограничения на то, как должна вести себя рассматриваемая функция. Существует необходимость учитывать некоторые рамки при разработке политики оптимизации. В гла-

ве 8 рассматривается вариант БО, который называется *ограниченной оптимизацией*, и разрабатываются необходимые механизмы для ее применения.

В главе 9 исследуется ситуация, в которой мы имеем доступ к множеству способов наблюдения за значениями функции с разными уровнями затрат и точности. Такой вариант БО широко известен как *разноточность* (**multifidelity**). Из этой главы мы также узнаем о естественном расширении энтропии для количественной оценки, насколько значимы проведения измерений на различных уровнях точности, а также применим алгоритм для балансировки информации и затрат.

Как уже было сказано, парные сравнения отражают предпочтения человека точнее, чем числовые оценки или рейтинги, поскольку они более просты и создают меньшую когнитивную нагрузку на маркировщика. В главе 10 БО применяется к подобному сценарию сначала с использованием специальной модели ГП, а затем путем изменения существующих политик, чтобы они соответствовали рабочему процессу попарного сравнения.

Многоцелевая оптимизация — это распространенный вариант использования, в котором мы стремимся одновременно оптимизировать несколько потенциально конфликтующих целей. Мы изучим задачу многоцелевой оптимизации и разработаем решение, которое совместно оптимизирует несколько целей.

В этом разнообразном наборе специализированных случаев оптимизации есть общая тема: компромисс между исследованием и эксплуатацией при учете структуры задачи. Увидев в этой части, как БО применяется к широкому спектру реальных сценариев, мы не только укрепим понимание этого метода, но и сделаем его более применимым в практическом использовании. Код, разработанный в этих главах, поможет сразу же приступить к решению любой задачи оптимизации, которая может возникнуть в реальной жизни.

7 Максимизация производительности за счет оптимизации пакетной обработки

Темы главы:

- Выполнение оценок функций в пакетном режиме.
- Распространение БО на пакетный режим.
- Оптимизация трудновычислимых оценок приобретения.

Цикл БО обрабатывает один запрос за раз и возвращает оценку функции от текущего запроса перед выполнением следующего. Мы используем этот цикл, когда оценки функций могут выполняться только последовательно. Однако многие реальные сценарии оптимизации «черного ящика» позволяют пользователю делать это в *пакетном режиме*. Например, при настройке гиперпараметров ML-модели можно попробовать запускать различные их сочетания параллельно, если есть доступ к нескольким процессорам или компьютерам, вместо запуска отдельных комбинаций одну за другой. Воспользовавшись всеми доступными ресурсами, можно увеличить количество проводимых экспериментов и максимизировать пропускную способность еще на этапе оценки функции.

Вариант БО, в котором несколько запросов могут выполняться параллельно, называется *пакетной байесовской оптимизацией* (**Batch Bayesian optimization**). В качестве реального примера можно привести открытие медицинских препаратов, когда ученые используют каждую из имеющихся вычислительных машин в своей лаборатории для синтеза индивидуального прототипа лекарства или рекомендации продуктов, когда алгоритм предоставляет покупателю несколько продуктов на выбор одновременно. В целом любая настройка оптимизации «черного ящика», при которой одновременно может проводиться более одного эксперимента, подразумевает пакетную БО.

Учитывая, что вычислительные и физические ресурсы часто можно распараллеливать, пакетный вариант является одной из наиболее распространенных байесовских оптимизаций в реальном мире. В этой главе мы познакомимся с данным сценарием и увидим, как политики, изученные в предыдущих главах, могут быть расширены

для его реализации. Мы обсудим, почему это не является тривиальной задачей и требует тщательного рассмотрения. Также мы узнаем о различных стратегиях, которые облегчают осуществление пакетного сценария, и о том, как реализовать политики БО в Python с помощью BoTorch. К концу главы вы будете понимать, что представляет собой пакетная БО, когда она применяется и как реализовать политики в подобных ситуациях. Зная, как распараллелить оптимизацию в пакетном режиме, можно сделать данную задачу более практичной и применимой в реальном мире.

7.1. Одновременное выполнение нескольких оценок функций

Возможность одновременно выполнять несколько оценок функций при оптимизации «черного ящика» распространена во многих реальных сценариях. При этом пакетный вариант — это такой случай байесовской оптимизации, в котором используется параллелизм. В этом разделе мы рассмотрим точную настройку пакетной БО, а также с какими проблемами можно столкнуться, используя политики для выполнения множественных запросов к целевой функции. Здесь будут рассмотрены различные стратегии по расширению политик для пакетного режима. Рассмотрим их далее в этой главе, начиная с раздела 7.2.

7.1.1. Параллельное использование всех доступных ресурсов

Одной из определяющих характеристик для задачи оптимизации «черного ящика» является то, что вычисление функций может оказаться непомерно дорогостоящим. В разделе 1.1 мы рассмотрели высокую стоимость вычислений во многих приложениях. Примером может служить настройка нейросети с помощью гиперпараметров, которая требует много времени и вычислительных ресурсов, или затраты на создание новых лекарств, которые в последние годы растут в геометрической прогрессии. Высокая стоимость запроса приводит к необходимости делать процесс более эффективным. Один из способов добиться этого — использовать *параллелизм*.

> **Определение**
>
> *Параллелизм* означает одновременное выполнение независимых процессов, при котором общее время, необходимое для их завершения, сокращается.

Преимущества параллелизма суммированы на рис. 7.1, где три процесса (которые могут быть программами, вычислениями и т. д.) выполняются либо последовательно, либо параллельно. При втором варианте они занимают лишь одну треть от общего времени, необходимого в первом случае.

Параллелизм особенно распространен в информатике, где компьютер может использовать несколько процессоров (или ядер) сразу для одновременной обработки нескольких программ. Если те, в свою очередь, независимы друг от друга (не используют данные друг друга и не пишут в одни и те же файлы), они могут работать параллельно без каких-либо проблем.

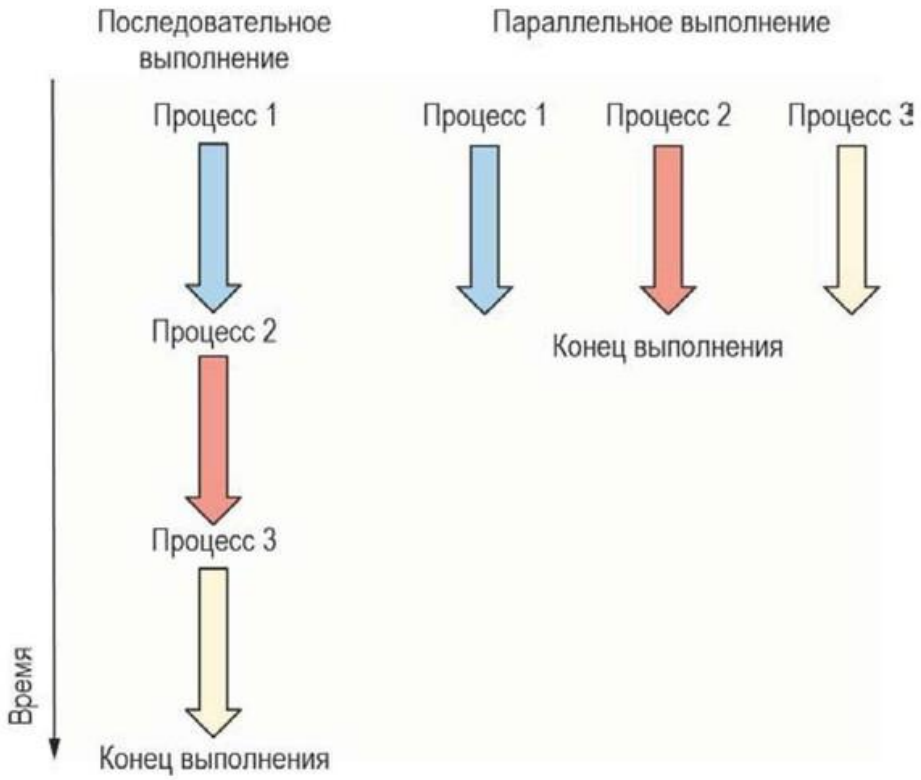


Рис. 7.1. Иллюстрация преимуществ параллелизма: три процесса выполняются либо последовательно (слева), либо параллельно (справа); при параллельной работе эти процессы занимают лишь одну треть от общего времени, необходимого при последовательном запуске

Та же идея применима к сценариям оптимизации, использующим БО. Например, ML-инженер, настраивающий нейросеть, может использовать сразу все графические процессоры, к которым у него есть доступ, для одновременного обучения нескольких моделей. Или ученый, пытающийся открыть новые лекарства, может экспериментировать с несколькими потенциальными вариантами сразу, используя лабораторное оборудование для одновременного синтеза нескольких рецептов. Выполнение множества запросов за один раз позволяет получить больше информации при том же количестве времени, потраченном на изучение целевой функции.

Что такое графические процессоры?

Графический процессор (**Graphics Processing Units**, **GPU**) — это аппаратное обеспечение, оптимизированное для выполнения параллельного умножения матриц. Поэтому их обычно используют для обучения нейросетей.

Возьмем, к примеру, выпечку партии печенья. Можно испечь их по одному за раз, но это приведет к пустой трате ресурсов (мощность духовки и время). Вместо этого гораздо проще испечь несколько печенек сразу.

БО для выпечки печенья

Что касается выпечки, существует исследовательская работа по пакетной БО, в которой рассматривается оптимизация рецептов печенья с помощью поиска оптимального количества яиц, сахара и корицы при приготовлении теста.

В байесовской оптимизации пакетный режим позволяет одновременно оценивать несколько входных данных целевой функции. Мы можем отправить множество

запросов (x_1, x_2, ..., x_k) в «черный ящик», выполнить оценку и получить соответствующие выходные значения $f(x_1)$, $f(x_2)$, ..., $f(x_k)$ сразу в пакете. В классической последовательной БО только после наблюдения $f(x_1)$ можно перейти к запросу x_2.

На каждой итерации пакетного цикла БО мы выбираем несколько входных мест поиска для оценки целевой функции вместо одного, как мы делали это до сих пор (рис. 7.2). Требование одновременного выполнения нескольких запросов означает, что нам нужны новые политики, чтобы оценить полезность входных локаций поиска. В следующем разделе мы поговорим подробнее, почему изученные политики не могут быть легко распространены на пакетный вариант. Другой компонент цикла БО (ГП) остается неизменным, поскольку нам все еще нужна модель машинного обучения, которая дает вероятностные прогнозы. Для использования пакетного режима необходимо изменить лишь компонент принятия решений.

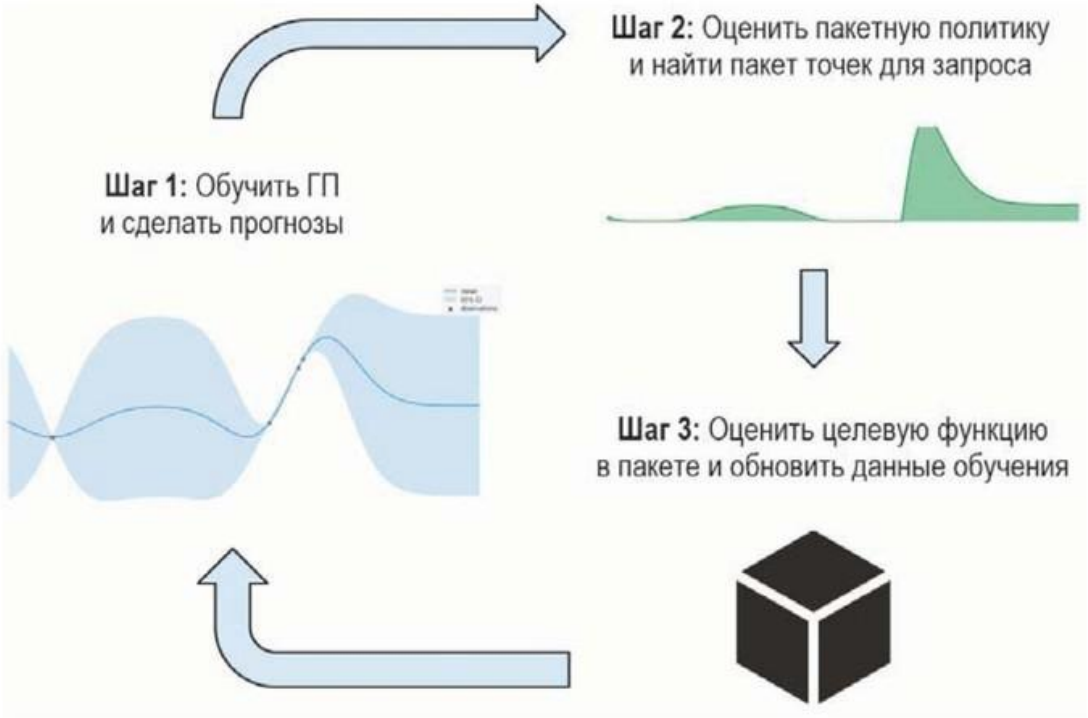


Рис. 7.2. Пакетный цикл БО: по сравнению с последовательным, пакетный вариант требует идентификации нескольких точек запроса на шаге 2 и одновременно оценивает целевую функцию в этих точках на шаге 3

Оценки приобретения политики

Политика каждой входной точке наблюдения в пространстве поиска присваивает оценку приобретения, которая количественно определяет, насколько полезны входные данные при поиске глобального оптимума. Каждая политика использует свою эвристику для расчета этого показателя (подробно описано в главах 4–6).

Количество запросов, которые могут быть выполнены одновременно (размер пакета), зависит от приложения. Например, сколько печенья можно испечь одновременно, зависит от размера духовки и противней. Доступные вычислительные ресурсы (количество вычислительных и графических процессоров) определяют, сколько нейросетей можно обучать параллельно при настройке гиперпараметров

модели. На рис. 7.1 в качестве примера показаны три процесса, выполняемых одновременно, поэтому размер пакета равен 3.

7.1.2. Почему нельзя использовать обычные политики БО в пакетном режиме?

Ранее мы обсуждали, что последовательные политики (когда запросы к целевой функции выполняются один за другим) не могут быть перепрофилированы и использованы в пакетном режиме без изменений. В этом разделе мы более подробно обсудим, почему это так и почему для этого нужны специальные политики.

Если вспомнить раздел 4.1, там политика присваивает оценку каждой точке в пространстве поиска, которая количественно определяет, насколько полезна та при нахождении глобального оптимума. Затем точка с наивысшей оценкой выбирается в качестве следующего запроса. На рис. 7.3 показана оценка, рассчитанная с помощью политики *ожидаемого улучшения* (раздел 4.3) в виде кривой на нижней части, где точка 1,75 (вертикальная отметка на кривой в нижней части рисунка) максимизирует ее и является следующим запросом.

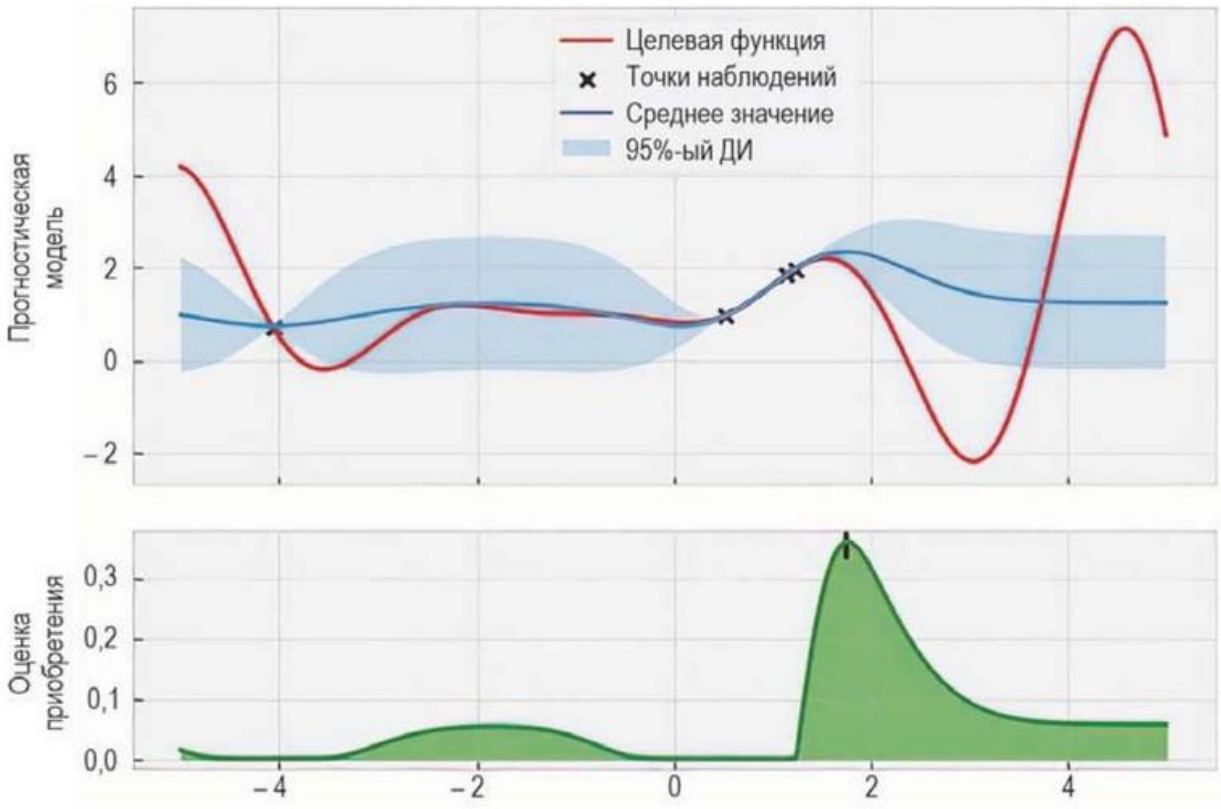


Рис. 7.3. Пример БО: на верхней части показаны прогнозы ГП и основная истинная целевая функция, а на нижней — оценки приобретения, полученные EI; вертикальная отметка на нижней кривой в области 1,75 указывает на точку, которая будет запрошена следующей

Что произойдет, если использовать одну и ту же оценку EI (нижняя кривая на рис. 7.3), чтобы выбрать сразу несколько точек для запроса целевой функции? Нужно будет определить множество точек, которые дают самые высокие оценки. Однако они просто группируются вокруг той, что выбрана при последовательном

режиме. Это связано с тем, что если переместиться на бесконечно малое расстояние от 1,75, то оценка EI все равно будет высокой. Точки, близкие к той, которая дает наивысшую оценку приобретения, также дают высокие оценки.

Если просто выбрать точки с наивысшей оценкой приобретения, запросы сгруппируются вокруг одного региона в пространстве поиска. Это все равно что сложить все яйца в одну корзину. На рис. 7.4 показаны запросы, дающие самые высокие оценки EI. Они группируются около 1,75. Такой эффект кластеризации нежелателен, поскольку он тратит ценные ресурсы на оценку целевой функции, по сути, в одном месте. Такие сгруппированные точки менее полезны, чем более разбросанные.

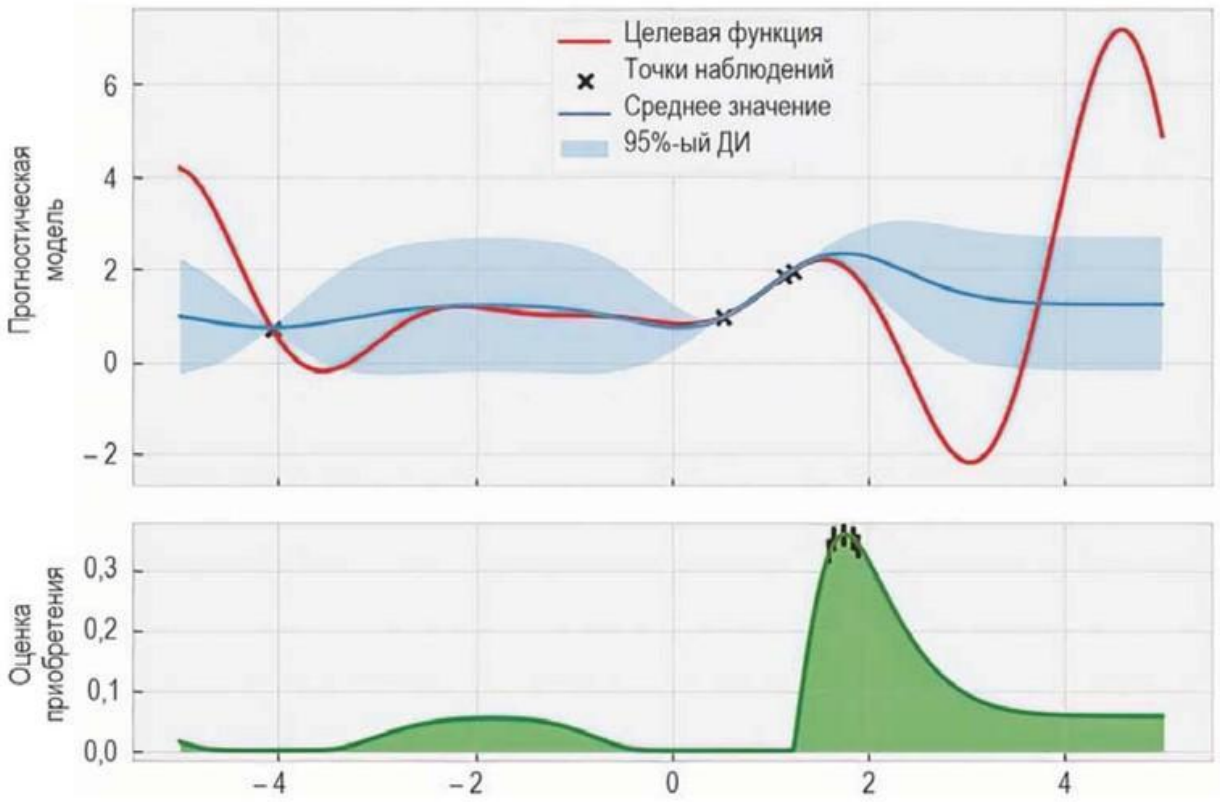


Рис. 7.4. Запросы, сделанные в пакетном режиме: мы просто выбираем точки с наивысшими оценками приобретения (обозначены вертикальными отметками на нижней кривой); эти запросы расположены близко друг к другу и менее полезны, чем если бы они были разбросаны

Когда все запросы сгруппированы в одном месте, это не дает нам возможности воспользоваться преимуществами параллелизма, присущими пакетной настройке. Пока что мы убедились лишь в том, что разработка пакета запросов в рамках политики не так проста, как выбор точек с самыми высокими оценками приобретения. В оставшейся части этой главы мы обсудим политики, которые специально созданы для работы в пакетном режиме. Они удобны в том плане, что являются расширением политик, которые были рассмотрены в главах 4–6, поэтому их изучение не будет сложным.

7.2. Вычисление улучшения и верхней доверительной границы для набора точек

Сначала мы попробуем расширить политики, основанные на улучшениях (глава 4), а также политику UCB (раздел 5.2). Эвристика, которую они все используют, позволяет модифицировать их для работы в пакетном режиме. Скоро мы сможем в этом убедиться.

Рассмотрим математические модификации их формулировок и обсудим, как работают получившиеся в результате пакетные политики. А также научимся объявлять и запускать их с помощью BoTorch.

7.2.1. Расширение эвристик оптимизации для пакетного режима

Как обсуждалось в разделе 7.1.2, выбор набора точек для оценки целевой функции не так прост, как поиск точек, максимизирующих оценку приобретения в последовательной политике. Вместо этого необходимо переопределить их математические формулировки, чтобы перепрофилировать для пакетного режима.

Существует одна стратегия, которая может быть применима к трем политикам, которые мы будем расширять (PoI, EI и UCB). Стоит помнить, что они формируют свои оценки приобретения как средние значения по нормальному распределению. То есть оценка, которую каждая из них присваивает определенной точке, может быть записана как среднее значение интересующей величины по нормальному распределению в последовательном режиме. Для PoI эта величина определяет, будет ли улучшение, а для EI — насколько сильное улучшение ожидается.

Как показано в верхней части рис. 7.5, последовательная политика оценивает потенциальный запрос x (запрос-кандидат), используя среднее значение некоторой величины $G(f(x))$ по нормальному распределению, которое является нашим предположением о значении целевой функции $f(x)$. Величина G зависит от эвристики, которую политика использует для балансировки между исследованием и эксплуатацией. Вместо этого с помощью пакета запросов $x_1, x_2, ..., x_k$ мы вычисляем среднее значение от максимума G по точкам из пакета (набора), как показано в нижней части рис. 7.5. Это среднее значение рассчитывается по многомерному распределению Гаусса, соответствующему целевым значениям $f(x_1), f(x_2), ..., f(x_k)$.

Баланс между исследованием и эксплуатацией

Все политики БО должны учитывать компромисс между концентрацией внимания на высокопроизводительном регионе в пространстве поиска (эксплуатация) и проверкой неисследованных регионов (исследование). Более подробно этот компромисс можно изучить в разделе 4.1.2.

Такая стратегия использования максимума интересующей величины G для представления полезности всего пакета имеет интуитивный смысл в контексте оптимизации, которая и является нашей целью. Чем выше максимальное значение G, тем ценнее весь пакет запросов. Теперь, имея возможность количественно измерить его ценность, мы можем приступить к поиску пакета, максимизирующего эту величи-

ну. Эвристика, которую мы используем, похожа на то, что мы делаем на спортивных соревнованиях, вроде Олимпийских игр: каждая страна может тренировать множество спортсменов в течение года, но, когда приходит время, для участия в соревнованиях выбираются только лучшие из них. Рис. 7.6 визуализирует этот процесс.

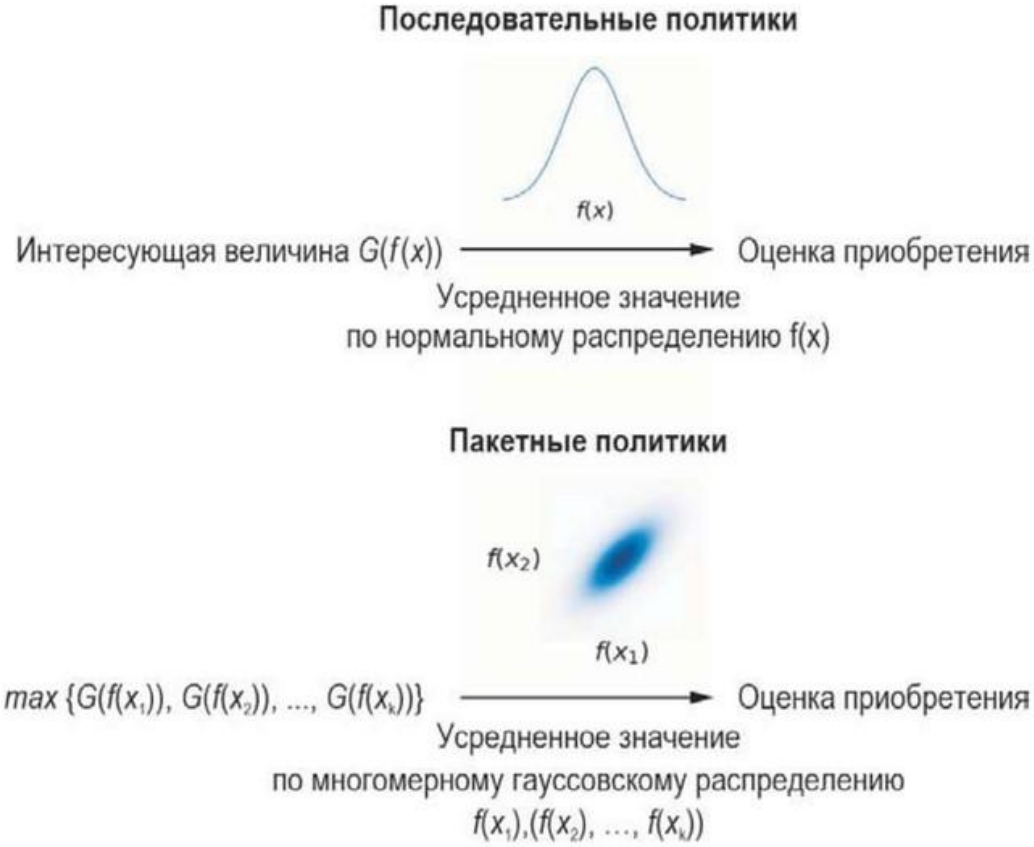


Рис. 7.5. Распространение математической формулировки политики на пакетный вариант: в обоих случаях используется среднее значение интересующей величины; мы берем максимальное значение по точкам в пакете, а затем находим среднее значение, представляющее полезность всего пакета

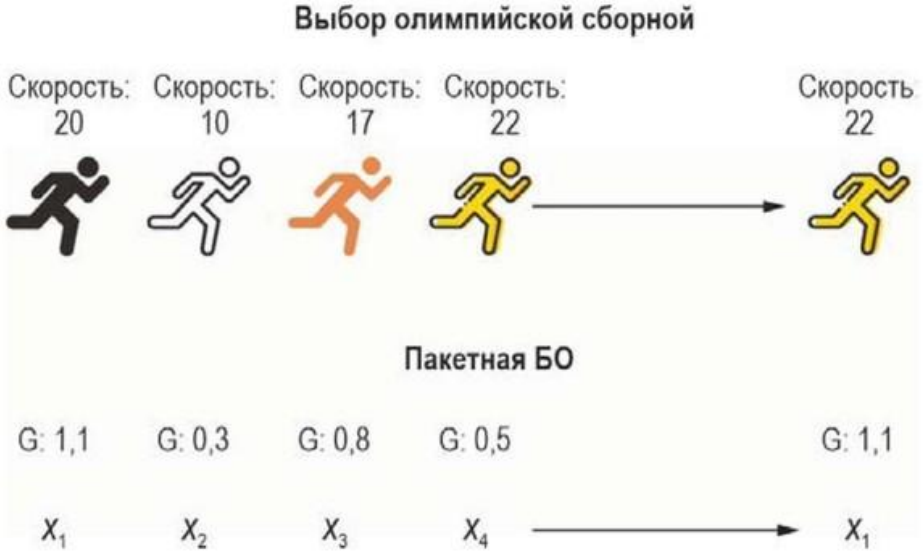


Рис. 7.6. Пакетная эвристика БО выбирает лучший элемент с наибольшим значением *G*, чтобы представить весь пакет (внизу): эта стратегия аналогична отбору команд на Олимпийских играх, где остаются только лучшие спортсмены, которые будут представлять страну

Как данная стратегия поможет реализовать три вышеупомянутые политики? Чтобы ответить, сначала обсудим первые две, основанные на улучшениях. Если вспомнить главу 4, PoI использует вероятность, что следующий запрос улучшится по сравнению с наиболее заметной точкой (*действующей*) в качестве оценки приобретения. Чем больше вероятность, что точка даст лучший результат, тем выше оценка, которую PoI присваивает ей. Политика EI, с другой стороны, учитывает масштабы улучшения, присваивая высокую оценку приобретения тем точкам, которые не только будут иметь улучшения по сравнению с действующей, но и будут иметь значительный размер этого улучшения.

Разница между двумя политиками визуализирована на рис. 7.7, где разные результаты лежат на оси X, а ось Y показывает целевое значение, которое необходимо оптимизировать. PoI одинаково обрабатывает все точки, которые дают более высокие значения, чем действующая, в то время как EI учитывает, насколько улучшится каждая из них.

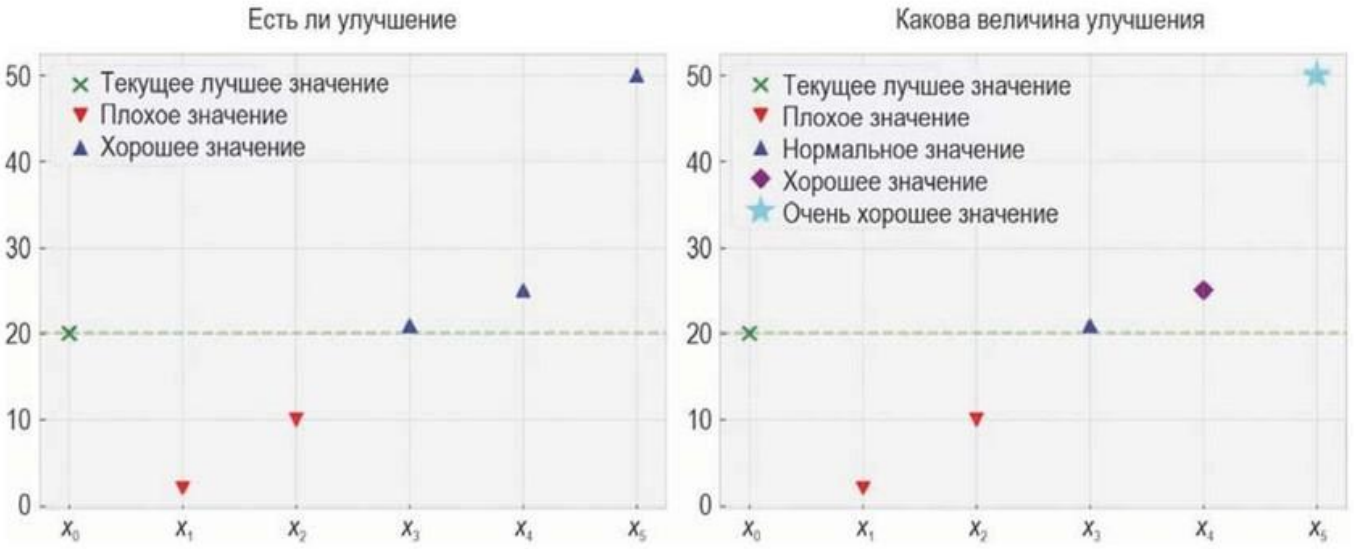


Рис. 7.7. Разница между PoI (слева) и EI (справа): первая политика учитывает только, есть ли улучшения по сравнению с действующей точкой или нет, а вторая — какова величина этих улучшений

В пакетном режиме можно аналогичным образом рассуждать об улучшении, которое наблюдается после текущей итерации цикла БО. Вместо рассуждений о множестве точек в пакете запросов мы можем выделить максимальное значение среди оценок функции в них. То есть, если пакет запросов к целевой функции находится в точках x_1, x_2, ..., x_i, нет необходимости использовать все оценки функции $f(x_1)$, $f(x_2)$, ..., $f(x_k)$ для рассуждений об улучшении. Понадобится только максимальное значение $max\{f(x_1), f(x_2), ..., f(x_k)\}$, поскольку оно определяет наблюдаемое улучшение.

Следуя примеру на рис. 7.7, предположим, что целевое значение нашей действующей точки равно 20, и рассмотрим сценарии, представленные на правой панели рис. 7.8:

- Если бы пакет запросов с размером 3 возвращал значения ниже 20 (что соответствует x_1 на правой части рисунка), то улучшения были бы незаметны. Наивыс-

шая оценка функции в точке x_1 равна 3. Это означает, что ни одна оценка функции из пакета не улучшилась по сравнению с действующей.

- Если бы все возвращенные значения превышали действующее (что соответствует x_2), то можно было бы наблюдать улучшение по сравнению с действующей точкой. В частности, максимум пакета (x_2) равен 30, что приводит к улучшению на 10.
- Что еще более важно, если только некоторые (но не все) возвращаемые оценки функций были бы лучше, чем действующая точка (например, x_3), то все равно можно было бы наблюдать улучшение. Максимум x_3 составляет 22, что действительно является улучшением по сравнению с действующим значением (20).

Сосредоточив внимание на максимальном значении, которое вернул пакет запросов, можем сразу определить, привел ли он к улучшению по сравнению с действующим значением. На рис. 7.8 показано обоснование политики PoI, основанное на улучшении, где пакеты X_2 и X_3 рассматриваются одинаково, поскольку они (или, точнее, их максимальные значения) оба приводят к улучшению. Теперь у нас есть способ расширить расчет вероятности улучшения от последовательного подхода к пакетному.

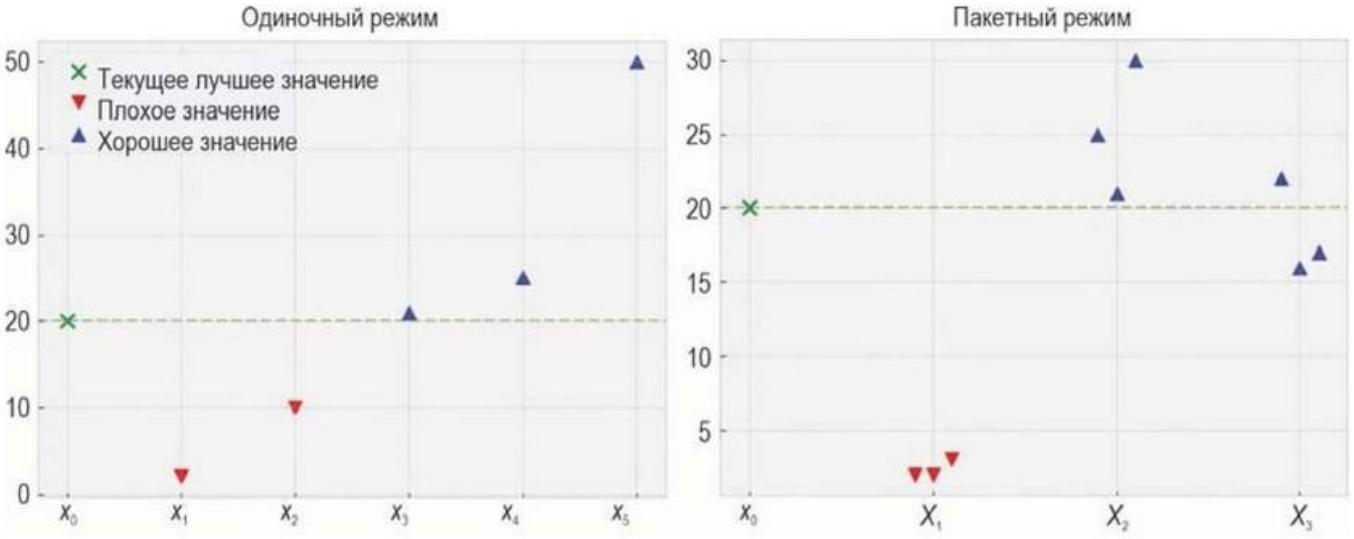


Рис. 7.8. Иллюстрация, приводит ли один запрос (слева) или пакет запросов (справа) к улучшению по сравнению с действующим значением: в пакетной настройке учитывается только максимальное значение в каждом пакете, чтобы определить, есть ли улучшение

Определение

Оценка приобретения, которую PoI присваивает определенному пакету запросов-кандидатов, равна вероятности, что максимум среди возвращаемых оценок функции превысит действующую точку.

Математически мы переходим от вычисления вероятности, что оценка функции $f(x)$ превысит действующую точку f^*, обозначаемую как $Pr(f(x) > f^*)$, к вычислению вероятности, что максимальная оценка функции превысит действующую точку $Pr(max\{f(x_1), f(x_2), ..., f(x_k)\} > f^*)$. Эта вероятность затем будет использоваться в качестве оценки приобретения PoI для пакета запросов $x_1, x_2, ..., x_k$.

Как упоминалось ранее, вероятности $Pr(f(x) > f^*)$ и $Pr(max\{f(x_1), f(x_2), ..., f(x_k)\} > f^*)$ можно рассмотреть как средние значения величин, которые важны для оптимизации по распределениям Гаусса. В частности, вероятности представляют собой средние значения двоичной случайной величины, которые указывают, верны ли $f(x) > f^*$ и $max\{f(1), f(x_2),..., f(x_k)\} > f^*$ соответственно. Это сравнение показано на рис. 7.9.

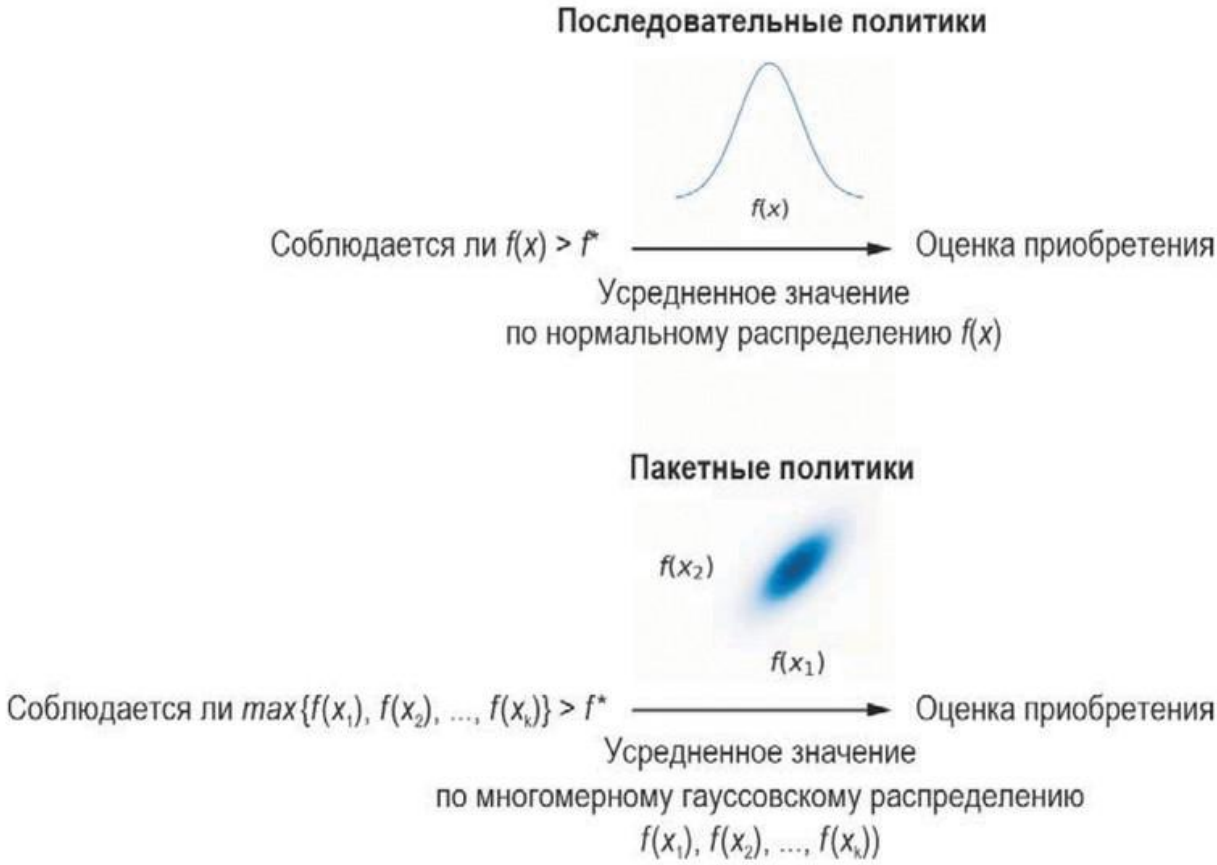


Рис. 7.9. Расширение политики POI для пакетного режима: в последовательном случае (вверху) мы рассматриваем, улучшается ли следующий запрос по сравнению с действующей точкой; в пакетном варианте (внизу) мы рассуждаем, улучшается ли максимальное значение по точкам из набора по сравнению с действующей точкой

Чтобы завершить пакетный цикл БО с политикой PoI, найдем пакет x_1, x_2, ..., x_k, который максимизирует оценку приобретения $Pr(max\{f(x_1), f(x_2), ..., f(x_k)\} > f^*)$. Как известно из раздела 4.1.1, мы можем использовать вспомогательную функцию `optimize_acqf()` из модуля `optim.optimize` в BoTorch, чтобы облегчить поиск пакета x_1, x_2, ..., x_k, который оптимизирует оценку приобретения (что мы увидим далее в разделе 7.2.2).

Теперь можно перейти к политике EI, которая вычисляет ожидаемую ценность улучшения действующего значения, которую мы будем наблюдать, запрашивая конкретную точку. Пакетное расширение EI представляет собой способ рассуждать об улучшении относительно действующей точки, наблюдая за пакетом оценок функции. То есть мы вычисляем только ожидаемое значение улучшения относительно действующей точки, которое является результатом максимальной оценки функции в возвращаемом пакете $max\{f(x_1), f(x_2), ..., f(x_k)\}$. Точно так же, как PoI вы-

числяет вероятность, что максимальное значение превышает действующее, EI учитывает, насколько максимальное значение превышает улучшение. Разница между EI и его пакетным вариантом представлена на рис. 7.10.

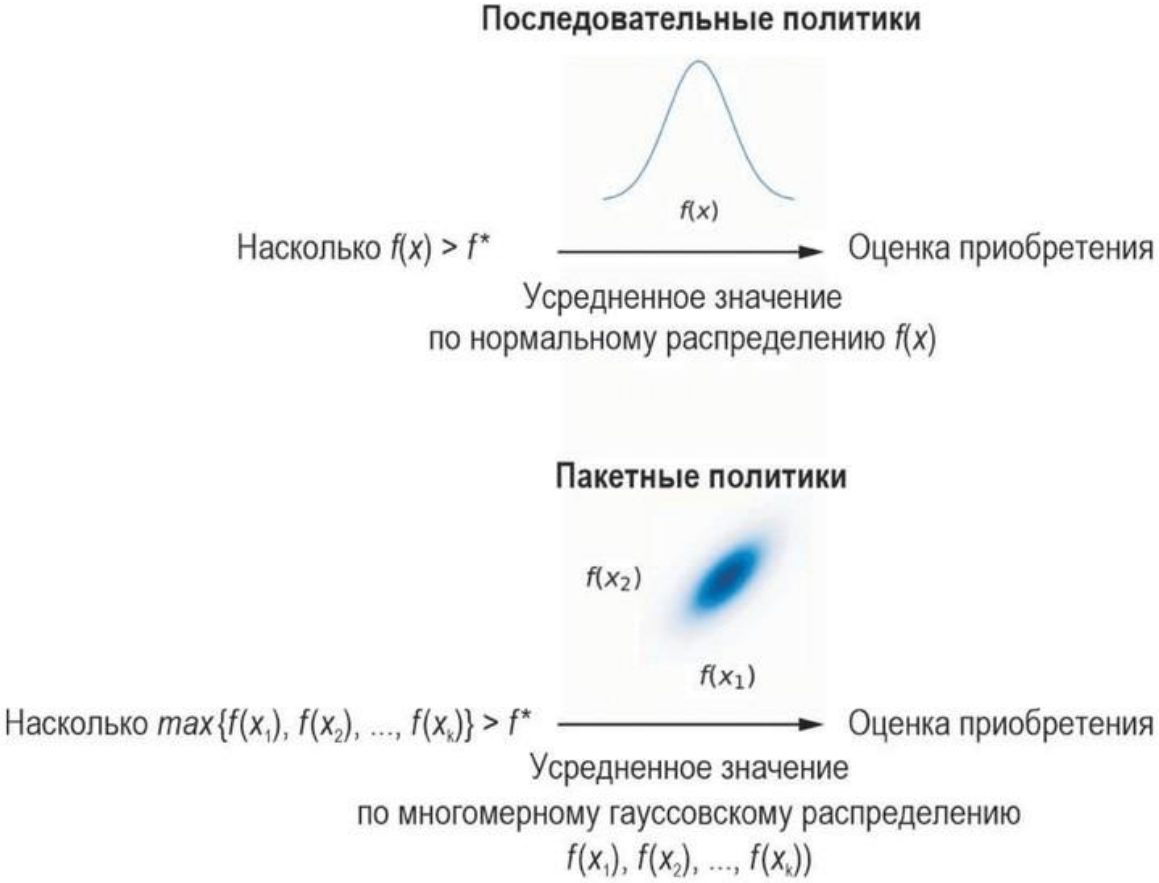


Рис. 7.10. Расширение политики EI для пакетного режима: в последовательном случае (вверху) используется среднее значение того, насколько следующий запрос улучшается по сравнению с действующим; в пакетной настройке (внизу) мы берем среднее значение того, насколько максимальное значение по всем точкам в пакете улучшается по сравнению с действующей точкой

Чтобы проиллюстрировать это рассуждение, на рис. 7.11 показано различие, как EI оценивает разные результаты в последовательном (слева) и пакетном режимах (справа). Для правой части верно следующее:

- Пакет (например, X_1), не имеющий ни одной точки, которая улучшилась бы по сравнению с действующей (имеющей значение 20), будет представлять собой нулевое улучшение.
- Максимальное значение для пакета X_2 составляет 22, поэтому мы наблюдаем улучшение на 2, даже несмотря на то, что в нем есть значения, которые ниже действующего.
- Несмотря на то что все значения в пакете X_3 выше действующего, наблюдаемое улучшение снова полностью определяется максимальным значением (30).
- Даже если большая часть пакета X_4 ниже действующего значения (20), максимальное значение X_4 равно 50, что делает это очень хорошим результатом.

Чтобы продолжить работу с пакетной EI, вычислим, насколько выше максимальное значение внутри пакета, чем у действующей точки. Данное ожидаемое улучшение

представляет собой набор оценок приобретения, который EI использует для ранжирования, насколько ценен определенный пакет x_1, x_2, ..., x_k. Вспомогательную функцию `optimize_acqf()` можно снова использовать для поиска пакета, который дает наибольшее ожидаемое улучшение.

То, что мы только что обсудили, поможет расширить PoI и EI до пакетного режима.

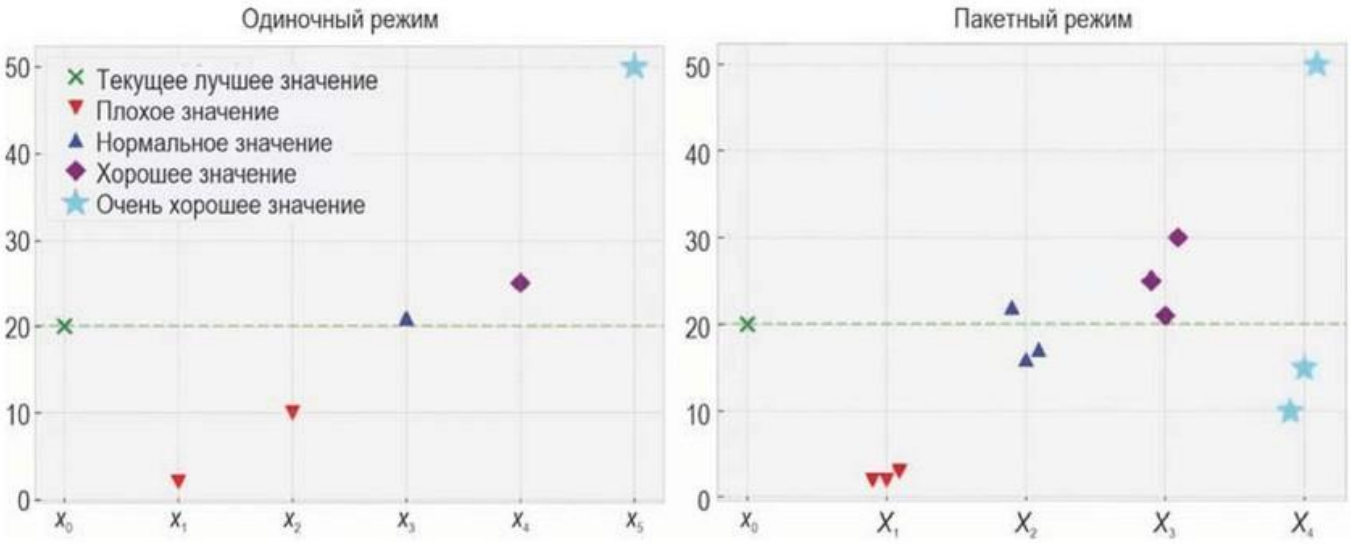


Рис. 7.11. Иллюстрация, приводит ли запрос (слева) или пакет запросов (справа) к улучшению по сравнению с действующим значением: в пакетном варианте (справа) мы учитываем только максимальное значение в каждом пакете, чтобы определить, есть ли улучшение

К счастью, стратегия выбора максимального значения из пакета запросов для расчета улучшения также применима к политике UCB. Чтобы сделать это, нужно переосмыслить оценку приобретения для нее как среднее значение по нормальному распределению.

Математические детали политики UCB

В разделе 5.2.2 мы обсуждали, что оценка приобретения UCB равна $\mu + \beta\sigma$. Здесь μ и σ представляют собой прогностическое среднее значение и стандартное отклонение $f(x)$, а β — регулируемый параметр, который помогает балансировать между исследованием и эксплуатацией. Нам нужно переписать $\mu + \beta\sigma$ как среднее значение некоторой величины по нормальному распределению $N(\mu, \sigma^2)$, чтобы расширить UCB до пакетного режима. Несмотря на то что можно изменить данную формулировку, мы не будем здесь вдаваться в математические подробности. Вы можете обратиться к приложению А данной книги, в котором излагаются математические детали.

Остальная часть расширения UCB выполняется по той же процедуре:

1. Мы берем среднее значение максимума величины, которое представляет собой переписанное $\mu + \beta\sigma$ для всего пакета, и используем его в качестве оценки приобретения пакетной UCB.
2. Затем мы используем вспомогательную функцию `optimize_acqf()`, чтобы найти пакет, который дал наивысшую оценку.

Это все, что нам нужно знать о расширении этих трех политик до пакетного режима. Далее мы узнаем, как реализовать их в BoTorch.

7.2.2. Реализация пакетных политик PoI, EI и UCB

BoTorch упрощает реализацию и использование политик БО в Python. Пакетные варианты, которые мы обсудили ранее, не являются исключениями. Несмотря на то что важно знать их математические формулировки, с помощью BoTorch можно просто заменить одну строку в программе для полноценного запуска. Код, который используется в этом разделе, можно найти в `CH07/01 – Batch BOloop.ipynb`.

Поскольку сейчас мы работаем в новых условиях, когда запросы к целевой функции выполняются в пакетном режиме, можно подумать, что код для цикла БО должен быть сильно изменен (одновременное получение нескольких оценок функции, добавление множества точек к обучающему набору, обучение модели ГП). Удивительно, но необходимые изменения минимальны, благодаря способности BoTorch беспрепятственно поддерживать пакетный режим. При использовании вспомогательной функции `optimize_acqf()` для поиска следующих запросов, которые максимизируют оценку приобретения, нужно всего лишь указать параметр $q = k$, где k — размер пакета (количество оценок функции, которые могут быть выполнены параллельно).

Весь пакетный цикл БО представлен на рис. 7.12, который очень похож на рис. 4.4. Некоторые изменения стоит прокомментировать:

- При использовании вспомогательной функции `optimize_acqf()`мы указываем размер пакета $q = k$.
- `optimize_acqf()` возвращает `next_x`, которая содержит `k` точек. Переменная `next_x` — это тензор PyTorch размером `k` на `d`, где `d` — количество измерений в пространстве поиска (количество объектов в наборе данных).
- Затем мы запрашиваем целевую функцию в местах, указанных в `next_x`, и получаем `next_y`, которая содержит оценки функции. В отличие от последовательной оптимизации, где `next_y` является скалярным или одноэлементным тензором, здесь `next_y` — это тензор, который содержит `k` элементов и соответствует оценкам функции в `next_x`.

> **Примечание**
>
> Для шага 1 на рис. 7.12 все еще требуется реализация класса модели и вспомогательная функция `fit_gp_model()`, которая обучает ГП на данных. К счастью, тот же код, который используется в последовательном режиме, можно применить повторно без каких-либо изменений. За подробностями реализации данного кода можно обратиться к разделу 4.1.1.

Чтобы упростить демонстрацию кода, мы используем двумерную целевую функцию, которая имитирует точность модели. Она впервые была представлена в упражнении к главе 3 и реализована следующим образом: мы указываем, что область определения функции (пространство поиска) находится между 0 и 2 в каждом из двух измерений.

```
#Определение функции
def f(x):
```

```
    return (
        torch.sin(5 * x[..., 0] / 2 - 2.5) * torch.cos(2.5 - 5 * x[..., 1])
        + (5 * x[..., 1] / 2 + 0.5) ** 2 / 10
    ) / 5 + 0.2

#Область определения функции от 0 до 2 в каждом измерении.
lb = 0
ub = 2
bounds = torch.tensor([[lb, lb], [ub, ub]], dtype=torch.float)
```

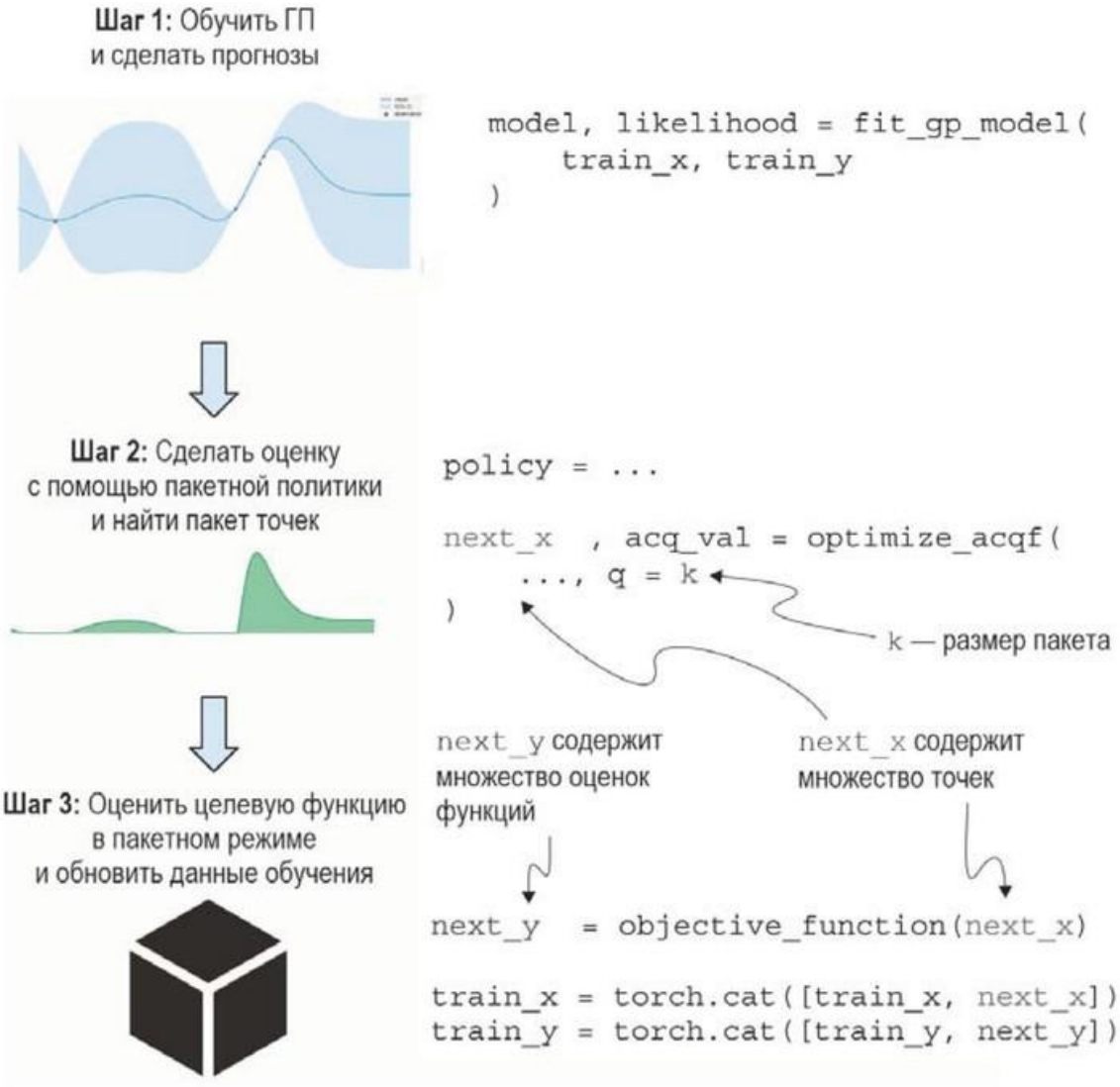


Рис. 7.12. Шаги пакетного цикла БО и соответствующий код:
по сравнению с последовательным режимом изменения в коде минимальны

Эта целевая функция представлена на рис. 7.13, где видно, что глобальный оптимум достигается в правом верхнем углу пространства. Это дает точность 90%.

Чтобы реализовать задачу пакетной оптимизации, предположим, что мы можем обучать модель четырем различным процессам одновременно. Другими словами, размер пакета равен 4. Кроме того, мы можем переобучить модель только пять раз, поэтому количество итераций пакетного цикла БО равно 5, а общее количество запросов, которые можно сделать, составляет $4 \times 5 = 20$:

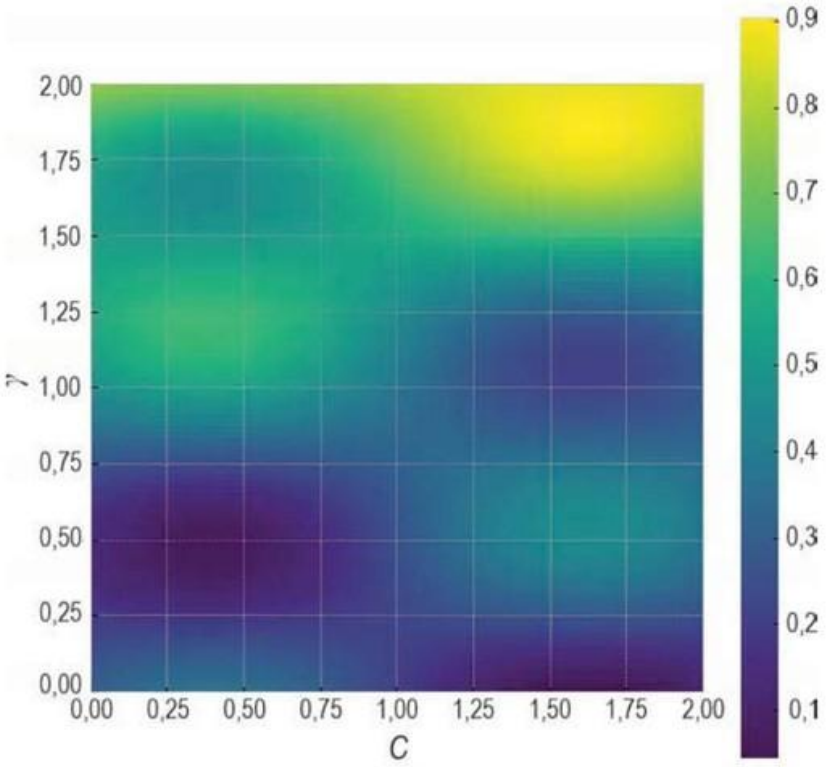


Рис. 7.13. Точность модели SVM на наборе обучающих данных как функция штрафного параметра *C* и параметра *γ* ядра RBF: эту целевую функцию мы стремимся оптимизировать в данной главе

```
num_queries = 20
batch_size = 4
num_iters = num_queries // batch_size #Эта переменная равна 5
```

Теперь все, что осталось сделать, — запустить пакетную политику. Сделать это можно с помощью кода, который сначала случайным образом выбирает точку в пространстве поиска в качестве обучающего набора:

```
torch.manual_seed(0)
#Случайным образом выбираем точку в пространстве поиска
train_x = bounds[0] + (bounds[1] - bounds[0]) * torch.rand(1, 2)
#Оцениваем целевую функцию в случайно выбранной точке
train_y = f(train_x)
```

Затем мы делаем следующее для каждой из пяти итераций:

1. Отслеживаем лучшую точность, которую когда-либо видели.
2. Переобучаем модель ГП с помощью текущего тестового набора.
3. Инициализируем пакетную политику.
4. Используем вспомогательную функцию `optimize_acqf()`, чтобы найти лучший пакет запросов.
5. Оцениваем целевую функцию в местах, указанных в пакете.
6. Добавляем новые наблюдения в обучающий набор и повторяем:

```
incumbents = torch.zeros(num_iters)
for i in tqdm(range(num_iters)):
    incumbents[i] = train_y.max() #Отслеживаем прогресс оптимизации
```

```
#Обучаем ГП на текущем обучающем наборе
model, likelihood = fit_gp_model(train_x, train_y)

#Инициализируем пакетную политику БО, которую вскоре обсудим
policy = ...

#Находим следующий пакет для запроса
next_x, acq_val = botorch.optim.optimize_acqf(
    policy,
    bounds=bounds,
    q=batch_size, #Задаем аргумент q равным размеру пакета
    num_restarts=40,
    raw_samples=100,
)

next_y = f(next_x) #Оцениваем целевую функцию в выбранном пакете

#Обновляем обучающие данные
train_x = torch.cat([train_x, next_x])
train_y = torch.cat([train_y, next_y])
```

Этот код почти идентичен тому, который используется в разделе 4.1.1. Он реализует последовательный режим байесовской оптимизации. Далее нужно установить для аргумента `q` правильный размер пакета.

Чтобы затем запустить пакетную политику, мы инициализируем ее, используя ее реализацию в классе BoTorch. Для PoI мы используем:

```
policy = botorch.acquisition.monte_carlo.qProbabilityOfImprovement(
    model, best_f=train_y.max()
)
```

Аналогично для EI мы используем:

```
policy = botorch.acquisition.monte_carlo.qExpectedImprovement(
    model, best_f=train_y.max()
)
```

Обратите внимание на литеру q перед именами классов. Она указывает на то, что они реализуют пакетные варианты политик БО. Для последовательных политик PoI и EI аргумент `best_f` указывает текущее действующее значение, которое мы устанавливаем в `train_y.max()`.

Для UCB мы используем эквивалентный API, где аргумент `beta` устанавливает компромиссный параметр β в формуле оценки приобретения $\mu + \beta\sigma$, где, как мы помним, μ и σ — среднее значение и стандартное отклонение прогноза в определенной точке соответственно:

```
policy = botorch.acquisition.monte_carlo.qUpperConfidenceBound(
    model, beta=2
)
```

Аргументы, принимаемые политиками

О реализации последовательных режимов для PoI, EI и UCB мы узнали в разделах 4.2.2, 4.3 и 5.2.3 соответственно. Аргументы, которые принимает каждая из них, идентичны аргументам их пакетных аналогов, что упрощает переход к пакетному режиму в BoTorch.

Поскольку теперь мы можем запустить пакетные версии PoI, EI и UCB, уделим немного времени и проверим поведение этих политик. В частности, предположим, что текущий прогресс в БО аналогичен тому, что изображен на рис. 7.3 (с одномерной целевой функцией). На нем также показаны оценки приобретения, рассчитанные EI для одной точки данных (внизу). Также интересно посмотреть, как выглядят оценки приобретения EI для группы из двух точек (ожидаемого улучшения по сравнению с действующей точкой по данным из двух запросов).

Эти оценки приобретения показаны с помощью тепловой карты на рис. 7.14, где яркость каждого местоположения на квадрате обозначает ожидаемое улучшение определенной пары запросов. Местоположения, которые дают самые высокие оценки, обозначены звездочками. Верхняя и правая части показывают наблюдаемые данные и текущее предположение ГП о целевой функции по осям тепловой карты. Можно увидеть несколько интересных тенденций:

- На тепловой карте есть две прямые полосы, обозначающие высокие оценки приобретения. Они близки к точке данных $x = 2$. Любой пакет запросов (размером 2), в котором есть элемент, близкий к $x = 2$, даст высокий балл. Это имеет смысл, поскольку в районе $x = 2$ апостериорный средний ГП максимален.

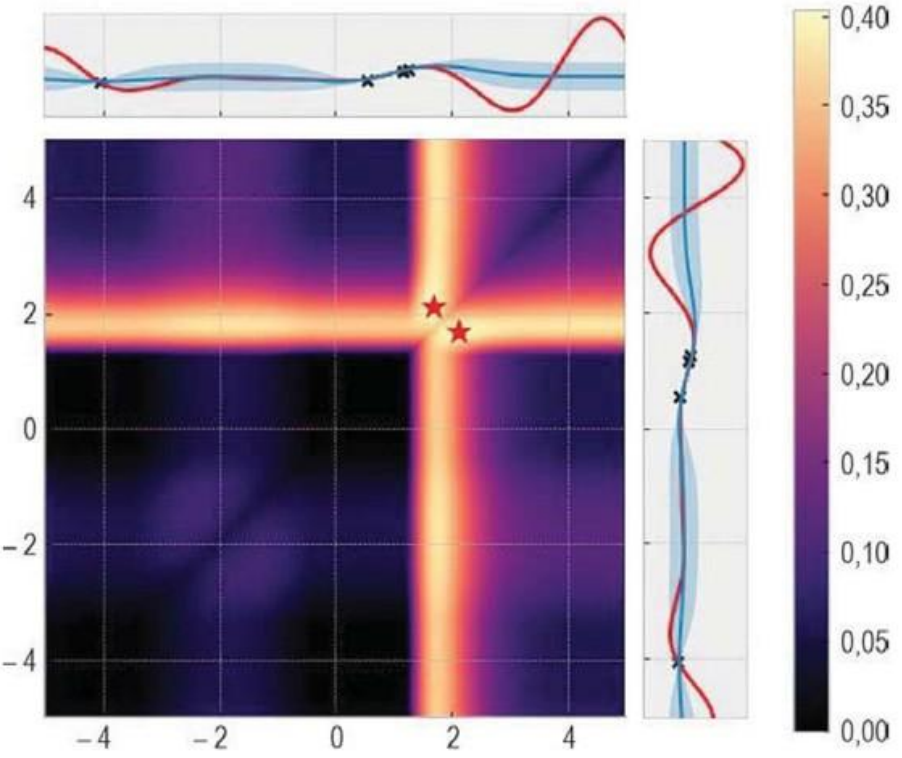


Рис. 7.14. Тепловая карта, показывающая оценки приобретения пакетной политики EI для пакетов размером 2 с одномерной целевой функцией: верхняя и правая части рисунка показывают наблюдаемые данные и текущее предположение ГП о целевой функции по осям тепловой карты; две оптимальные пары запросов, обозначенные звездочками, содержат значения 1,68 и 2,12, которые находятся относительно далеко друг от друга

- Диагональ тепловой карты затемнена. Это означает, что запрос пакетов X_1 и X_2, где X_1 примерно равен X_2, скорее всего, даст небольшое улучшение. Это наблюдение подтверждает сказанное в разделе 7.1.2: группировать запросы в пакете рядом друг с другом — это плохая стратегия, в которой вы, по сути, «складываете все яйца в одну корзину».
- Два оптимальных пакета запросов, отмеченные звездочками, представляют собой один и тот же пакет, поскольку местоположения симметричны друг другу. Он содержит значения 1,68 и 2,12, которые все еще близки к $x = 2$, где ГП предполагает, что целевая функция дает высокие значения. Кроме того, два выбранных запроса 1.68 и 2.12 находятся далеко друг от друга и, таким образом, помогают избежать ловушки кластеризации запросов рядом друг с другом.

На рис. 7.14 показано, что EI разумным образом оценивает заданный пакет запросов, отдавая приоритет тем из них, которые с большей вероятностью дадут высокие целевые значения и будут достаточно распределены.

Пакетная EI против последовательной EI

Две точки пакетной EI (1,68 и 2,12) отличаются от той, что максимизирует оценку приобретения для последовательной EI (1,75). Эта разница показывает, что оптимальное решение при последовательном режиме не обязательно является таковым при пакетном.

Возвращаясь к примеру настройки гиперпараметров, мы готовы использовать эти инициализации для запуска пакетных политик. После сохранения текущих действующих значений, достигнутых каждой политикой, и сопоставления их друг с другом, визуализируем ход оптимизации (рис. 7.15) каждой политики. Сначала можно заметить, что прогресс отображается пакетами по четыре штуки, что имеет смысл, поскольку 4 — это используемый размер пакета. Что касается производительности, то видно, что EI и UCB вначале могут развиваться быстрее, чем PoI, но в конце все три сходятся примерно с одинаковой точностью.

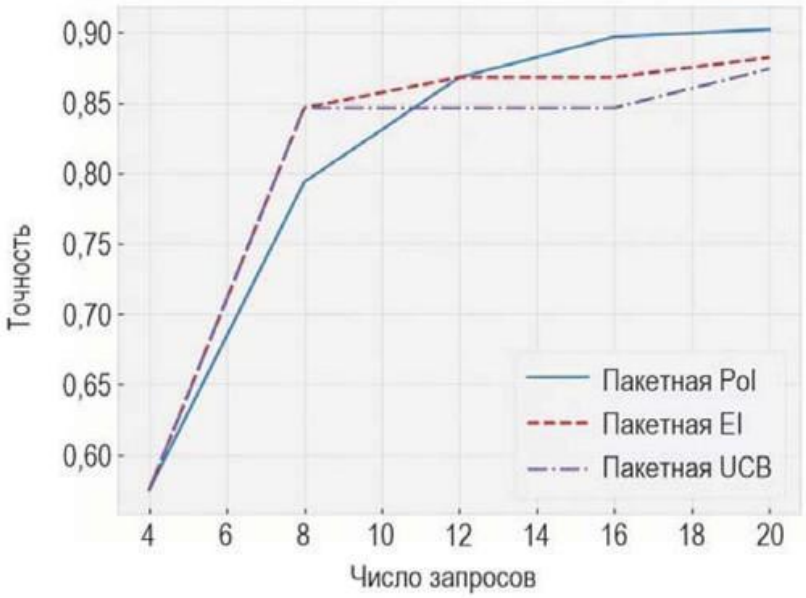


Рис. 7.15. Прогресс, достигнутый различными пакетными политиками БО для настройки гиперпараметров: прогресс выполняется пакетами по четыре, что соответствует используемому размеру

Повторные эксперименты в БО

Более точно сравнить производительность данных политик можно, повторив эксперимент, используя разные начальные обучающие наборы, которые генерируются случайным образом. Чтобы узнать, как можно проводить повторные эксперименты в БО, вы можете обратиться к шагу 9 упражнения 2 к главе 4.

Мы узнали, как реализовать пакетные версии политик PoI, EI и UCB в BoTorch, и увидели, что переход от последовательного режима к пакетному требует минимальных изменений в коде. Теперь перейдем к остальным политикам (TS и MES), которые требуют иных стратегий при расширении для пакетной оптимизации.

7.3. Упражнение 1: Расширение TS до пакетной оптимизации с помощью повторной выборки

В отличие от других политик, выборка Томпсона (TS) может быть легко расширена до пакетного режима благодаря ее стратегии выборки. В этом упражнении мы рассмотрим, как это можно реализовать. Как нам уже известно (раздел 5.3), TS при последовательном выполнении извлекает одну выборку из текущего предположения ГП о целевой функции и запрашивает точку данных, которая эту выборку максимизирует.

При пакетном выполнении мы просто повторяем процесс выборки из ГП и ее максимизацию несколько раз, чтобы собрать пакет запросов желаемого размера. Например, если он равен 3, то мы извлекаем три выборки из ГП. Пакет, который мы получаем, содержит 3 точки (3 *максимизатора*), которые максимизируют три выборки (по одной для каждой). Эта логика проиллюстрирована на рис. 7.16. Мы

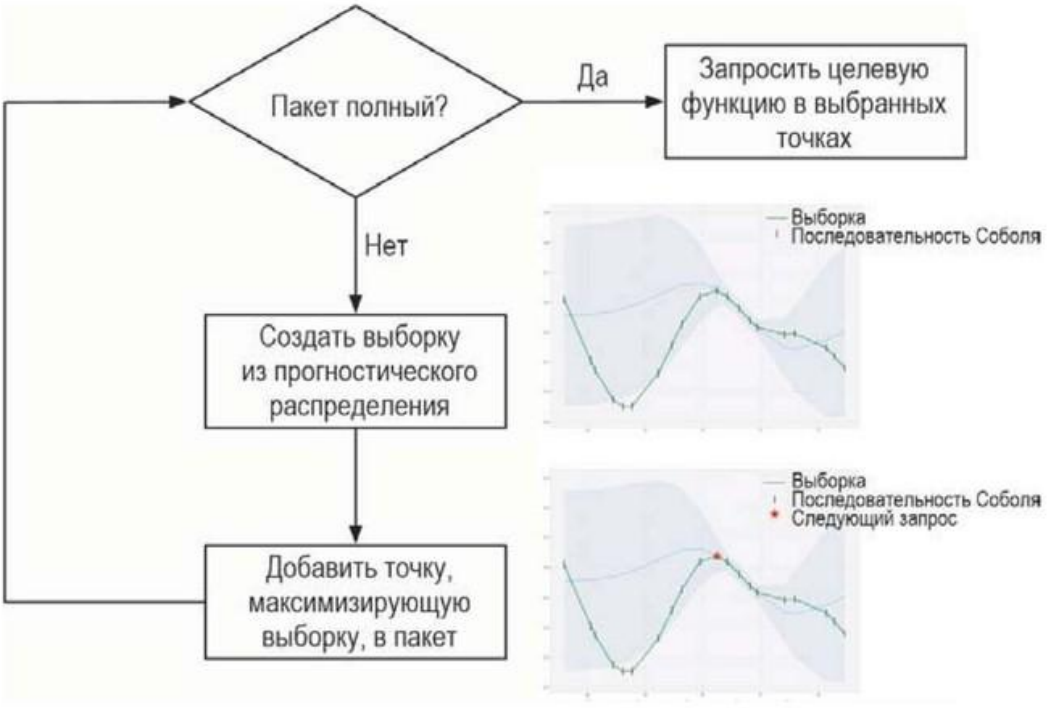


Рис. 7.16. Блок-схема реализации пакетной версии TS: мы делаем выборки из ГП и добавляем точку, которая максимизирует выборку, к текущему пакету, пока он не заполнится

делаем выборки из ГП и добавляем точки к текущему пакету до тех пор, пока он не заполнится (пока не будет достигнут соответствующий размера пакета).

При каждой выборке мы имеем разную реализацию того, какой может быть целевая функция. Оптимизируя несколько выборок, мы простым способом получаем несколько точек, которые могут привести к глобальному оптимуму целевой функции. Чтобы реализовать и запустить эту политику для настройки гиперпараметров, необходимо предпринять следующие шаги (`CH07/02 - Exercise 1.ipynb`):

1. Воссоздайте пакетный цикл БО из `CH07/01 - Batch BayesOpt loop.ipynb`.
2. Реализуйте TS, используя последовательность Соболя (как описано в разделе 5.3):
   - Необходимо использовать 2000 кандидатов для последовательности Соболя.
   - При вызове объекта TS нужно указать, что количество выборок равно размеру пакета:

```
ts = botorch.generation.MaxPosteriorSampling(model, replacement=False)
next_x = ts(candidate_x, num_samples=batch_size)
```

3. Запустите политику TS для целевой функции и понаблюдайте за ее производительностью.

Последовательность Соболя

Последовательность Соболя может покрывать пространство лучше, чем последовательность с равномерной выборкой. Более подробную информацию об этом можно найти в разделе 5.3.2.

7.4. Вычисление ценности пакета точек с помощью теории информации

Далее мы узнаем, как создать пакетный вариант еще одной политики в нашем наборе инструментов — *энтропийного поиска по максимальному значению* (MES). В отличие от PoI, EI и UCB, для MES требует более тщательное рассмотрение для эффективной работы. В следующем разделе мы обсудим пакетную версию MES и проблемы, с которыми мы можем столкнуться при ее расширении до пакетной оптимизации, а также как реализовать ее в BoTorch.

> **Примечание**
>
> MES — это тема главы 6, где речь идет об основах теории информации и о том, как реализовать данную политику с помощью BoTorch.

7.4.1. Нахождение наиболее информативного пакета точек с помощью цикличного уточнения

В последовательном режиме MES оценивает каждый запрос-кандидат в соответствии с тем, сколько информации о максимальном значении f^* целевой функции мы

получим после запроса точки-кандидата. Чем больше информации, тем больше вероятность, что точка приведет нас к глобальному оптимуму x^* целевой функции.

Используем ту же стратегию для пакетного режима. Нам необходимо вычислить, сколько информации о максимальном целевом значении f^* мы получим после запроса пакета точек-кандидатов. Теоретико-информационная ценность набора точек представляет собой четко определенную математическую величину. Мы теоретически можем вычислить и использовать ее в качестве оценки приобретения в пакетном варианте. Однако вычисление этой величины на практике обходится довольно дорого.

Основное «узкое место» подхода связано с тем, что нам нужно рассмотреть все возможные оценки функций в пакете, чтобы знать, сколько информации о f^* мы получим. Несмотря на то что эти оценки следуют многомерному распределению Гаусса, предлагающему множество математических удобств, вычисление прироста информации об f^* является одной из задач, которые не упрощаются благодаря «гауссовости». Такие вычислительные затраты означают, что мы можем вычислить оценку приобретения для набора точек, но это стоит очень дорого и не поддается оптимизации. То есть найти набор точек, который максимизирует информацию об f^*, очень сложно.

Примечание

Поиск запроса, максимизирующего оценку приобретения, выполняется с помощью L-BFGS. Это квазиньютоновский метод оптимизации (который обычно работает лучше, чем градиентный спуск) во вспомогательной функции `optimize_acqf()`. Однако из-за способа вычисления оценки приобретения в пакетной версии ни L-BFGS, ни градиентный спуск не могут эффективно ее оптимизировать.

Оценки приобретения в БО

Следует помнить, что оценка приобретения количественно определяет ценность запроса (или пакета запросов), помогая нам найти глобальный оптимум целевой функции, поэтому на каждой итерации цикла БО нам необходимо идентифицировать запрос (или пакет запросов), который максимизирует оценку приобретения. В разделе 4.1 подробно обсуждается максимизация оценки приобретения.

Точная максимизация этого показателя возможна только в последовательной ситуации. Обычно используемые алгоритмы неэффективны при поиске пакета, который максимизирует оценку.

Если метод L-BFGS, который мы обычно используем для поиска следующего оптимального запроса с точки зрения теории информации, работает только в последовательном режиме для одной точки-кандидата, как мы можем использовать его в пакетном варианте? Наша стратегия здесь состоит в том, чтобы использовать его для поиска отдельных членов пакета циклически (по одному) до тех пор, пока не произойдет сходимость. В частности, мы делаем следующее:

1. Начинаем со стартовой партии x_1, x_2, ..., x_k. Она может быть случайно выбрана из пространства поиска.
2. Поскольку L-BFGS не может работать на всех элементах пакета x_1, x_2, ..., x_k одновременно, мы запускаем его только на x_1, оставляя фиксированными

остальные элементы x_2, x_3, ..., x_k. L-BFGS действительно может оптимизировать x_1 индивидуально, поскольку эта задача аналогична максимизации оценки приобретения в последовательном режиме.

3. Как только L-BFGS возвращает значение для x_1, мы запускаем его для x_2, сохраняя при этом x_1 и другие члены x_3, x_4,..., x_k фиксированными.
4. Повторяем эти отдельные процедуры до тех пор, пока не закончим обработку последнего члена x_k, после чего возвращаемся к x_1 и повторяем всю процедуру.
5. Запускаем эти циклы оптимизации, пока не произойдет сходимость. То есть до тех пор, пока полученная оценка приобретения больше не перестанет увеличиваться. Эти шаги обобщены на рис. 7.17.

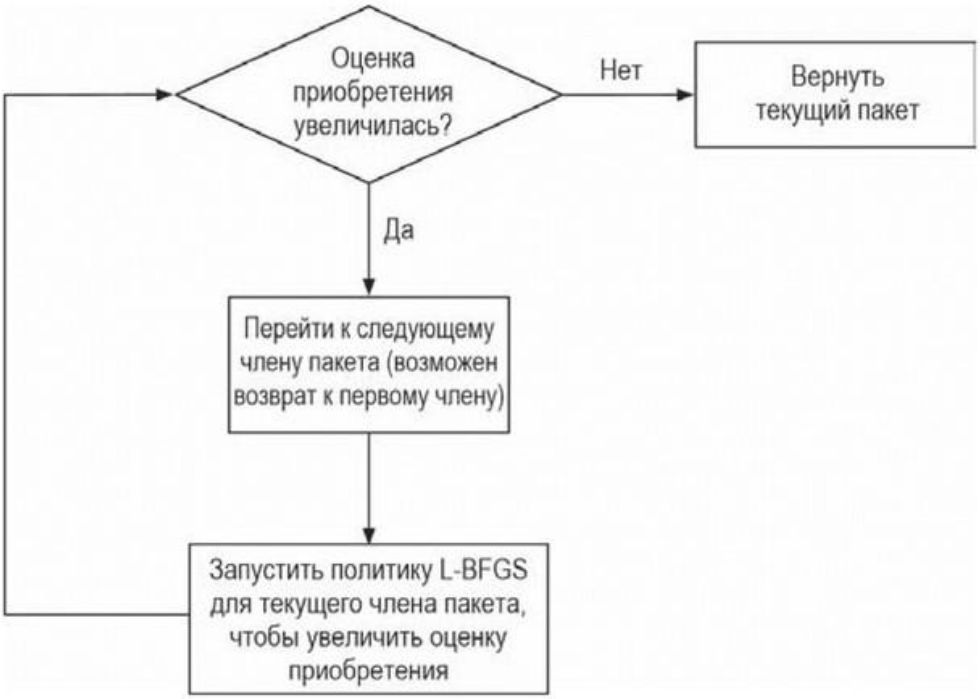


Рис. 7.17. Блок-схема циклической оптимизации, используемая при поиске пакета, который максимизирует информацию о наибольшем целевом значении в пакетной политике MES: процедура является циклической, поскольку мы последовательно уточняем каждый член пакета, пока не достигнем хорошей оценки приобретения

Определение

Вся процедура называется *циклической оптимизацией*, поскольку мы последовательно уточняем каждый член пакета в цикле, пока не достигнем хорошей оценки приобретения.

Стратегия циклической оптимизации позволяет обойти проблему запуска L-BFGS для пакета из нескольких точек, поскольку вместо этого мы запускаем алгоритм только для отдельных точек, внося индивидуальные уточнения в оценку приобретения. Благодаря этой стратегии оптимизации мы можем реализовать политику MES в пакетном режиме.

Примечание

Можно провести аналогию между циклической оптимизацией и тем, как художник рисует картину. Он может работать над отдельными ее частями и менять их по мере

продвижения. Они могут быть на переднем плане, ненадолго отходить на задний, а затем снова возвращаться на передний, каждый раз приобретая небольшие дополнения.

7.4.2. Реализация пакетного энтропийного поиска с помощью BoTorch

Теперь мы можем попробовать объявить пакетную политику MES в BoTorch и подключить ее к нашему циклу БО. К счастью, детали циклической оптимизации из предыдущего раздела абстрагированы в BoTorch, и мы можем напрямую инициализировать пакетную оптимизацию MES. Следующий код включен в `CH07/03 - Max-value Entropy Search.ipynb`.

В этом примере мы по-прежнему используем задачу настройки гиперпараметров. Во-первых, нужно внести небольшие изменения в модель ГП. В частности, чтобы можно было рассуждать об энтропии апостериорного ГП («фантазировать» о будущих наблюдениях), реализация класса модели должна наследоваться от класса `FantasizeMixin` из модуля `botorch.models.model`:

```
class GPModel(
    gpytorch.models.ExactGP,
    botorch.models.gpytorch.GPyTorchModel,
    #Наследование от FantasizeMixin позволяет более эффективно
    #рассуждать об апостериорном ГП
    botorch.models.model.FantasizeMixin
):
    _num_outputs = 1
    ... #Остальной код остается неизменным
```

Остальная часть кода с реализацией класса остается неизменной. Далее внутри `for`, реализующего цикл БО, объявляем MES так же, как и в последовательном режиме:

1. Извлекаем выборки из последовательности Соболя и используем их в качестве кандидатов для политики MES. Изначально они располагаются внутри единичного куба, а затем их размер меняется, чтобы охватить пространство поиска.
2. Политика MES инициализируется с использованием модели ГП и заранее сгенерированного набора кандидатов:

```
num_candidates = 2000

#Пространство поиска является двумерным
sobol = torch.quasirandom.SobolEngine(2, scramble=True)
candidate_x = sobol.draw(num_candidates)

#Изменяем размеры кандидатов, чтобы охватить пространство поиска
candidate_x = (bounds[1] - bounds[0]) * candidate_x + bounds[0]

policy = botorch.acquisition.max_value_entropy_search.qMaxValueEntropy(
    model, candidate_x
)
```

Последовательность Соболя

Последовательность Соболя впервые обсуждалась в разделе 5.3.2 совместно с политикой TS. Для реализации политики MES нам также необходима последовательность Соболя (раздел 6.2.2).

Несмотря на то что инициализация пакетной MES аналогична ее последовательному варианту, нам нужна вспомогательная функция, отличная от `optimize_acqf()`. Она необходима для процедуры циклической оптимизации, описанной в предыдущем разделе, чтобы идентифицировать пакет, максимизирующий апостериорную информацию об f^*.

В частности, здесь используется `optimize_acqf_cyclic()`, доступ к которой можно получить из того же модуля `botorch.optim`. Мы лишь отключаем `optimize_acqf()`, чтобы использовать `optimize_acqf_cyclic()`, остальные аргументы, вроде границ и размера пакета, остаются прежними:

```
next_x, acq_val = botorch.optim.optimize_acqf_cyclic(
    policy,
    bounds=bounds,
    q=batch_size,
    num_restarts=40,
    raw_samples=100,
)
```

Предупреждение об измерениях BoTorch

При запуске кода пакетной MES может возникнуть предупреждение:

```
BotorchTensorDimensionWarning:

Non-strict enforcement of botorch tensor conventions. Ensure that target tensors Y has an
explicit output dimension.
```

Оно указывает на то, что мы не форматируем тензор, содержащий наблюдаемые значения `train_y`, в соответствии с соглашением BoTorch. Однако это не является ошибкой. Поэтому, чтобы иметь возможность продолжать использовать ту же реализацию ГП, что и с другими политиками, мы просто проигнорируем это предупреждение с помощью модуля `warnings`.

> **Примечание**
>
> Из-за своей алгоритмической сложности выполнение пакетной политики MES может занять довольно много времени. Вы можете пропустить часть кода, запускающую цикл оптимизации, и продолжить читать эту главу.

Теперь мы можем запустить пакетный вариант MES. Его прогресс с использованием тех же исходных обучающих данных визуализирован на рис. 7.18. На нем можно увидеть, что политика сопоставима с другими политиками в этом запуске.

На данном этапе мы узнали, как преобразовать политики БО для пакетного выполнения, при котором несколько запросов выполняются параллельно. Преобразование каждой из них требует различного уровня внимания. В случае с PoI, EI и UCB используется эвристика, согласно которой наиболее эффективный член должен

представлять весь пакет. В упражнении 1 видно, что TS можно расширить до пакетного режима, просто повторяя процесс выборки, пока мы не получим пакет нужного размера. Политика MES, с другой стороны, нуждается в модификации, которая будет использовать циклическую оптимизацию для поиска пакета, максимизирующего ее оценку приобретения. В следующей главе мы познакомимся с еще одним специализированным сценарием БО, в котором необходимо учитывать ограничения при оптимизации целевой функции.

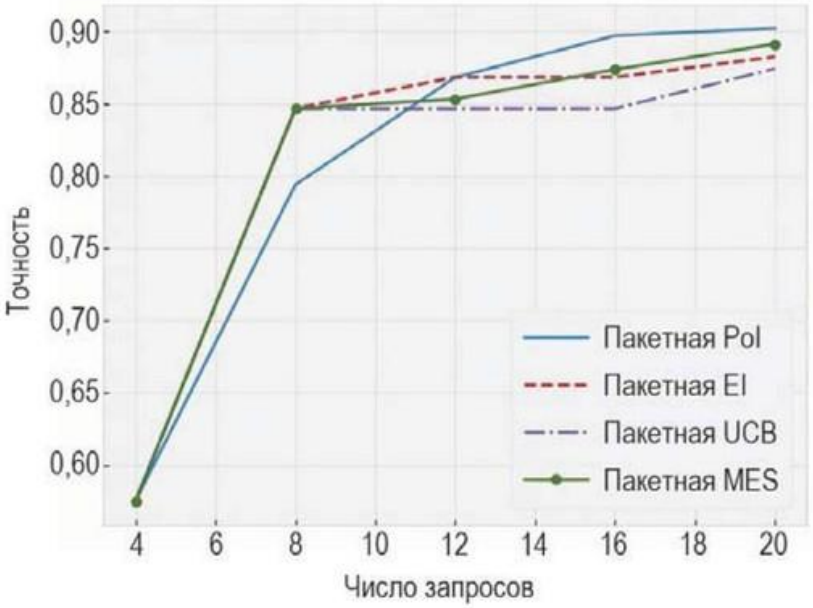


Рис. 7.18. Прогресс, достигнутый различными пакетными политиками в задаче настройки гиперпараметров (включая MES)

7.5. Упражнение 2: Оптимизация конструкции самолета

В этом упражнении мы запустим пакетные варианты политик, рассмотренные в данной главе, для оптимизации в области физики. Эта задача является самой высокоразмерной из тех, что мы встречали. Она даст возможность увидеть, как БО решает общую задачу оптимизации «черного ящика» в больших размерностях. Мы увидим, как различные пакетные политики работают при решении реальных задач.

В этом упражнении нас интересует оптимизации авиастроения, с которой обычно сталкиваются авиаконструкторы. У нас есть различные настраиваемые параметры (каждый из которых является измерением в пространстве поиска), которые управляют работой самолета. Ими могут быть длина и ширина самолета, форма и угол наклона крыльев относительно корпуса или угол наклона лопастей турбины и скорость их вращения. Задача инженера-оптимизатора — настроить значения этих параметров так, чтобы самолет работал наилучшим образом, или оптимизировать некоторые показатели производительности, вроде скорости или энергоэффективности.

Несмотря на то что у инженеров может быть хорошее представление, как некоторые из переменных влияют на характеристики самолета, хороший способ прове-

рить его работу — запустить различные компьютерные модели и понаблюдать за их поведением. На основе этих симуляций можно оценить конструкцию самолета на предмет того, насколько хорошо она эффективна по различным показателям производительности. Имея на руках программу моделирования, можно рассматривать процесс настройки как задачу оптимизации «черного ящика». Другими словами, мы не знаем, как каждый настраиваемый параметр влияет на конечные характеристики моделируемого самолета, но нам хотелось бы их оптимизировать для достижения наилучшего результата.

В упражнении представлена целевая функция, моделирующая процесс сравнительного анализа характеристик в конструкции самолета. Код представлен в `CH07/04 - Exercise 2.ipynb`. Рассмотрим задачу по шагам:

1. Реализуйте целевую функцию, которая симулирует производительность при тестировании. Она имеет четыре входных параметра и вычисляет оценку, которая количественно определяет полезность самолета, заданную ими. Поскольку функция рассматривается как «черный ящик», мы не знаем, что происходит внутри нее и как формируется результат:

```
def flight_utility(X):
    X_copy = X.detach().clone()
    X_copy[:, [2, 3]] = 1 - X_copy[:, [2, 3]]
    X_copy = X_copy * 10 - 5

    return -0.005*(X_copy**4 - 16 * X_copy**2 + 5 * X_copy).sum(dim=-1)+3
```

   Четыре параметра — это различные настройки самолета, масштабированные так, чтобы они располагались от 0 до 1. Другими словами, пространство поиска — это четырехмерный единичный гиперкуб. Несмотря на то что параметры не являются существенными для нашего подхода к оптимизации, их имена следующие:

```
labels = [
    "scaled body length",
    "scaled wing span",
    "scaled ?",
    "scaled ?"
]
```

   Несмотря на то что визуализировать всю четырехмерную функцию непросто, можно показать, как она ведет себя в двумерном пространстве. Рис. 7.19 демонстрирует это для различных пар настраиваемых параметров, показывая сложную нелинейную тенденцию в двумерных пространствах.

   Наша цель — найти максимальное значение этой функции с помощью БО.

2. Реализуйте модель ГП с постоянной функцией среднего и ядром Матерна (2.5) с помощью масштаба вывода, реализованного как объект `gpytorch.kernels.ScaleKernel`:

   - Учтите тот факт, что целевая функция является четырехмерной. Для этого необходимо указать аргумент `ard_num_dims = 4` при инициализации ядра.

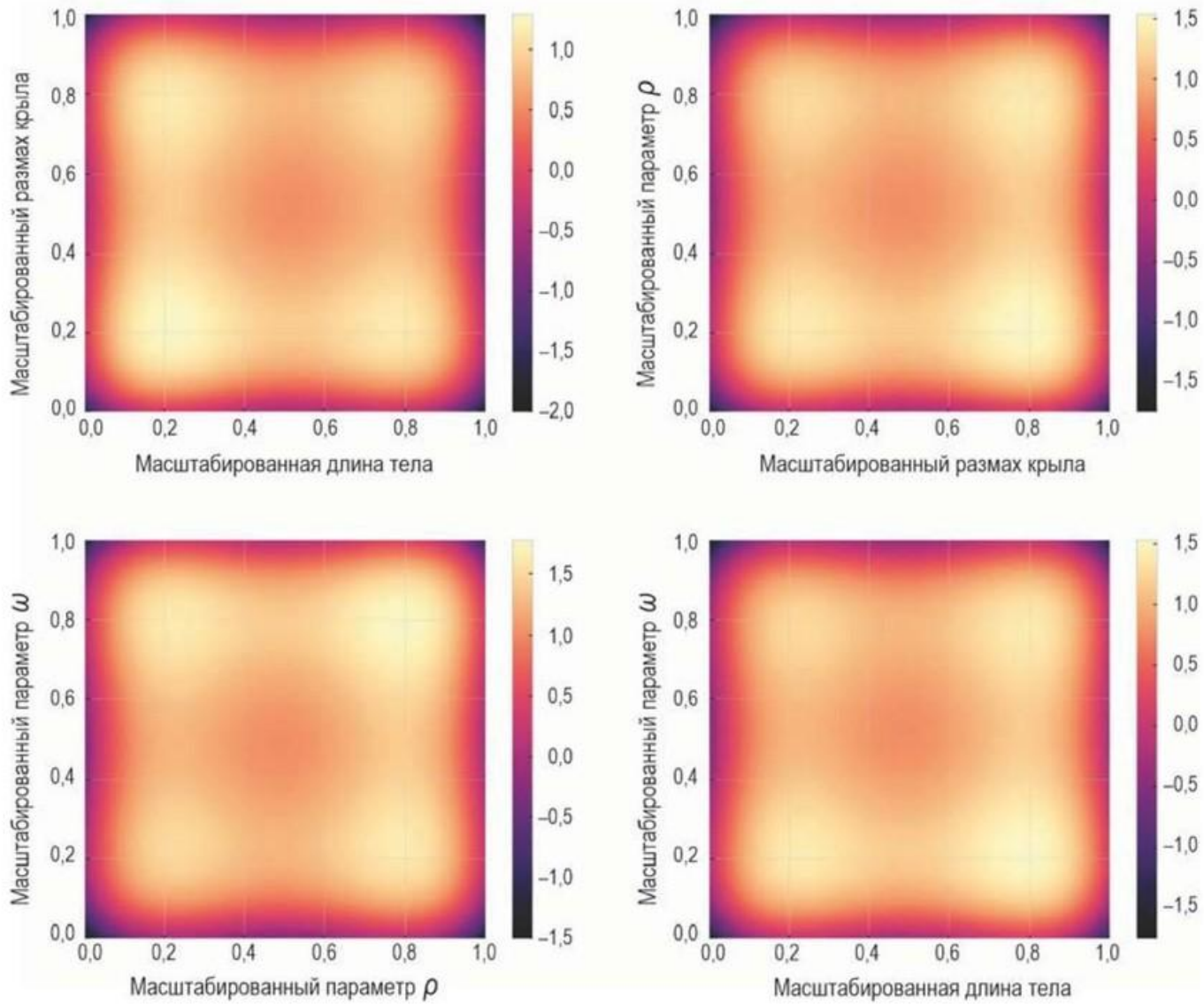


Рис. 7.19. Целевая функция моделируемой задачи по оптимизации конструкции самолета в различных двумерных подпространствах (парах настраиваемых параметров), которые показаны в виде осевых меток: яркие пятна указывают на высокие искомые значения, которые являются нашими целями оптимизации

ПРИМЕЧАНИЕ

Как работать с ядрами Матерна, описано в разделе 3.4.2, а также в упражнении к главе 3.

3. Реализуйте вспомогательную функцию, которая обучает ГП на заданном наборе данных. Она должна принять набор, обучить ГП с помощью градиентного спуска, чтобы минимизировать отрицательную логарифмическую вероятность и вернуть ГП вместе с его функцией вероятности (раздел 4.1.1).
4. Определите настройки задачи оптимизации:
   - Пространство поиска представляет собой четырехмерный единичный гиперкуб, поэтому нам нужна переменная с именем `bounds`, которая хранит следующий тензор с границами:

```
tensor([[0., 0., 0., 0.],
        [1., 1., 1., 1.]])
```

- Мы передаем эти границы политикам, которые запустим позже в упражнении.
- Суммируя каждый запуск, политика может выполнить в общей сложности 100 запросов к целевой функции (100 оценок функции) пакетами по 5 штук. Для каждой политики необходимо повторить эксперименты пять раз.

5. Запустите каждую пакетную политику из этой главы на только что реализованной целевой функции:
   - Каждый эксперимент должен начинаться со случайно выбранной оценки функции в качестве обучающего набора.
   - Запишите лучшие значения, найденные в ходе поиска.
   - Используйте последовательность Соболя из 5000 точек для TS и MES.
   - Использование MES для решения многомерной задачи требует больших вычислительных затрат. Распространенной стратегией упрощения этого является ограничение количества циклов оптимизации. Например, чтобы прекратить оптимизацию оценки приобретения после пяти циклов, передайте `cyclic_options={"maxiter":5}` во вспомогательную функцию `optimize_acqf_cyclic()`. Затем запустите в экспериментах эту более облегченную версию MES.
6. Постройте график хода оптимизации для политик и понаблюдайте за их эффективностью. Каждая из них должна иметь кривую, показывающую наилучшую среднюю точку в зависимости от количества используемых запросов и стандартных ошибок. Более подробную информацию, как сделать визуализацию, можно найти на последнем шаге упражнения 2 к главе 4.

Заключение

- Многие настройки оптимизации «черного ящика» в реальном мире позволяют проводить одновременно несколько оценок функции. Используя этот параллелизм, можно реализовать больше экспериментов в БО и потенциально добиться лучшей производительности.
- На каждой итерации пакетной оптимизации выбирается пакет запросов, по которым оценивается целевая функция. Это нужно, чтобы политики имели возможность оценивать пакеты запросов в соответствии с их полезностью, помогая найти глобальный оптимум.
- Расширить политики для пакетного выполнения не так просто, как выбрать точки данных, которые получают наибольшую оценку приобретения при последовательном режиме. Это приводит к тому, что выбранные запросы оказываются очень близко друг к другу, что противоречит цели параллелизма.
- Политики PoI, EI и UCB можно расширить для пакетного режима с использованием той же стратегии: максимальное значение в пакете запросов используется для количественной оценки всего пакета. С математической точки зрения такой подход требует перезаписи оценки приобретения как среднего значения некоторой интересующей величины.

- Благодаря своей случайной природе TS может быть легко расширена до пакетного режима. Вместо одноразовой выборки из ГП и ее максимизации данная политика повторяет процесс, пока не будет достигнут нужный размер пакета.
- Нахождение теоретико-информационного значения нескольких точек является трудной вычислительной задачей. Сложность заключается в том, что она не позволяет использовать алгоритм L-BFGS для поиска точки (или пакета точек), который максимизирует оценку приобретения для пакетной политики MES. Чтобы обойти эту вычислительную сложность, мы задействуем циклическую оптимизацию. Такая стратегия предполагает циклическое уточнение отдельных членов текущего пакета запросов, пока оценка приобретения не сойдется. Циклическую оптимизацию можно использовать в BoTorch с помощью вспомогательной функции `optimize_acqf_cyclic()`.
- Чтобы максимизировать производительность оптимизации, важно указать правильный размер пакета при использовании вспомогательных функций `functions optimize_acqf()` и `optimize_acqf_cyclic()` при поиске пакета, который максимизирует оценку приобретения для определенной политики. Это делается с помощью задания аргументу `q` желаемого размера пакета.
- Реализация в BoTorch большинства пакетных политик БО использует тот же интерфейс, что и реализация последовательных политик. Такая согласованность позволяет программисту перейти к пакетному режиму без необходимости существенно изменять свой код.

8 Удовлетворение особых ограничений с помощью ограниченной оптимизации

Темы главы:

- ❒ Оптимизация «черного ящика» с помощью ограничений.
- ❒ Учет ограничений при принятии решений.
- ❒ Реализация политик БО с учетом ограничений.

В предыдущих главах решения задач оптимизации «черного ящика» были направлены исключительно на максимизацию целевой функции без каких-либо других соображений. Это называется задачей *неограниченной оптимизации*, поскольку в данном случае можно свободно исследовать пространство поиска. Однако многие реальные ситуации не подходят под такую формулировку, и могут возникнуть издержки, из-за которых достижение глобального оптимума на практике становится невозможным.

Например, настраивая архитектуру нейросети, можно обнаружить, что увеличение количества слоев обычно дает более высокую точность. Соответственно миллионы и миллиарды слоев должны обеспечить наилучшую работу. Однако, если нет доступа к дорогим и мощным вычислительным ресурсам, запуск таких больших нейросетей нецелесообразен. В противном случае в задачах с настройкой гиперпараметров существовали бы вычислительные затраты, соответствующие глобальному оптимуму целевой функции. Поэтому при создании нейросети необходимо их учитывать и искать только те архитектуры, которые действительно возможно реализовать на практике.

Еще одна задача оптимизации «черного ящика», в которой нужно учитывать дополнительные ограничения, — это научные открытия (например, в химии или материаловедении). Предположим, ученые стремятся разработать химические вещества и материалы, которые оптимизировали бы некоторые желаемые характеристики. Это могут быть лекарства, которые очень эффективны против болезней, стекло, максимально устойчивое к давлению, или металлы, которые податливы и с ними легко работать. К сожалению, чем эффективнее лекарства против болезней, тем больше побочных эффектов они могут иметь, что делает их опасными. Или же

очень прочный тип стекла может оказаться слишком дорогим для производства в крупных масштабах.

Это примеры задач *ограниченной оптимизации* (**Constrained optimization**), когда нужно оптимизировать целевую функцию, удовлетворяя при этом другим требованиям (*ограничениям*). Стремление оптимизировать только целевую функцию может привести к решениям, которые будут нарушать важные правила, становясь бесполезными на практике. Вместо этого лучшим решением было бы определить другие регионы в пространстве поиска, которые одновременно давали бы высокие объективные значения и удовлетворяли важным требованиям.

В этой главе мы узнаем о задаче ограниченной оптимизации и рассмотрим примеры, когда наличие дополнительных требований (ограничений, правил, лимитов, договоренностей) может полностью изменить решение. Это приводит нас к необходимости использовать политики БО, которые учитывали бы данные требования. Мы познакомимся с вариантом политики *ожидаемого улучшения* (EI), который учитывает ограничения, и узнаем, как реализовать ее в BoTorch. К концу главы вы поймете задачу ограниченной оптимизации, узнаете, как она решается с помощью БО, и увидите, что используемая политика, удовлетворяющая ограничениям, работает намного лучше, чем та, что их не учитывает. Изученное в этой главе поможет вам решать наиболее приближенные к реальной жизни задачи и, таким образом, принимать более эффективные решения.

8.1. Учет ограничений в задаче ограниченной оптимизации

В реальном мире существует множество задач ограниченной оптимизации: создание лекарств с высокой эффективностью и минимальными побочными эффектами, поиск материалов, которые максимизируют желаемые характеристики и дешевы в производстве, или настройка гиперпараметров при сохранении низких вычислительных затрат.

> **Примечание**
>
> Мы фокусируемся именно на *ограничительных неравенствах* (**Inequality constraints**), где требуется, чтобы результат (y) находился в пределах заранее определенного числового диапазона ($a \le y \le b$).

Сначала необходимо более подробно рассмотреть данную задачу и выяснить, почему она математически отличается от аналогичной, но без учета ограничений, рассмотренных в предыдущих главах. Затем мы переопределим имеющуюся структуру БО с целью учесть дополнительные требования.

8.1.1. Ограничения могут изменить решение задачи оптимизации

Каким образом ограничения усложняют оптимизацию функции «черного ящика»? Во многих случаях регионы в пространстве поиска, дающие высокие целевые значения, могут нарушать ограничения, связанные оптимизацией.

Примечание

В данной задаче мы стремимся найти области, которые дают высокие целевые значения, поскольку наша цель — максимизировать значение целевой функции.

Если регионы с высокими целевыми значениями нарушают заданные ограничения, необходимо проводить поиск в других локациях, которые удовлетворяют требованиям.

Определение

Точка данных, нарушающая ограничения в задаче ограниченной оптимизации, называется *недопустимой* (**infeasible**), поскольку ее нельзя использовать в качестве решения. С другой стороны, точка данных, удовлетворяющая ограничениям, называется *допустимой* (**infeasible**). Наша цель — найти допустимую точку, которая максимизирует значение целевой функции.

Ограничения могут повлиять на качество оптимального решения или вообще изменить его. Рассмотрим пример на рис. 8.1, где целевая функция (сплошная линия) — это функция Форрестера, которую мы использовали в предыдущих главах. В дополнение к ней есть также *функция затрат* (пунктирная линия). Предположим, в этой задаче ограничение состоит в том, что *затраты* не должны превышать ноль ($c \leq 0$). Это означает, что только допустимые точки в заштрихованных областях на правой части рисунка могут быть использованы в качестве результата.



Рис. 8.1. Пример одномерной задачи ограниченной оптимизации: сплошная линия — целевая функция, которую нужно максимизировать, пунктирная линия — функция затрат, которая ограничивает условия задачи; допустимы только заштрихованные области (правая часть), поскольку они приводят к отрицательным затратам; здесь ограничение на положительные затраты приводит к уменьшению максимального целевого значения с 8 до 4

Примечание

Впервые функция Форрестера в качестве примера целевой функции была использована в разделе 2.4.1.

В результате область (где $x > 4$), содержащая истинный глобальный оптимум (обозначена ромбовидным маркером в правой части), отсекается. Глобальный оптимум, который дал бы целевое значение около 8, невозможен, а оптимальное ограничен-

ное решение (обозначенное звездочкой) достигается возле 4. Примером такого «отсечения» из реальной жизни является ситуация, когда эффективное лекарство имеет слишком серьезный побочный эффект, поэтому фармацевтическая компания решает использовать менее эффективный вариант того же химического вещества, чтобы сделать продукт безопасным.

Другой пример с той же целевой функцией и похожей функцией затрат показан на рис. 8.2, где наличие дополнительных затрат на ограничения меняет оптимальное решение. Без ограничения глобальный оптимум целевой функции находится при $x = 4{,}6$. Однако эта точка недопустима, поскольку она дает положительные затраты и, таким образом, нарушает ограничение. Оптимальное решение находится при $x = 1{,}6$. В реальном мире подобное может произойти, например, когда все семейство какого-то высокоэффективного препарата опасно для пациентов и не может быть произведено. Поэтому необходимо искать другие решения, химически отличающиеся от опасного препарата.

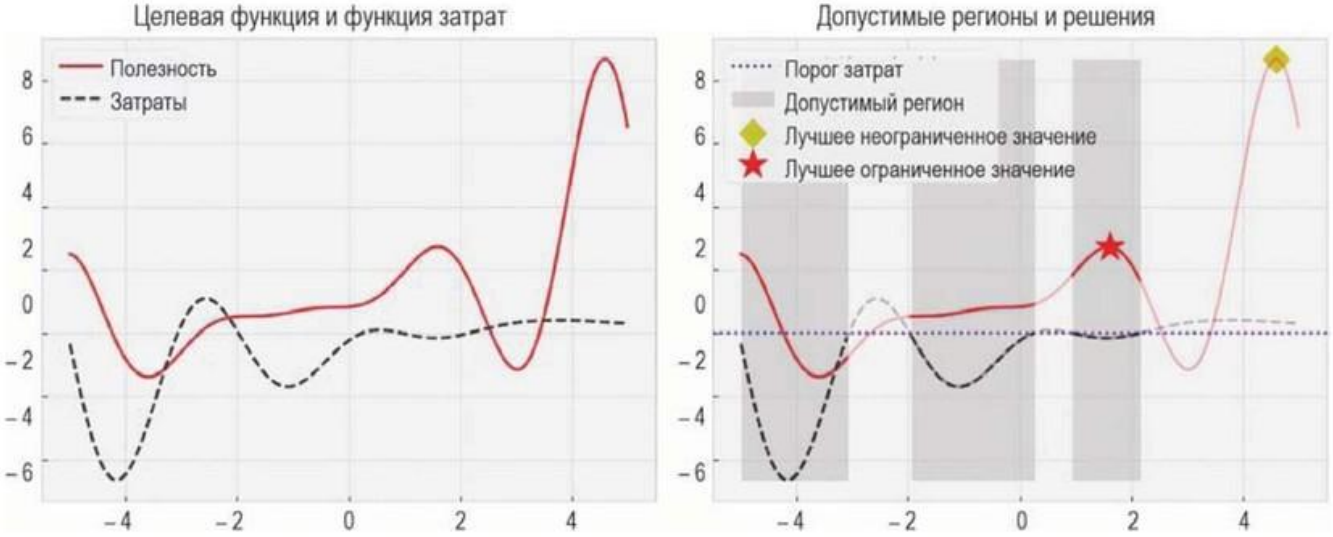


Рис. 8.2. Пример одномерной задачи ограниченной оптимизации: оптимальное решение меняется на другой локальный оптимум, поскольку ограничение стоимости исключает область, где $x > 3$

В целом ограничения в виде неравенств могут налагать сложные требования на задачу оптимизации и менять ее оптимальное решение. Таким образом, глобальный оптимум функции может быть исключен как недопустимая точка. Это распространенный сценарий в реальном мире:

- Непомерно большие нейросети имеют тенденцию достигать хороших прогностических характеристик, но их невозможно реализовать на практике.
- Самые эффективные лекарства зачастую слишком агрессивны и опасны для производства.
- Лучшие материалы часто слишком дороги в использовании.

Вместо использования неограниченной оптимальной точки, нарушающей определенные требования, следует изменить нашу стратегию решения, чтобы учесть ограничения и найти оптимальное допустимое решение. Иначе говоря, нам нужно преследовать две цели: оптимизацию целевой функции и удовлетворение заданных

требований. Неограниченная оптимизация приведет к неосуществимым решениям, которые невозможно использовать на практике. Вместо этого стоит поискать точки, которые одновременно дают высокие целевые значения и удовлетворяют ограничениям.

> **Замечание**
>
> В задаче ограниченной оптимизации нам необходимо сбалансировать оптимизацию целевой функции и удовлетворение требований.

8.1.2. Структура БО, учитывающая ограничения

Как следует подойти к задаче ограниченной оптимизации с точки зрения БО? В этом разделе мы узнаем, как изменить структуру БО, чтобы учесть ограничения.

Для обучения ГП мы используем точки данных, наблюдаемые в целевой функции, а затем делаем прогнозы на точки, которые не попали под наблюдения. Кроме того, при ограниченной оптимизации у нас есть одна или несколько функций, определяющих требования, которые нужно удовлетворять. Например, в разделе 8.1.1 функция затрат (пунктирная линия на рис. 8.1 и 8.2) определяет ограничения, при которых решение должно иметь отрицательные затраты ($c < 0$).

> **Примечание**
>
> Для дополнительной информации о структуре БО вы можете обратиться к разделам 1.2.3 и 4.1.1.

Предположительно, как и в случае с целевой функцией, нам не известно, как выглядит функция затрат. Она также представляет собой «черный ящик». Мы можем наблюдать значения только в тех точках данных, где мы запрашиваем целевую функцию, и исходя из этого определять, удовлетворяют ли они требованиям или нет.

> **Примечание**
>
> Если бы нам было известно, как выглядят функции, которые определяют ограничения, мы могли бы просто найти возможные области и ограничить пространство поиска в них. В нашей задаче мы предполагаем, что ограничения также являются «черными ящиками».

Поскольку функции, определяющие ограничения, являются «черными ящиками», можно использовать ГП для моделирования каждой из них, чтобы механизм принятия решений знал, где запрашивать целевую функцию в следующий раз. Процедура обучения каждого ГП аналогична, разница кроется лишь в использовании обучающих наборов (для каждого ГП свой набор):

- ГП, моделирующий целевую функцию, обучается на наблюдаемых целевых значениях.
- ГП, который моделирует функцию ограничений, обучается на наблюдаемых значениях затрат.

Структура БО с учетом ограничений показана на рис. 8.3. Она представляет собой модифицированную версию рис. 1.6:

- На первом шаге мы обучаем один ГП на данных целевой функции, а другой — на данных каждой функции, определяющей ограничение.
- На шаге 3 мы используем точку, определенную политикой, для запроса как целевой функции, так и функций, определяющих ограничения.

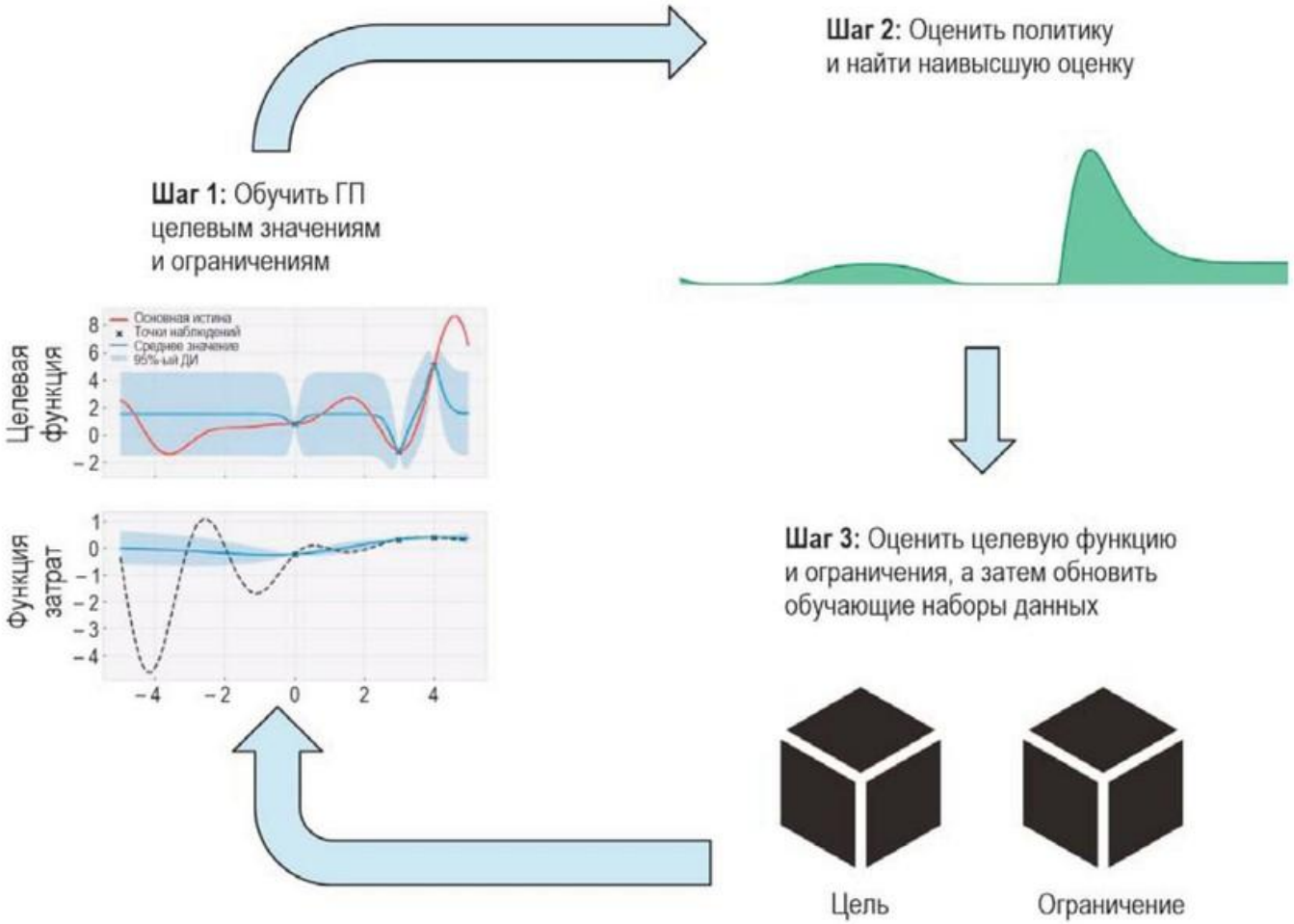


Рис. 8.3. Цикл БО с ограничениями: все функции (целевая и ограничительные) имеют свой ГП; политика рекомендует на следующем шаге запросить как целевую функцию, так и функции, определяющие ограничения

Шаги 1 и 3 на рис. 8.3 просты в реализации: нужно лишь одновременно поддерживать несколько ГП, отслеживать соответствующие наборы данных и обновлять их. Более интересный вопрос возникает на втором шаге (принятие решений). Как следует разрабатывать политику, которая помогает находить возможные регионы, дающие высокие целевые значения? Это тема для обсуждения в следующем разделе.

8.2. Принятие решений с учетом ограничений в БО

Эффективная политика должна обеспечивать как оптимизацию, так и соблюдение ограничений. Один из простых способов разработать ее — включить ограничения в механизм принятия решений при обычных условиях (без ограничений).

Для примера модифицируем политику EI, о которой более подробно вы можете узнать в разделе 4.3 (другие политики рассмотрим позже в этом разделе). Как вы помните, EI оценивает каждую запрашиваемую точку данных с помощью значения, которое показывает размер улучшения по сравнению с текущей действующей точкой.

> **Определение**
>
> Термин «действующая» относится к точке, которая имеет наивысшее целевое значение в обучающем наборе. «Преодолев» данное значение, можно добиться прогресса в оптимизации.

Оценка приобретения политики EI, которая вычисляет среднее значение улучшения для каждого потенциального запроса, игнорирует ограничительные неравенства в задаче ограниченной оптимизации, поэтому нельзя использовать эту политику «как есть». К счастью, существует простой способ учесть эти ограничения — масштабировать оценку приобретения EI для каждой *ненаблюдаемой* точки согласно тому, насколько вероятно, что она удовлетворяет ограничениям (является допустимой):

- Если точка данных удовлетворяет ограничениям, ее оценка будет умножена на большое число (высокая вероятность допустимости), таким образом, сохраняя высокое значение.
- Если точка данных не удовлетворяет ограничениям, ее оценка будет умножена на малое число (низкая вероятность допустимости), тем самым снижая ее приоритет.

> **Совет**
>
> Оценка приобретения ограниченной EI представляет собой произведение обычной оценки EI и вероятности, что точка данных удовлетворяет ограничениям.

Формула оценки приобретения для данной политики показана на рис. 8.4. При этом соблюдаются два условия: оценка EI способствует оптимизации целевой функции, а вероятность допустимости побуждает оставаться в пределах возможных регионов. Баланс между оптимизацией целевой функции и удовлетворением ограничений — это именно то, чего нам нужно достичь.

Как вычислить оценку EI мы уже знаем. Осталось найти второй член — вероятность допустимости для точки данных. Как отмечено на рис. 8.4, это делается с по-



Рис. 8.4. Формула для оценки приобретения ограниченной EI, которая выражается как произведение оценки приобретения для обычной EI и вероятности допустимости; политика направлена на оптимизацию целевого значения и удовлетворение ограничений одновременно

мощью гауссовских процессов, которые моделируют ограничения. Каждый из них предоставляет вероятностное предположение о форме функций, которые определяют ограничения. Исходя из этого, можно вычислить вероятность, что точка данных будет удовлетворять соответствующему неравенству.

В качестве примера предположим, что при решении задачи, определенной на рис. 8.2, мы наблюдаем целевую функцию и функцию затрат при $x = 0$, $x = 3$ и $x = 4$. На этом наборе мы обучаем два ГП: один для целевой функции, другой для функции затрат. Мы получим прогнозы, представленные на рис. 8.5.

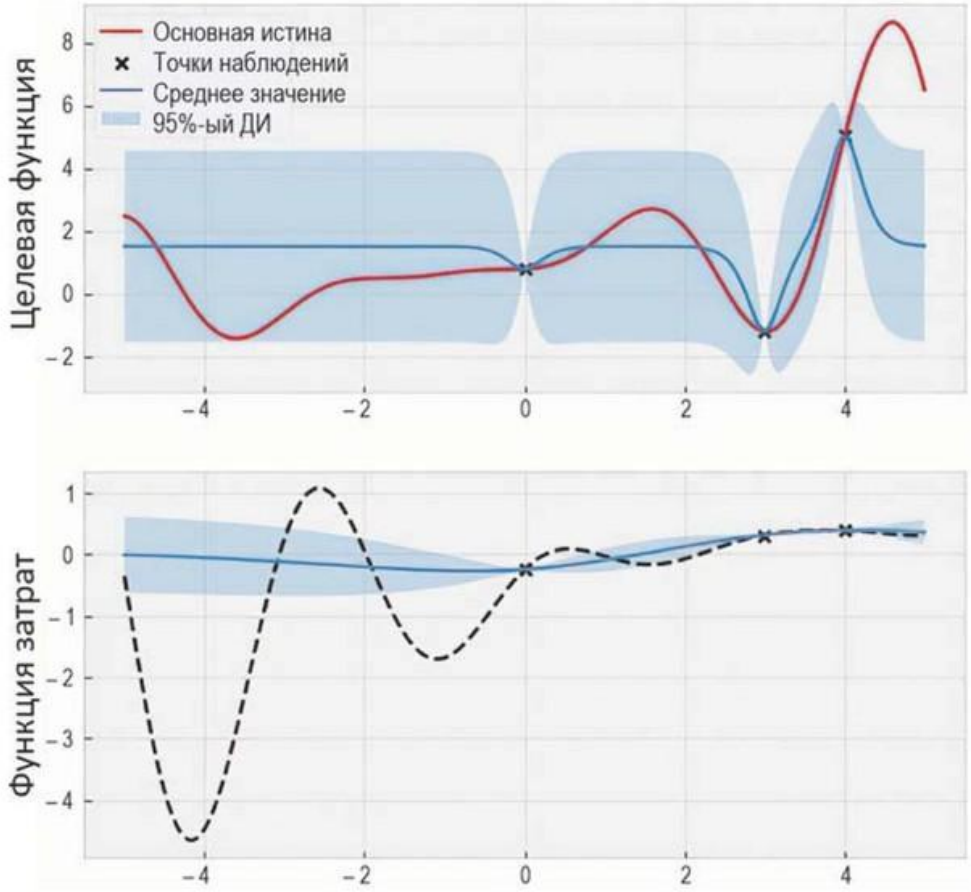


Рис. 8.5. Прогнозы относительно целевой и затратной функций, сделанные соответствующими ГП: каждый ГП позволяет рассуждать о форме каждой из функций вероятностным образом

Далее предположим, что нам необходимо вычислить оценку приобретения ограниченной EI для $x = 1$. У нас уже есть способ вычислить обычную оценку EI для любой точки данных, поэтому все, что нужно сделать сейчас, — посчитать вероятность, что $x = 1$ является допустимой точкой. Для этого рассмотрим нормальное распределение, представляющее наш прогноз относительно оценки при $x = 1$, как показано на рис. 8.6.

Левая часть рис. 8.6 содержит тот же ГП, что и нижняя часть рис. 8.5. Он «отсечен» по порогу ограничения, равному 0. Также здесь дополнительно показан ДИ прогноза для нормального распределения в точке $x = 1$. «Разрезая» ГП в этой точке, мы получаем правую часть рис. 8.6, где ДИ на обеих частях одинаковы. Другими словами, на рис. 8.6 показано увеличение масштаба по вертикали. А также вместо

демонстрации функции затрат мы сохраняем только ограничение (пунктирная линия) и прогноз ГП в точке $x = 1$, что является нормальным распределением. На рисунке видно: только выделенная часть нормального распределения в правой части представляет вероятность, что $x = 1$ удовлетворяет ограничению. Это нам и требовалось.

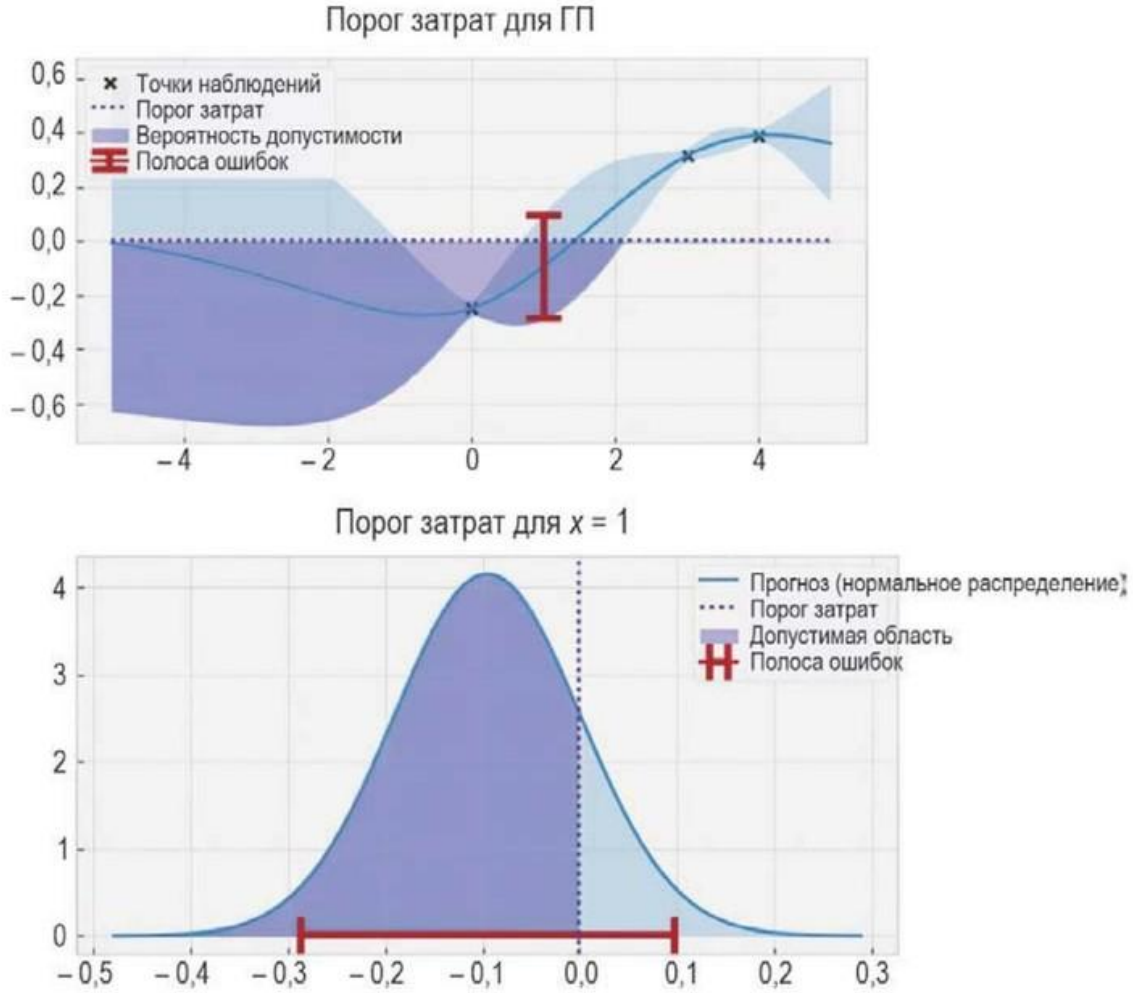


Рис. 8.6. Вероятность, что x = 1 является допустимой точкой, выделена более темным оттенком: на левой части показан весь ГП, а на правой — только нормальное распределение, соответствующее прогнозу при x = 1 (шкалы ошибок одинаковы на обеих частях); здесь допустимость соответствует усеченному нормальному распределению

Если рис. 8.6 напоминает вам рис. 4.9 и 4.10, посвященные политике PoI, то это потому, что процесс рассуждений в обоих случаях один и тот же:

- С помощью PoI мы вычисляем вероятность, что определенная точка данных дает целевое значение, которое *выше* действующего. Поэтому оно, в свою очередь, используется в качестве *нижней* границы, указывая, что нас интересуют только те сценарии, в которых целевое значение *выше* действующего.
- С помощью вероятности допустимости мы вычисляем вероятность, что определенная точка данных дает значение затрат ниже порогового (равного 0). Мы используем 0 в качестве *верхней* границы, указывая, таким образом, что мы нацелены на сценарии, в которых значение затрат *ниже* порогового (чтобы соблюдать ограничение по затратам).

Управление различными ограничениями в неравенствах

Ограничение в текущем примере требует, чтобы стоимость была меньше 0. Если бы у нас было ограничение, требующее, чтобы значение функции было выше некоторого порога, тогда вероятность допустимости была бы вероятностью того, что данная точка дает значение функции выше, чем некоторый порог (тогда заштрихованная область на рис. 8.6 будет справа от границы).

Если бы существовало ограничение, требующее, чтобы значение находилось внутри диапазона ($a \le y \le b$), то вероятностью допустимости была бы вероятностью того, что точка данных дает значение между нижней и верхней границами диапазона.

В нашем случае мы хотим вычислить вероятность, что значение затрат при $x = 1$ будет ниже 0, что представляет собой площадь заштрихованной области под кривой в правой части рис. 8.6. Как мы уже обсуждали в разделе 4.2.2, нормальное распределение позволяет вычислить данную площадь под кривой, используя функцию *кумулятивной плотности* (CDF). На рис. 8.6 вероятность, что точка $x = 1$ допустима, составляет примерно 84 %. Это вторая составляющая на рис. 8.4, которую

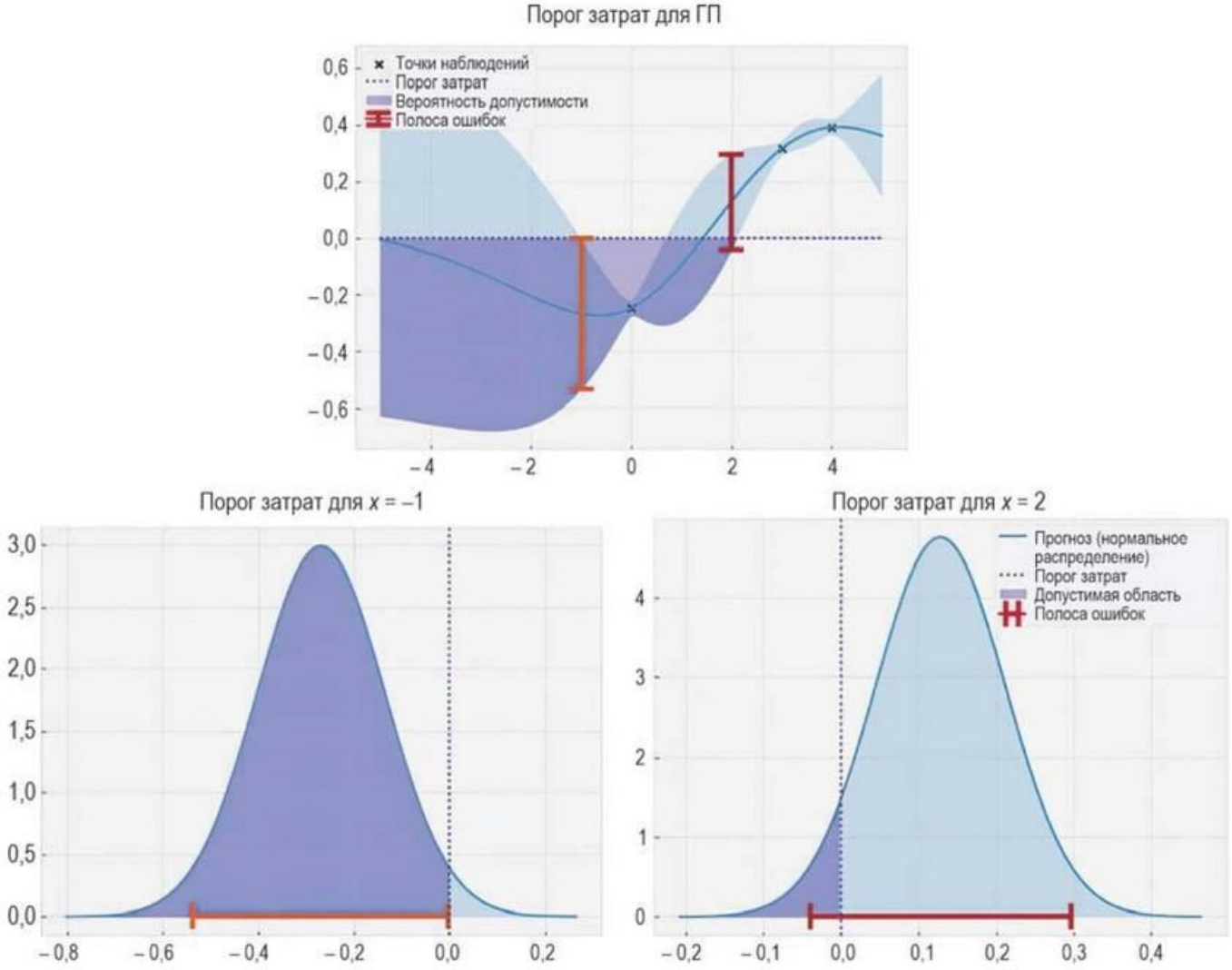


Рис. 8.7. Вероятность допустимости при $x = -1$ и $x = 2$ выделена более темным оттенком: на левой части показан весь ГП, на центральной — прогноз при x = –1, а на правой — прогноз при x = 2; выделенные части показывают вероятность допустимости, которая зависит от нормального распределения в заданной точке

мы используем для вычисления оценки приобретения для ограниченной политики EI.

Кроме того, мы можем вычислить вероятность допустимости в любой точке пространства поиска. Например, на рис. 8.7 показаны усеченные нормальные распределения для $x = -1$ (центральная часть) и $x = 2$ (правая часть). Как можно увидеть, вероятность, что данная точка допустима, зависит от прогностического нормального распределения в ней:

- При $x = -1$ почти все прогностическое нормальное распределение лежит ниже порога затрат в значении 0, поэтому вероятность допустимости здесь высока (почти 98%).
- При $x = 2$ только небольшая часть нормального распределения оказывается ниже порога затрат, в результате чего вероятность допустимости становится намного ниже (примерно 6%).

Имея возможность вычислить вероятность допустимости для любой заданной точки, теперь можно посчитать оценку приобретения ограниченной EI (рис. 8.4). Эта оценка балансирует между потенциально высоким целевым значением, которое количественно определяется обычной оценкой EI, и соблюдением ограничений неравенства, которое количественно определяется вероятностью допустимости.

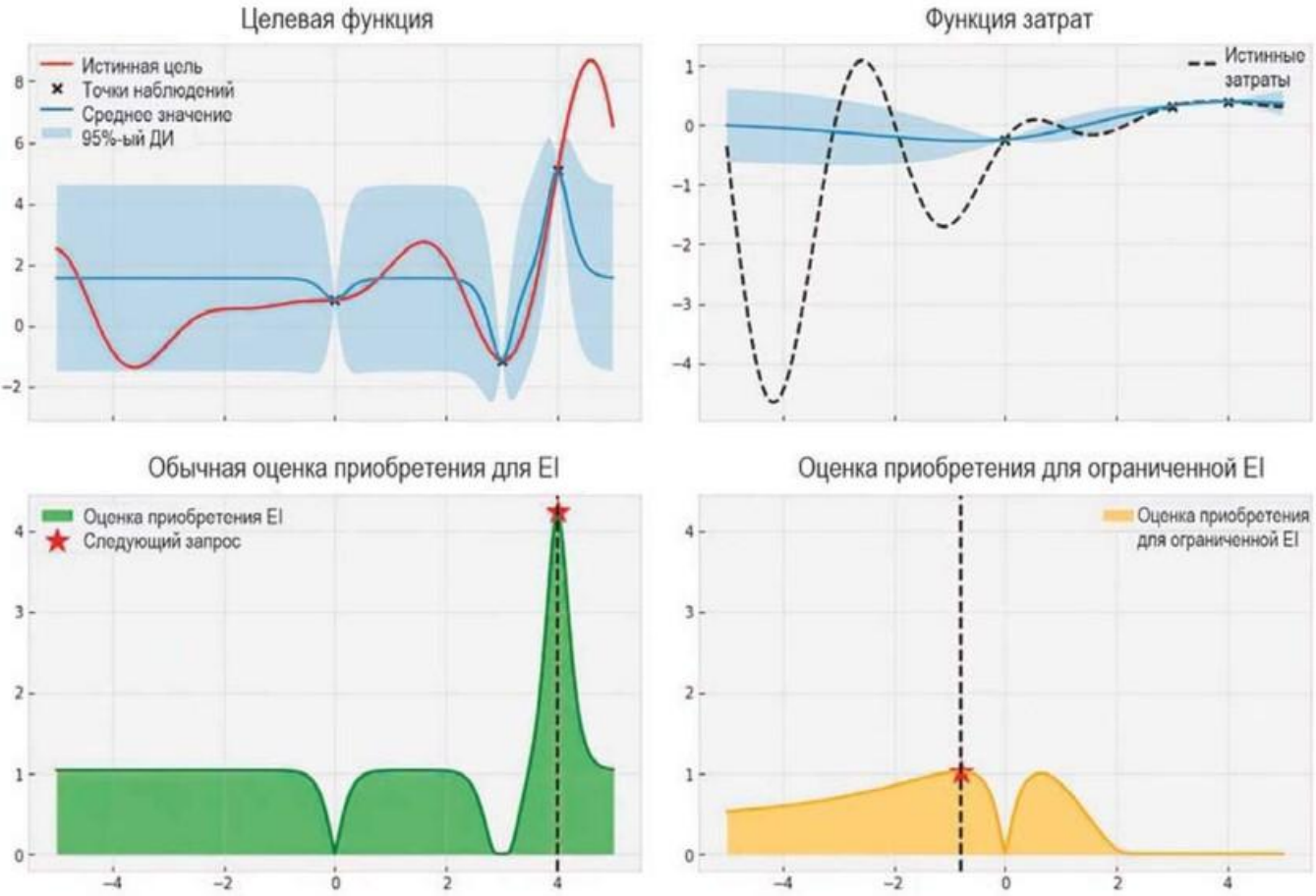


Рис. 8.8. Оценка приобретения обычной EI (внизу слева), оценка приобретения ограниченной EI (внизу справа), текущие прогнозы о целевой функции (вверху слева) и предположения о затратной функции (вверху справа): зная о требовании по затратам, ограниченная EI может избежать недопустимой области и порекомендовать для запроса совершенно другую точку

На рис. 8.8 эта оценка показана в нижней правой части вместе с обычной оценкой EI и текущими ГП. Можно увидеть, что ограниченная EI знает об ограничении на затраты, которое необходимо соблюсти, и присваивает нулевую оценку для области справа от пространства поиска (где $x > 2$). Это связано с тем, что затратный ГП (верхняя правая часть) считает этот регион недопустимым, и его следует избегать. В конечном счете обычная политика EI рекомендует в качестве следующей точки для запроса целевую функцию с недопустимой точкой $x = 4$. В свою очередь, ограниченная EI рекомендует точку $x = -0{,}8$, что удовлетворяет ограничению на затраты.

Несмотря на то что мы можем использовать ту же эвристику для других изученных политик, вроде UCB, TS и энтропийного поиска, мы больше не будем применять эту математически строгую процедуру к ним. На момент написания книги BoTorch поддерживает только ограниченную политику EI, которая широко используется для решения задач ограниченной оптимизации на практике. Поэтому в оставшейся части этой главы мы сосредоточимся только на ней и результатах ее оптимизации.

8.3. Упражнение 1: Ручное вычисление ограниченной EI

На рис. 8.4 видно, что оценка приобретения ограниченной политики EI является произведением оценки обычной EI и вероятности допустимости. Несмотря на то что класс `ConstrainedExpectedImprovement` в BoTorch обеспечивает реализацию оценки для ограниченной EI, фактически мы можем выполнить вычисления вручную. В этом упражнении мы попробуем это реализовать и сверим полученное с результатом из класса `ConstrainedExpectedImprovement`. Решение упражнения находится в `CH08/02 - Exercise 1.ipynb`:

1. Воссоздайте задачу ограниченной БО из `CH08/01 - ConstrainedOptimization.ipynb`, включая целевую и затратную функции, реализацию ГП, а также вспомогательную функцию `fit_gp_model()`, которая обучает ГП на некотором наборе данных.
2. Создайте тензор PyTorch, представляющий собой плотную сетку от −5 до 5, используя метод `torch.linspace()`. Он будет служить нашим тестовым набором.
3. Создайте набор данных для обучения, выбрав случайным образом 3 точки из пространства поиска (между –5 и 5), и оцените целевую и затратную функции в них.
4. Обучите один ГП на данных целевой функции, а другой — на данных функции затрат, используя вспомогательную функцию `fit_gp_model()`.
5. Используйте ГП, обученный на данных из функции затрат, чтобы вычислить вероятность допустимости для каждой точки в наборе. Вы можете использовать класс `torch.distributions.Normal` для инициализации объекта нормального распределения и вызова его метода `cdf()` для значения 0 (реализован как `torch.zeros(1)`), чтобы вычислить вероятность, что каждая точка данных даст стоимость ниже 0.
6. Инициализируйте обычную политику EI с аргументом `model` (поскольку ГП обучается на данных целевой функции) и аргументом `best_f` в качестве текущего допустимого действующего значения:

- Вычислите оценку EI для каждой точки в тестовом наборе.
- Более подробную информацию о реализации политики EI можно найти в разделе 4.3.

7. Инициализируйте ограниченную политику EI и вычислите ее оценку для каждой точки в тестовом наборе.
8. Вычислите произведения оценок EI и вероятностей допустимости и убедитесь, что ручное вычисление приводит к тем же результатам, что и реализация BoTorch. Для удобства можно использовать `torch.isclose(a,b,atol=1e-3)`, который выполняет поэлементное сравнение между двумя тензорами `a` и `b`, определяя `atol = 1e-3` для учета числовой нестабильности с целью убедиться, что все соответствующие оценки совпадают.
9. Постройте оценки приобретения обычной EI и ограниченной EI на графике и визуально убедитесь, что первые всегда больше или равны вторым. Докажите это, используя рис. 8.4.

8.4. Реализация ограниченной EI с помощью BoTorch

Несмотря на то что можно вручную умножить оценку EI и вероятность допустимости, BoTorch берет на себя все вычисления на низком уровне. Это означает, что мы можем импортировать ограниченную политику EI из BoTorch и использовать ее как любую другую политику БО без особых затрат. В этом разделе мы узнаем, как это сделать, а реализованный здесь код включен в `CH08/01 – ConstrainedOptimization.ipynb`.

Для начала нам нужны целевая и затратная функции, которые определяют ограничение, как на рис. 8.2. В следующем коде целевая функция реализована как `objective()`, а функция затрат — `cost()`. Пространство поиска находится между –5 и 5, поэтому переменная `bounds` будет содержать эти границы, позже они будут переданы в политики:

```
#Целевая функция, которую необходимо максимизировать
def objective(x):
    y = -((x + 1) ** 2) * torch.sin(2 * x + 2) / 5 + 1 + x / 3
    return y

#Функция затрат
def cost(x):
    return -(0.1 * objective(x) + objective(x - 4)) / 3 + x / 3 - 0.5

#Границы пространства поиска
lb = -5
ub = 5
bounds = torch.tensor([[lb], [ub]], dtype=torch.float)
```

Нам также нужна реализация класса модели и вспомогательная функция `fit_gp_model()`, которая обучает ГП на наборе данных. Поскольку ограниченная оптимизация не требует каких-либо изменений (ни в ГП, ни в том, как мы его обу-

чаем), можно повторно использовать реализацию класса и вспомогательную функцию из предыдущих глав. Более подробное обсуждение этого можно найти в разделе 4.1.1.

Чтобы оценить эффективность оптимизации для используемых политик, укажем, что каждый запуск БО имеет 10 запросов, а общее количество запусков может быть реализовано 10 раз:

```
num_queries = 10
num_repeats = 10
```

> **Примечание**
>
> Мы запускаем каждую политику БО несколько раз, чтобы получить целостное представление о ее эффективности. Для более подробной информации о повторных экспериментах можно обратиться к упражнению 2 в главе 4.

В заключение нужно изменить цикл БО, чтобы учесть изменения, показанные на рис. 8.3. На шаге 1 (т. е. в начале каждого шага цикла) необходимо переобучить несколько ГП: один для целевой функции, а другой (другие) для ограничения (ограничений).

Поскольку уже есть вспомогательная функция `fit_gp_model()`, обучающая ГП, этот шаг сводится к передаче ей соответствующих наборов данных. В текущем примере функция затрат, определяющая ограничение, только одна, поэтому всего мы имеем два ГП, которые можно переобучить с помощью следующего кода:

```
#Обучаем ГП на данных целевой функции
utility_model, utility_likelihood = fit_gp_model(
    train_x, train_utility.squeeze(-1)
)

#Обучаем ГП на данных затратной функции
cost_model, cost_likelihood = fit_gp_model(
    train_x, train_cost.squeeze(-1)
)
```

Переменная `train_x` содержит места, в которых мы оценили целевые и затратные функции, `train_utility` — соответствующие целевые значения, а `train_cost` — значения затрат.

Шаг 2 на рис. 8.3 отсылает нас к запуску политики, которую мы вскоре научимся делать. На шаге 3 мы оцениваем целевую и затратную функции в точке данных, которая рекомендована политикой и хранится в переменной `next_x`:

```
#Оцениваем целевую и затратную функции в рекомендуемой точке
next_utility = objective(next_x)
next_cost = cost(next_x)

#Обновляем различные наборы данных
train_x = torch.cat([train_x, next_x])
train_utility = torch.cat([train_utility, next_utility])
train_cost = torch.cat([train_cost, next_cost])
```

Еще одна вещь, которую необходимо предпринять, — отследить прогресс оптимизации. В отличие от задачи неограниченной оптимизации, где мы просто на каждом этапе записываем действующее значение (наивысшее целевое значение, наблюдаемое на данный момент), здесь необходимо отфильтровать недопустимые наблюдения, прежде чем брать максимум. Для этого надо сначала создать тензор со строками `num_repeats` (по одной для каждого повторного запуска) и столбцами `num_queries` (по одному для каждого шага). По умолчанию он содержит только одно значение, обозначающее полезность, если в БО не найдено возможных точек.

На рис. 8.2 видно, что целевая функция превышает значение −2 повсюду в пространстве поиска (между –5 и 5), поэтому оно используется в качестве значения по умолчанию:

```
#Проверяем, найдена ли допустимая точка
default_value = -2
feasible_incumbents = torch.ones((num_repeats, num_queries)) * default_value
```

Затем на каждом шаге цикла записываем только допустимую действующую точку, выбирая максимум из отфильтрованных наблюдений:

```
#Проверяем, найдена ли допустимая точка
feasible_flag = (train_cost <= 0).any()

if feasible_flag:
    feasible_incumbents[trial, i] = train_utility[train_cost <= 0].max()
```

Данный код завершает ограниченный цикл БО. Все, что осталось сделать, — объявить политику, которую необходимо использовать для решения задачи ограниченной оптимизации. В данном случаем мы задействуем ограниченную EI (раздел 8.2), используя класс `ConstrainedExpectedImprovement` из BoTorch. Он принимает ряд важных аргументов:

- `model` — список моделей ГП для целевой функции (`utility_model` в данном случае), а также для функций, определяющих ограничения (`cost_model` в данном случае). Список создан с использованием класса `model_list_gp_regression.ModelListGP` из модуля `models` в BoTorch: `ModelListGP(utility_model, cost_model)`.
- `objective_index` — индекс ГП, который моделирует целевую функцию в списке моделей `model`. Поскольку `utility_model` — это первый ГП, который передается в `ModelListGP`, в нашем случае индекс равен 0.
- `constraints` — словарь, сопоставляющий индекс каждой функции, которая определяет ограничение, со списком из двух элементов, которые хранят нижнюю и верхнюю границы этого ограничения. Если ограничение не имеет нижней или верхней границы, вместо фактического числового значения используется `None`. В данном примере нам нужны затраты, соответствующие `cost_model`, которая имела бы индекс 1, чтобы быть максимумом в 0, поэтому мы устанавливаем `constraints={1: [None, 0]}`.
- `best_f` — текущая допустимая действующая точка. Переменная равна или `train_utility[train_cost<= 0].max()`, если найдена хотя бы одна такая точка, или значению по умолчанию (−2) в противном случае.

В целом ограниченная политика EI инициализируется следующим образом:

```
policy = botorch.acquisition.analytic.ConstrainedExpectedImprovement(
    #Список моделей ГП
    model=botorch.models.model_list_gp_regression.ModelListGP(
        utility_model, cost_model
    ),
    #Текущая допустимая действующая точка
    best_f=train_utility[train_cost <= 0].max(),
    #Индекс целевой функции в списке моделей
    objective_index=0,
    #Словарь, сопоставляющий индекс каждого ограничения с нижней и верхней границами
    constraints={1: [None, 0]}
)
```

Имея реализацию ограниченной EI, теперь мы можем применить ее к нашей одномерной задаче ограниченной оптимизации и посмотреть ее производительность. Для сравнения мы также запустим обычную версию EI.

На рис. 8.9 показаны средние допустимые действующие значения этих двух политик, а также полосы ошибок в зависимости от времени. Можно наблюдать, что по сравнению с обычной EI вариант с ограничениями в среднем находит лучшее допустимое решение и почти всегда приходит к наилучшему возможному варианту.

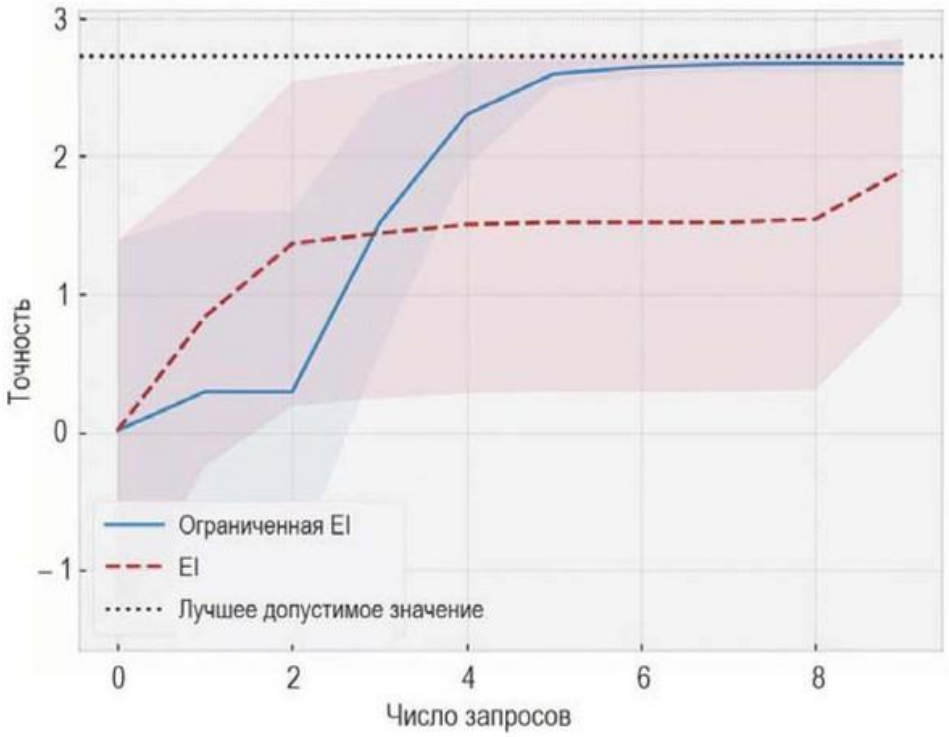


Рис. 8.9. Прогресс в одномерной задаче оптимизации ограниченной политики EI: по сравнению с обычной EI ограниченный вариант в среднем находит лучшее допустимое решение

Ограниченная EI имеет тенденцию отклоняться в сторону недопустимого оптимума. Проверяя действующие значения, найденные политикой, можно заметить, что во многих запусках она не прогрессирует относительно своего начального значения:

```
torch.set_printoptions(precision=1)
print(ei_incumbents)

Output:
tensor([[ 0.8,  0.8,  0.8,  0.8,  0.8,  0.8,  0.8,  0.8,  0.8,  0.8],
        [-2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0],
        [ 2.2,  2.2,  2.7,  2.7,  2.7,  2.7,  2.7,  2.7,  2.7,  2.7],
        [ 2.5,  2.5,  2.5,  2.5,  2.5,  2.5,  2.5,  2.5,  2.5,  2.5],
        [-2.0,  0.2,  1.9,  2.3,  2.6,  2.7,  2.7,  2.7,  2.7,  2.7],
        [-2.0,  0.5,  2.1,  2.4,  2.5,  2.5,  2.5,  2.5,  2.7,  2.7],
        [-2.0,  1.5,  2.5,  2.5,  2.5,  2.5,  2.5,  2.5,  2.5,  2.5],
        [-2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0],
        [ 1.9,  1.9,  2.5,  2.5,  2.7,  2.7,  2.7,  2.7,  2.7,  2.7],
        [ 2.7,  2.7,  2.7,  2.7,  2.7,  2.7,  2.7,  2.7,  2.7,  2.7]])
```

В этой главе мы узнали о том, что такое задача ограниченной оптимизации функции «черного ящика», а также чем она отличается от классической задачи оптимизации из предыдущих глав. Мы увидели, что эффективная политика должна преследовать как оптимизацию целевой функции, так и удовлетворение ограничений, которые на нее наложены. Мы также реализовали ограниченный вариант EI на практике, добавив к оценке приобретения коэффициент, равный вероятности допустимости. Новая полученная оценка приобретения смещает стратегию поиска в сторону возможных регионов, тем самым лучше направляя нас к возможному оптимуму.

В следующей главе мы обсудим новый сценарий БО — *разноточную оптимизацию*, при которой запрос целевой функции требует различных затрат. Этот параметр вынуждает нас найти баланс между поиском высокого целевого значения и сохранением имеющегося количества запросов.

8.5. Упражнение 2: Ограниченная оптимизация при конструировании самолета

В этом упражнении приведена задача ограниченной оптимизации с использованием целевой функции полезности из упражнения 2 к главе 7. Данный процесс позволяет запустить ограниченную БО для задачи более высокой размерности, в которой неочевидно, где находится оптимальное решение с точки зрения допустимости. Упражнение включено в `CH08/03 - Exercise 2.ipynb`. Выполните следующие шаги:

1. Воссоздайте задачу из `CH07/04 - Exercise 2.ipynb`, включая целевую функцию с именем `flight_utility()`, границы пространства поиска (четырехмерный единичный гиперкуб), реализацию ГП и вспомогательную функцию `fit_gp_model()`, которая обучает ГП на некотором наборе данных.
2. Реализуйте функцию затрат, которая имитирует стоимость создания конструкции самолета, заданную четырехмерными входными данными:

```
def flight_cost(X):
    X = X * 20 - 10

    part1 = (X[..., 0] - 1) ** 2

i = X.new(range(2, 5))
    part2 = torch.sum(i * (2.0 * X[...,1:] ** 2 - X[...,:-1]) ** 2, dim=-1)
    return -(part1 + part2) / 100_000 + 2
```

Рис. 8.10 визуализирует функцию затрат для различных пар настраиваемых параметров, показывая сложную нелинейную тенденцию в данных двумерных пространствах.

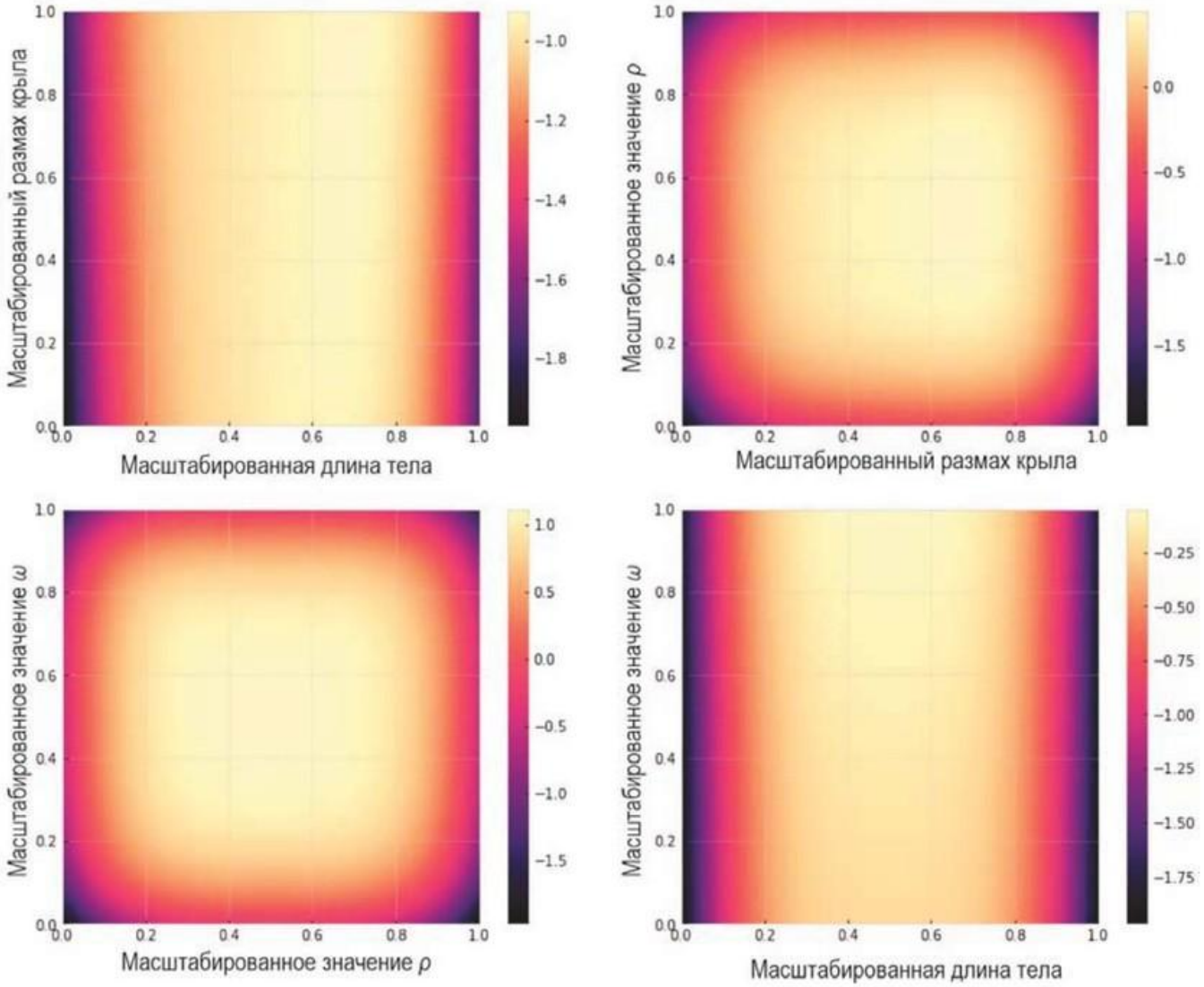


Рис. 8.10. Функция затрат моделируемой задачи оптимизации для конструкции самолета в различных двумерных подпространствах, которые соответствуют парам настраиваемых параметров, показанных в виде меток осей

3. Максимизируйте целевую функцию `flight_utility()`, соблюдая при этом ограничение, согласно которому затраты, вычисленные функцией `flight_cost()`, меньше или равны 0:

- Количество запросов, которые политика может выполнить в каждом эксперименте, установите равным 50. Также каждая политика должна провести 10 повторных экспериментов.
- Значение по умолчанию, определяющее ход оптимизации, если осуществимое решение не найдено, установите на −2.

4. Запустите для этой задачи ограниченную и обычную политики EI, а затем визуализируйте и сравните их средний прогресс (вместе с полосами ошибок). График должен выглядеть примерно так, как показано на рис. 8.9.

Заключение

- Ограниченная оптимизация — это задача, в которой, помимо целевой функции, необходимо также удовлетворять некоторым ограничениям (требованиям), которые на нее накладываются. Данный вариант оптимизации распространен при открытии материалов и лекарств, а также при настройке гиперпараметров, когда оптимум целевой функции слишком сложен или опасен при использовании на практике.
- Точка данных, которая удовлетворяет ограничениям, называется *допустимой*. Точка данных, которая не удовлетворяет ограничениям, — *недопустимой*. Мы стремимся найти такую точку, которая максимизирует целевую функцию среди других допустимых точек.
- Ограничения могут существенно изменить решение задачи оптимизации, отсекая или даже исключая область с высокими целевыми значениями. Поэтому необходимо внимательно учитывать ограничения при оптимизации целевой функции.
- В рамках ограниченной байесовской оптимизации мы обучаем ГП для каждой из функций, которые определяют ограничения. Эти ГП позволяют предполагать, будет ли точка данных удовлетворять ограничениям вероятностным образом. В частности, вероятность допустимости определенной точки легко вычислить благодаря факту, что прогностические распределения ГП являются нормальными распределениями.
- Политику EI можно изменить, чтобы учесть ограничения, добавив вероятность допустимости к ее оценке приобретения. Ограниченная политика EI может сбалансировать оптимизацию целевой функции и удовлетворение ограничений.
- BoTorch предоставляет реализацию класса ограниченной политики EI. Для этого нужно передать ГП, которые моделируют целевые функции и функции ограничений, а также объявить нижнюю и верхнюю границы ограничения.

9 Балансирование между полезностью и затратами с помощью многовариантной оптимизации

Темы главы:

- Задача разноточной оптимизации с переменными затратами.
- Обучение ГП на данных из нескольких источников.
- Реализация разноточной политики с учетом затрат.

Рассмотрим следующие вопросы:

- Стоит ли доверять онлайн-обзорам, в которых говорится, что новый сезон вашего любимого телешоу не так хорош, как предыдущие, и можно перестать его смотреть? Или же, наоборот, нужно потратить на него время, чтобы выяснить, действительно ли он так же хорош?
- Если модель нейросети не работает достаточно хорошо после обучения, должен ли ML-инженер сократить свои потери и переключиться на другую модель? Или же ему следует продолжить обучать модель еще в течение какого-то времени в надежде достичь более высокой производительности?
- Когда физик хочет понять какое-то явление, достаточно ли для этого будет компьютерного моделирования? Или же получить представление можно только с помощью настоящих физических экспериментов?

Эти вопросы схожи тем, что требуют от человека выбора между двумя возможными вариантами, способными дать ответ на интересующую задачу. С одной стороны, можно выбрать действие с относительно небольшими затратами, но ответ, полученный в результате, может быть искажен шумом (недостаточно точен) и, следовательно, быть ошибочным. С другой стороны, можно выбрать более дорогостоящее действие, которое поможет прийти к более точному выводу:

- Изучение обзоров о новом сезоне телешоу займет всего несколько минут. Но есть вероятность, что предпочтения рецензентов будут сильно отличаться от ваших, а сериал все равно вам понравится. Единственный способ узнать наверняка — посмотреть самостоятельно, но это займет много времени.

- Производительность нейросети после обучения может выдавать хорошую производительность, но при этом быть намного ниже максимально возможной. Более продолжительное обучение займет больше времени и ресурсов, потраченных на потенциально бесполезную задачу, если в конечном итоге модель будет работать плохо.
- Компьютерное моделирование может рассказать физику многое о явлении, но не способно охватить все аспекты реального мира. Поэтому вполне вероятно, что оно не сможет дать правильное представление. С другой стороны, проведение физических экспериментов, безусловно, прояснит многие моменты, но будет стоить дополнительных денег и усилий.

Эти ситуации относятся к классу задач, который называется *многовариантностью* или *разноточностью* (**multifidelity**) при принятии решений, где можно наблюдать какое-то явление на различных уровнях детализации и при разных затратах. Наблюдение на поверхностном уровне может быть недорогим и простым, но оно не даст достаточно информации. С другой стороны, более внимательное исследование может потребовать больше усилий. Термин «точность» (**fidelity**) здесь означает, насколько верно наблюдение отражает истину о рассматриваемом явлении. Недорогие наблюдения с низкой точностью имеют много шума и, таким образом, могут привести к неправильному выводу. Однако высокоточные (высокого качества) наблюдения являются дорогостоящими и, следовательно, не могут быть легкодоступными. Оптимизация «черного ящика» имеет свой собственный вариант разноточности.

> **Определение**
>
> *Разноточная оптимизация* (**Multifidelity optimization**) — это разновидность задачи оптимизации, при которой, помимо истинной целевой функции, мы также наблюдаем некоторые низкоточные «приближения» (аппроксимации, приближенные модели). Они не совсем точны, но все же предоставляют достаточно информации о цели. Приближения с низкой точностью могут быть оценены с меньшими затратами, чем истинная целевая функция.

При разноточной оптимизации необходимо использовать несколько источников данных одновременно, чтобы получить как можно больше информации о том, что нас интересует (это и есть оптимум целевой функции). В этой главе мы более подробно рассмотрим проблему разноточной оптимизации, а также подход к ней с точки зрения БО. Узнаем о стратегии, которая балансирует между исследованием целевой функции и затратами на нее, что приведет нас к созданию рациональной и экономичной политики для разноточного сценария. Затем мы узнаем, как реализовать данную задачу оптимизации и политику учета затрат в Python. К концу главы мы научимся реализовывать разноточную БО и увидим, что наша стратегия учета затрат более эффективна, чем алгоритмы, использующие лишь основную целевую функцию.

9.1. Использование приближений с низкой точностью для изучения дорогостоящих явлений

Сначала обсудим, зачем нужна задача разноточности в БО, а также ее постановку и примеры из реальной жизни. Это поможет прояснить, что нам нужно от политики принятия решений в данной ситуации.

При простейшем сценарии мы оцениваем целевую функцию на каждой итерации поиска, хорошо обдумывая, где это лучше сделать, чтобы добиться максимального прогресса в оптимизации. Необходимость в тщательном поиске вытекает из высокой стоимости вычислений, что типично для дорогостоящих задач оптимизации «черного ящика». Затраты могут касаться как времени, которое тратится на обучение большой нейросети при поиске лучшей архитектуры, так и денег/усилий, необходимых, например, для синтеза экспериментальных лекарств и проведения опытов для проверки его эффективности.

Но что, если существуют способы оценить результат вычисления без фактического исследования целевой функции $f(x)$? Например, в дополнение к ней мы можем запросить некоторый недорогой «суррогат» $\overline{f}(x)$. Он является неточным приближением (аппроксимацией) к цели, поэтому его оценка не даст нам полной картины об основной истине $f(x)$. Однако это все же даст некоторое неполное, но достаточное понимание $\overline{f}(x)$ для решения задачи. Вопрос, который нужно задать себе, заключается в следующем: как сбалансировать использование $f(x)$, запрос которой требует больших затрат, но дает точную информацию, и использование суррогата $\overline{f}(x)$, запрос которого имеет низкую точность, но стоит недорого? Этот баланс проиллюстрирован на рис. 9.1, где $f(x)$ — это источник данных с высокой точностью, а суррогат $\overline{f}(x)$ — низкоточное приближение.

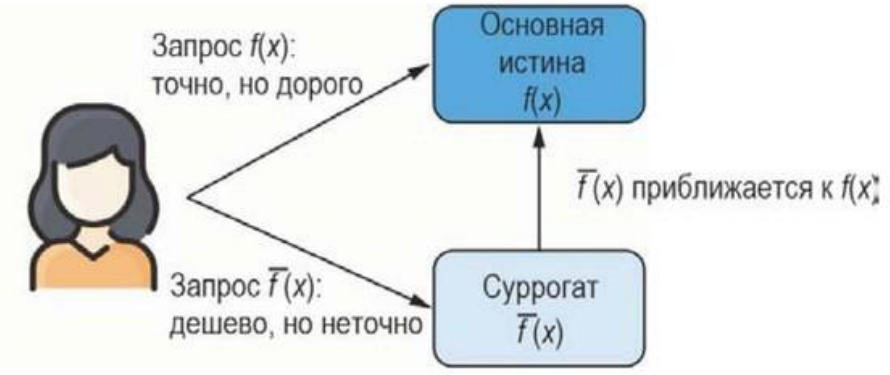


Рис. 9.1. Модель разноточного принятия решений, где необходимо сбалансировать запрос истинной целевой функции $f(x)$ для получения точной информации и запрос недорогого суррогата $\overline{f}(x)$

Использование приближений с низкой точностью достаточно распространено в реальном мире:

- *Обучение нейросети в течение ограниченного небольшого количества времени, чтобы оценить ее производительность на некотором наборе данных.* Например, обучение за 5 итераций — это низкоточное приближение к оптимальной производительности, которая может быть достигнута за 50 итераций.

- *Запуск компьютерного моделирования вместо реального эксперимента в целях изучить какое-либо научное явление.* Это имитирует физические процессы, происходящие в реальном мире, и делает некоторое приближение явления, которое хочет изучить физик. Однако его низкая точность не может на достаточном уровне отобразить реальный мир.

В задаче разноточной оптимизации мы можем выбирать между высокоточным запросом $f(x)$ и низкоточным запросом $\overline{f}(x)$, чтобы наилучшим образом изучить и оптимизировать целевую функцию $f(x)$. Конечно, первый вариант даст больше информации, но его стоимость не позволит делать такие запросы много раз. Вместо этого можно воспользоваться недорогим и низкоточным приближением $\overline{f}(x)$, чтобы узнать, как можно больше о нашей цели, минимизируя при этом затраты.

Множественные низкоточные приближения

Для простоты в примерах этой главы мы работаем только с одним приближением низкой точности $\overline{f}(x)$. Однако в реальных ситуациях их может быть множество $\overline{f}_1(x)$, $\overline{f}_2(x)$, ..., $\overline{f}_k(x)$, и каждое из них может иметь свою собственную стоимость запроса и точность.

Байесовский подход (который мы изучим в следующем разделе) не ограничивает их количество, поэтому в упражнении 2 к этой главе мы решим задачу разноточной оптимизации с двумя низкоточными приближениями $\overline{f}_1(x)$ и $\overline{f}_2(x)$.

Наличие нескольких приближений применимо, когда, например, компьютерное моделирование реального эксперимента имеет настройку, которая контролирует качество приближения. Если оно будет низким, программа выполнит грубое моделирование и быстрее вернет результат. С другой стороны, если оно будет высоким, программе может потребоваться больше времени, чтобы лучше приблизиться к реальному явлению. На данный момент мы придерживаемся одной целевой функции и одного приближения с низкой точностью.

Рассмотрим рис. 9.2, где в дополнение к функции Форрестера (сплошная линия) у нас также есть ее приближение с низкой точностью (пунктирная линия). Несмот-

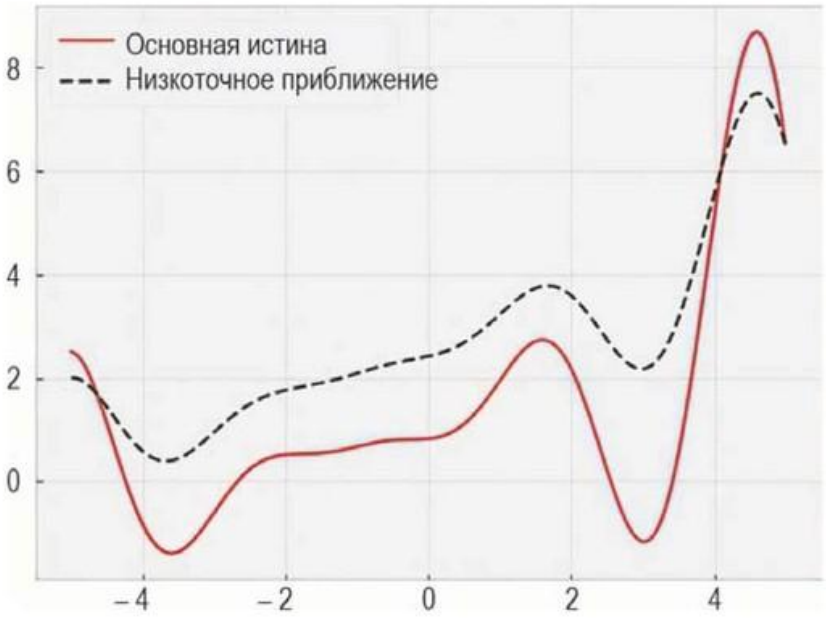


Рис. 9.2. Функция Форрестера (сплошная линия) и ее низкоточное приближение (пунктирная линия): несмотря на то что низкоточное приближение не совсем соответствует истине, оно дает достаточно информации о цели, поскольку обе функции имеют примерно одинаковую форму

ря на то что приближение не совсем соответствует истине, оно отражает общую ее форму и, следовательно, может быть полезно в поиске оптимума.

Поскольку данное приближение достаточно информативно, мы могли бы многократно его запрашивать, чтобы изучить поведение основной истины (функции Форрестера) в пространстве поиска, запрашивая при этом целевую функцию лишь тогда, когда нужно «обнулить» искомый оптимум. Наша задача в этой главе — разработать политику, которая будет управлять подобным поиском и решать, где и какую функцию запрашивать с целью максимально быстро и дешево оптимизировать цель.

Разноточный цикл БО представлен на рис. 9.3 вместе с изменениями по сравнению с традиционным циклом на рис. 1.6:

- На шаге 1 ГП обучается на данных из обоих источников: высокоточная истинная функция и низкоточное приближение. Иначе говоря, обучающие данные разделены на два набора — точки, оцененные на основе $f(x)$ и $\tilde{f}(x)$. Обучение на обоих наборах гарантирует, что прогностическая модель сможет рассуждать о целевой функции в регионах, где есть низкоточные данные, но нет высокоточных.

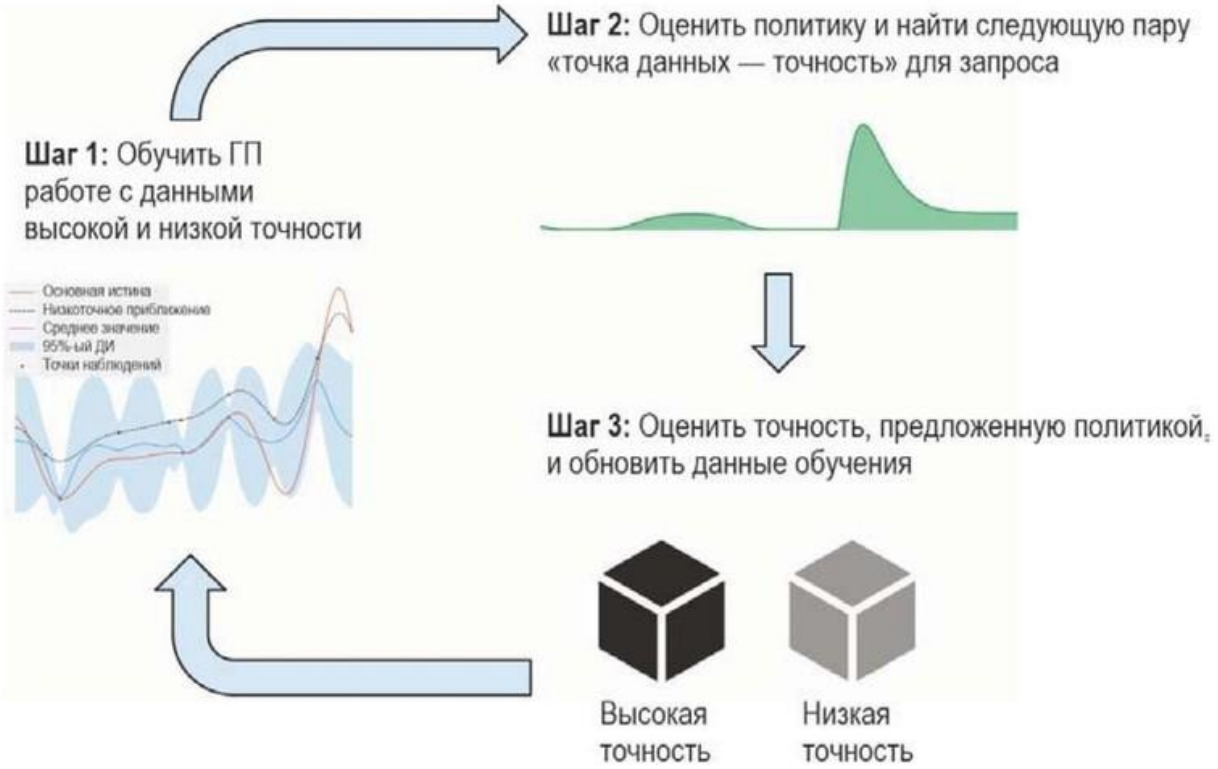


Рис. 9.3. Разноточный цикл БО: ГП обучается на данных как из целевой функции высокой точности, так и из низкоточного приближения, а политика решает, где и что в них запрашивать на каждой итерации цикла

- На шаге 2 политика генерирует оценку приобретения для каждой из точек в пространстве поиска, чтобы количественно оценить их значение и помочь определить оптимум. Однако оцениваются не только сами точки, но и пары «точка данных — точность», а именно политика количественно определяет ценность самого запроса каждой точки для конкретной функции (высокоточной истины

или ее низкоточного приближения). Данная оценка должна сбалансировать оптимизацию цели и стоимость запроса.

- На шаге 3 мы запрашиваем точку из соответствующей пары, точность которой максимизирует оценку приобретения политики. Затем обновляем наборы обучающих данных новыми наблюдениями и возвращаемся к шагу 1, чтобы продолжить процедуру БО.

Далее мы рассмотрим компоненты разноточного цикла БО, а также как их реализовать на Python. Начнем с обучения ГП на наборе данных, который включает в себя наблюдения обеих функций.

9.2. Разноточное моделирование с помощью ГП

Как отмечено на рис. 9.3, модель ГП обучена на комбинированном наборе данных, который включает в себя наблюдения с разными уровнями точности. Это позволяет ей делать прогнозы относительно целевой функции даже в регионах, где есть только низкоточные наблюдения, и впоследствии информировать политику для принятия решений, связанных с оптимизацией. Далее мы узнаем, как представлять набор разноточных данных и обучать на нем ГП. Код, использованный в этой главе, включен в `CH09/01 – Multifidelity modeling.ipynb`.

9.2.1. Форматирование разноточного набора данных

Чтобы настроить задачу разноточной оптимизации, мы используем следующий код для одномерной целевой функции Форрестера и ее низкоточного приближения, показанного на рис. 9.2 (пространство поиска находится между –5 и 5):

```
#Истинная целевая функция
def objective(x):
    y = -((x + 1) ** 2) * torch.sin(2 * x + 2) / 5 + 1 + x / 3
    return y

#Низкоточное приближение целевой функции
def approx_objective(x):
    return 0.5 * objective(x) + x / 4 + 2

#Границы пространства поиска, которые позже
#будут использоваться политиками оптимизации
lb = -5
ub = 5
bounds = torch.tensor([[lb], [ub]], dtype=torch.float)
```

Особое значение имеет PyTorch-тензор, который хранит информацию о корреляции между каждой функцией точности, к которой у нас есть доступ, и истинной целевой функцией, которую мы стремимся максимизировать. Предполагается, что нам известны значения этих корреляций, и точность тензора объявляется следующим образом:

```
fidelities = torch.tensor([0.5, 1.0])
```

Он имеет два аргумента, которые соответствуют двум доступным нам значениям точности:

- «0,5» — используется для обозначения корреляции между функцией Форрестера $f(x)$ и ее низкоточным приближением $\overline{f}(x)$. Обозначены сплошными и пунктирными линиями на рис. 9.2.
- «1» — соответствует корреляции функции Форрестера к самой себе.

Эти значения важны, поскольку они сообщают, насколько ГП (который мы будем обучать позже) должен полагаться на данные определенной точности:

- Если корреляция между основной функцией и приближением высока, то второе дает достаточно информации о цели. Крайним случаем этого является сама целевая функция непосредственно. Она дает максимально точную информацию, поскольку является самым точным приближением к самой себе и, следовательно, имеет значение корреляции, равное 1.
- Низкоточное приближение со значением корреляции 0,5 дает неточную, но все же ценную информацию.
- И наоборот, приближение со значением корреляции 0 ничего не говорит о целевой функции. Примером является идеально горизонтальная линия, поскольку она постоянна во всей области.

Рис. 9.4 иллюстрирует данную шкалу корреляции: чем выше ее значение, тем больше информации об основной истинной дает низкоточное приближение.

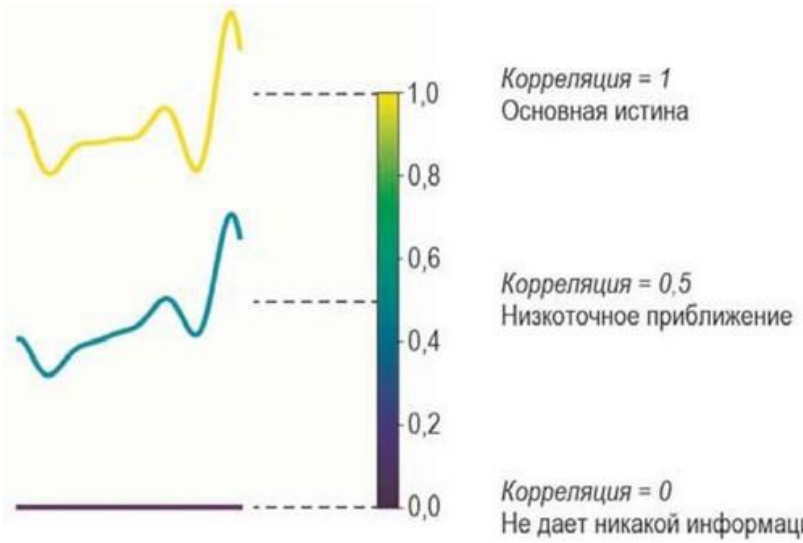


Рис. 9.4. Шкала корреляции от 0 до 1 между низкоточным приближением и истинной целью: чем выше значение, тем больше информации дает приближение

Настройка переменной `fidelities`

Переменная `fidelities` — это тензор с k элементами, где k — количество функций, которые можно запросить, включая целевую. Элементами являются числа от 0 до 1, обозначающие корреляцию между приближенными функциями и основной целью. Для последующих задач обучения и принятия решений более удобно иметь 1 (корреляцию истинной цели по отношению к самой себе) в конце тензора.

К сожалению, не существует конкретных правил относительно того, как устанавливать данные значения точности. Решение остается за инженером. Если в вашем собственном варианте использования они не известны, можно сделать приблизительную оценку на основе рис. 9.4, определив, где низкоточное приближение находится между целевой функцией (основная истина) и неинформативным источником данных.

Имея функции и значения корреляции, теперь мы можем создать пример набора обучающих данных. Сначала случайным образом выберем 10 местоположений внутри пространства поиска и сохраним их как тензор в `train_x`:

```
#Размер обучающего набора
n = 10

#Исправляем случайное начальное число для воспроизводимости
torch.manual_seed(0)

#Равномерно рисуем точки из пространства случайным образом
train_x = bounds[0] + (bounds[1] - bounds[0]) * torch.rand(n, 1)
```

Тензор `train_x` имеет 10 строк и 1 столбец, поскольку у нас есть 10 точек данных в одномерном пространстве. Каждая из точек связана с точностью, с которой происходит наблюдение. Мы кодируем эту информацию в нашем наборе данных, добавляя дополнительный столбец в `train_x`, чтобы указать точность каждой точки, как показано на рис. 9.5.

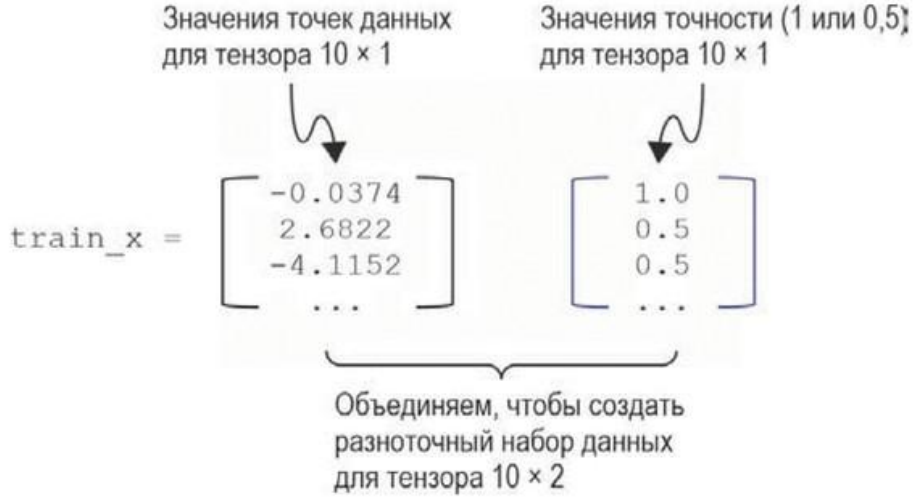


Рис. 9.5. Форматирование объектов в наборе разноточных данных: каждая точка связана с соответствующей точностью; значения точностей сохраняются в дополнительном столбце обучающего набора

Примечание

Следует помнить, что мы стремимся обучить ГП на данных из обоих источников. Поэтому мы случайным образом назначаем каждую из 10 имеющихся точек данных для той или иной степени точности.

Мы используем `torch.randint(2)` для случайного выбора целого числа от 0 (включительно) до 2 (не включая), фактически выбирая между 0 и 1. Это число определяет, из какой функции поступает каждая точка данных: 0 означает, что точка оценива-

ется по низкоточному приближениию $\overline{f}(x)$; 1 означает, что точка оценивается по целевой функции $f(x)$. Затем мы извлекаем соответствующие значения корреляции в `fidelities` для каждой точки и объединяем этот массив с обучающими данными:

```
#Случайным образом выбираем точность каждой точки данных
#(и, следовательно, значение корреляции)
train_f = fidelities[torch.randint(2, (n, 1))]
#Добавляем значения корреляции к обучающим данным
train_x_full = torch.cat([train_x, train_f], dim=1)
```

Взглянув на полные данные обучения `train_x_full`, мы можем наблюдать первые две точки данных:

```
tensor([[-0.0374, 1.0000], #Первая точка данных оценивается по f(x)
        [ 2.6822, 0.5000], #Вторая точка данных оценивается по f¯(x)
        ...
```

Первый столбец `train_x_full` содержит местоположения точек между –5 и 5, а второй — значения корреляции. Данный вывод означает, что первая точка обучения находится на уровне –0,0374 и оценивается по $f(x)$. С другой стороны, вторая точка находится на отметке 2,6822 и оценивается уже по $\overline{f}(x)$.

Теперь нужно соответствующим образом сгенерировать наблюдения `train_y`, чтобы они вычислялись с помощью корректной функции: первый элемент `train_y` равен $f(-0{,}0374)$, второй элемент равен $\overline{f}(2{,}6822)$ и т. д. Для этого напишем вспомогательную функцию, которая принимает полный обучающий набор, где последний столбец содержит значения корреляции, и вызовем ее для генерации `train_y`. Если значение корреляции равно 1, мы вызываем `objective()`, которая соответствует $f(x)$, как было определено ранее. Если значение корреляции равно 0,5, мы вызываем `approx_objective()` для $\overline{f}(x)$:

```
def evaluate_all_functions(x_full):
    y = []
    #Перебираем точки данных
    for x in x_full:
        #Запрашиваем f(x), если значение корреляции равно 1
        if torch.isclose(x[-1], torch.ones(1)):
            y.append(objective(x[:-1]))
        #Запрашиваем f¯(x), если значение корреляции равно 0,5
        else:
            y.append(approx_objective(x[:-1]))

    #Изменяем форму тензора наблюдения, чтобы он имел правильную форму
    return torch.tensor(y).unsqueeze(-1)
```

Вызов `evaluate_all_functions()` для `train_x_full` дает наблюдаемое значение `train_y`, оцененное в соответствующих функциях. Наш обучающий набор представлен на рис. 9.6, где показаны три наблюдения с высокой точностью и семь наблюдений с низкой точностью.

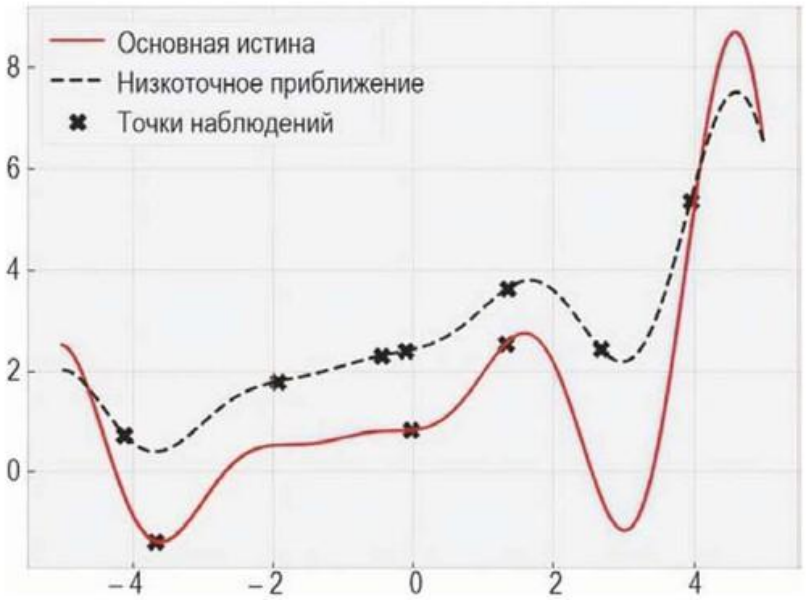


Рис. 9.6. Случайный разноточный набор обучающих данных на основе функции Форрестера и ее низкоточного приближения: он содержит три наблюдения с высокой точностью и семь наблюдений с низкой точностью

Таким образом мы генерируем и форматируем обучающий набор в разноточной байесовской оптимизации. Наша следующая задача — обучить ГП на этом наборе так, чтобы можно было использовать как целевую функцию, так и ее низкоточное приближение.

9.2.2. Обучение разноточного ГП

Наша цель в этом разделе — создать ГП, который принимает набор разноточных наблюдений и выдает вероятностные прогнозы относительно целевой функции $f(x)$.

Из раздела 2.2 нам известно, что ГП — это MVN-распределение (нормальное распределение) бесконечного числа переменных. Гауссов процесс моделирует ковариацию (и, следовательно, корреляцию) между любой парой переменных, используя функцию ковариации. Именно благодаря этому между любыми двумя переменными ГП может делать прогнозы относительно одной переменной при наблюдении значения другой.

Напоминание о корреляции и обновленное представление о переменной

Предположим, есть три переменные A, B и C, совместно смоделированные с помощью трехмерного распределения Гаусса, где корреляция между A и B высокая, а между A и C, а также между B и C — низкая.

При наблюдении значения A неопределенность в обновленном убеждении о B (представленном как апостериорное распределение значения B) значительно уменьшается. Это связано с тем, что корреляция между A и B высока, поэтому наблюдение за одним значением дает много информации о другом. Однако это не относится к C, поскольку корреляция между A и C низкая. Поэтому обновленное представление об C все еще имеет высокую неопределенность.

Как нам известно из раздела 2.2.2, пока есть способ смоделировать корреляцию между любой парой переменных (значениями функции в любых двух заданных ме-

стоположениях), можно соответствующим образом обновить ГП, чтобы отразить обновленное убеждение о функции в любом месте пространства. Это остается справедливым и при условии разноточности: пока есть способ смоделировать корреляцию между двумя наблюдениями, даже если одно из них имеет высокую точность, а другое — низкую, мы можем обновить ГП по целевой функции $f(x)$.

Все, что остается сделать, — использовать ковариационную функцию, способную вычислить ковариацию между двумя заданными наблюдениями, которые могут иметь или не иметь одинаковой точности. К счастью, BoTorch предлагает модифицированную версию ядра Матерна, которая учитывает значение точности, связанное с каждой точкой данных в обучающем наборе:

- Если значение велико, ядро создаст высокую ковариацию между наблюдаемой и любой близлежащей точками данных, что позволит уменьшить неопределенность ГП с помощью информативного наблюдения.
- Если значение корреляции — низкое, ядро выдаст низкую ковариацию, а апостериорная неопределенность останется высокой.

> **Примечание**
>
> Впервые в этой книге ядро Матерна упоминалось в разделе 3.4.2. Несмотря на то что мы не будем здесь вдаваться в подробности разноточности ядра Матерна, вы можете найти дополнительную информацию в документации к BoTorch.

Поскольку ГП с разноточным ядром реализован как специальный класс, можно импортировать его из BoTorch без необходимости создавать собственную реализацию. В частности, этот ГП является экземпляром класса `SingleTaskMultiFidelityGP`, который принимает разноточный обучающий набор `train_x_full` и `train_y`. Инициализация также имеет аргумент `data_fidelity`, которому должен быть присвоен индекс столбца в `train_x_full`, содержащий значения корреляции (в нашем случае это 1):

```
#Импортируем реализацию класса ГП
from botorch.models.ГП_regression_fidelity import SingleTaskMultiFidelityGP

#Инициализируем многоточный ГП
model = SingleTaskMultiFidelityGP(train_x_full, train_y, data_fidelity=1)
```

После инициализации модели нужно обучить ее, максимизируя вероятность наблюдаемых данных (подробнее о том, почему мы решили максимизировать вероятность обучения ГП, можно посмотреть в разделе 3.3.2). Поскольку имеющийся ГП является экземпляром специального класса BoTorch, можно воспользоваться вспомогательной функцией `fit_gpytorch_mll()`. Это облегчит обучающий процесс «за кулисами». Все, что нужно сделать, — это инициализировать вероятностный объект (логарифм) в качестве цели обучения и передать его вспомогательной функции:

```
#Импортируем логарифмическую вероятностную целевую функцию и
#вспомогательную функцию для обучения
from gpytorch.mlls.exact_marginal_log_likelihood import
  ExactMarginalLogLikelihood
from botorch.fit import fit_gpytorch_mll
```

```
#Инициализируем логарифмическую вероятностную целевую функцию
mll = ExactMarginalLogLikelihood(model.likelihood, model)
#Обучаем ГП, чтобы максимизировать логарифмическую вероятность
fit_gpytorch_mll(mll);
```

Эти несколько строк кода — все, что нужно для обучения разноточного ГП на наборе наблюдений.

Предупреждения BoTorch о типе данных и масштабировании

При запуске предыдущего кода новые версии GPyTorch и BoTorch могут отображать два предупреждения, первое из которых:

```
UserWarning: The model inputs are of type torch.float32. It is strongly
recommended to use double precision in BoTorch, as this improves both
precision and stability and can help avoid numerical errors. See
https://github.com/pytorch/botorch/discussions/1444
 warnings.warn(
```

Оно указывает на то, что нам следует использовать для `train_x` и `train_y` другой тип данных, отличный от `torch.float32` (по умолчанию), чтобы повысить числовую точность и стабильность. Для этого можно добавить в код (в начале скрипта) следующее:

```
torch.set_default_dtype(torch.double)
```

Второе предупреждение касается масштабирования входных признаков `train_x` так, чтобы они находились в единичном кубе (каждое значение признака находится в диапазоне от 0 до 1), а значения `train_y` должны быть стандартизированы, чтобы иметь нулевое среднее значение и единичную дисперсию:

```
InputDataWarning: Input data is not
 contained to the unit cube. Please consider min-max scaling the input data.
 warnings.warn(msg, InputDataWarning)
InputDataWarning: Input data is not standardized. Please consider scaling
the input to zero mean and unit variance.
 warnings.warn(msg, InputDataWarning)
```

Масштабирование `train_x` и `train_y` таким образом помогает подогнать ГП более легким и численно стабильным методом. Чтобы сохранить код простым, мы не будем здесь реализовывать такое масштабирование и отфильтруем эти предупреждения с помощью модуля `warnings`. Более подробную информацию можно найти в упражнении к главе 2.

Далее для проверки, способен ли ГП обучиться на наборе, визуализируем его прогнозы относительно $f(x)$ между –5 и 5 с помощью среднего значения и 95%-ного ДИ.

Тестовый набор `xs` представляет собой плотную сетку (более 200 элементов) между –5 и 5:

```
xs = torch.linspace(-5, 5, 201)
```

В отличие от предыдущих глав, нам нужно дополнить данный тестовый набор дополнительным столбцом, обозначающим точность, на основе которой мы будем прогнозировать. Другими словами, тестовый набор `xs` должен иметь тот же формат, что и обучающий набор `train_x_full`. Поскольку мы заинтересованы в предсказаниях

ГП относительно $f(x)$, добавляем дополнительный столбец, заполненный единицами (поскольку 1 — это значение корреляции $f(x)$):

```
with torch.no_grad():  #Отключаем отслеживание градиента
 #Дополняем тестовый набор столбцом точности и передаем его в модель
 pred_dist = model(torch.vstack([xs, torch.ones_like(xs)]).T)
 #Вычисляем средние значения прогнозов
 pred_mean = pred_dist.mean
 #Вычисляем 95%-ный ДИ
 pred_lower, pred_upper = pred_dist.confidence_region()
```

Прогнозы визуализированы на рис. 9.7, который иллюстрирует ряд важных особенностей данного разноточного ГП:

1. Средние прогнозы относительно $f(x)$ проходят через высокоточные наблюдения примерно при –3,6, 0 и 1,3. Такая интерполяция имеет смысл, поскольку эти точки данных действительно были оценены по $f(x)$.



Рис. 9.7. Предсказания разноточного ГП относительно целевой функции (основная истина): средние прогнозы соответствующим образом учитывают высокоточные наблюдения, но в отношении низкоточных наблюдений неопределенность по-прежнему снижается

2. В регионах, где есть низкоточные наблюдения, но нет высокоточных (например, при –2 и около 2,7), неопределенность относительно $f(x)$ все еще имела тенденцию к уменьшению. Это связано с тем, что наблюдения с низкой точностью предоставляют информацию об $f(x)$, даже если они не оценивались по ней.
3. Среди этих низкоточных наблюдений видно, что точка данных под номером 4 может предоставить ценную информацию для политики оптимизации, поскольку она фиксирует тенденцию роста целевой функции в этом регионе. Используя

эту информацию, политика оптимизации может обнаружить ближайший глобальный оптимум (около 4,5).

На рис. 9.7 показано, что ГП успешно обучился на наборе разноточных данных. Чтобы максимально расширить способность учиться на наблюдениях с низкой точностью и прогнозировать $f(x)$, можно изменить способ создания обучающего набора, чтобы он содержал только низкоточные наблюдения. Это можно сделать, устанавливая значения корреляции в дополнительном столбце переменной `train_x_full` равными 0,5:

```
train_f = torch.ones_like(train_x) * fidelities[0]
train_x_full = torch.cat([train_x, train_f], dim=1)
```

Повторный запуск последней части кода показан на левой части рис. 9.8, где видно, что все точки данных действительно взяты из низкоточного приближения $\overline{f}(x)$. По сравнению с рис. 9.7 здесь наша неуверенность в предсказаниях стала больше, что вполне уместно, поскольку, принимая во внимание только низкоточные наблюдения, ГП не может узнать достаточно много о целевой функции $f(x)$.

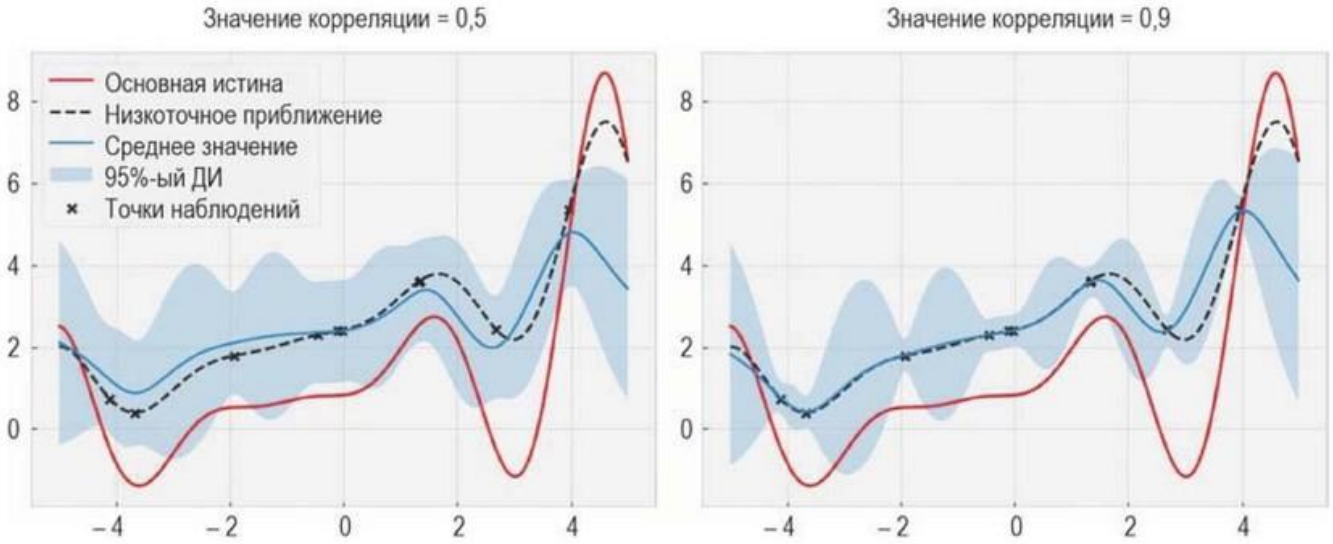


Рис. 9.8. Прогнозы ГП относительно целевой функции (основная истина), основанные только на наблюдениях с низкой точностью: слева показан результат, когда значение корреляции равно 0,5; справа — когда значение корреляции равно 0,9, что демонстрирует меньшую неопределенность

Чтобы дополнительно продемонстрировать гибкость разноточного ГП, можно попробовать поиграть со значениями корреляции, хранящимися в `fidelities` (при условии, что мы знаем, как правильно их установить). Как известно из раздела 9.2.1, первый элемент в этом тензоре обозначает корреляцию между $f(x)$ и $\overline{f}(x)$, что примерно означает, насколько ГП должен «доверять» наблюдениям с низкой точностью. Установив для него значение 0,9 (в отличие от 0,5, как в текущем случае), можно придать большую значимость наблюдениям с низкой точностью. Так мы говорим ГП, чтобы он мог извлечь больше информации из низкоточных данных, поскольку они предлагают много информации об $f(x)$. На правой части рис. 9.8 показан итоговый ГП, где неопределенность действительно ниже, чем на левой части.

Помимо гибкости разноточной модели ГП, рис. 9.8 также показывает, насколько важно иметь правильные значения корреляции в тензоре `fidelities`. Сравнивая обе

части рис. 9.8, можно увидеть, что 0,5 — лучшее значение для корреляции между двумя показателями точности, чем 0,9:

- Справа прогнозы не соответствуют истинной цели $f(x)$ в большей части пространства из-за чрезмерной уверенности и доверия к низкоточным наблюдениям.
- Слева 95%-ные ДИ соответственно шире, чтобы отразить неопределенность относительно $f(x)$.

Другими словами, мы стараемся не переоценить, насколько информативным является низкоточное приближение $\overline{f}(x)$ относительно цели $f(x)$.

На этом этапе мы уже научились моделировать функцию с помощью разноточного ГП. Далее в оставшейся части главы обсудим вторую часть задачи — принятие решений. В частности, мы изучим как разработать политику разноточной оптимизации, которая выбирает, в каком месте и какую функцию запрашивать на каждом этапе цикла БО.

9.3. Баланс между информацией и затратами в разноточной оптимизации

Чтобы найти компромисс между информативностью запроса и стоимостью его выполнения, необходим способ моделирования и обоснования последнего. Далее мы рассмотрим, как представить стоимость запроса заданной точности с помощью линейной модели. Используя ее, затем мы реализуем разноточную политику, которая обеспечит компромисс между затратами и достижением прогресса в оптимизации. Используемый код хранится в `CH09/02 – Multi-FidelityOptimization.ipynb`.

9.3.1. Моделирование затрат на запросы различной точности

В задаче разноточной оптимизации предполагается, что нам известно, сколько стоит запросить каждую из функций, к которым есть доступ: либо сама целевая функция $f(x)$, либо ее низкоточные приближения $\overline{f}(x)$. Облегчить рабочий процесс такой оптимизации можно, представив информацию о стоимости запроса каждой функции в виде модели затрат. Она принимает заданную точку данных (с дополнительной функцией, содержащей значение корреляции, как мы наблюдали в разделе 9.2.1) и возвращает известную стоимость запроса этой точки с заданной точностью.

> **Примечание**
>
> Поскольку стоимость запроса точности нам известна, в модели затрат нет никакого прогнозирования. Нам просто нужна формулировка модели, чтобы сохранить процедуру оптимизации, о которой мы узнаем позже.

BoTorch предоставляет такую реализацию класса для модели линейной стоимости. Она называется `AffineFidelityCostModel` и находится в модуле `botorch.models.cost`. Эта модель предполагает, что затраты на запрос подчиняются соотношениям, показанным на рис. 9.9, где стоимость запроса точки данных о точности линейно масшта-

бируется в зависимости от корреляции между этой точностью и основной истиной $f(x)$. Наклон данной линейной закономерности — это весовой параметр на рис. 9.9, а стоимость любого запроса фиксирована.



Рис. 9.9. Линейная модель затрат для разноточной оптимизации: стоимость запроса точности линейно зависит от корреляции между этой точностью и основной истиной $f(x)$

Модель инициализируется с помощью кода ниже, где устанавливается фиксированная стоимость 0 и вес 1. Это означает, что запрос точки данных с низкой точностью будет стоить столько же, сколько и значение корреляции для низкоточного приближения, которое равно 0,5. Аналогично запрос к точке данных с высокой точностью будет стоить 1. Здесь аргумент `fidelity_weights` принимает словарь, который сопоставляет вес с индексом столбца, содержащим значение корреляции в `train_x_full` (1 в данном случае):

```
from botorch.models.cost import AffineFidelityCostModel

cost_model = AffineFidelityCostModel(
    #Фиксированная стоимость запроса
    fixed_cost=0.0,
    #Линейный вес, который нужно умножить на значения корреляции
    fidelity_weights={1: 1.0},
)
```

> **Примечание**
>
> Используемая единица измерения стоимости зависит от конкретного приложения. Затраты сводятся к разнице в «удобстве» между запросом цели и запросом приближения с низкой точностью. Это могут быть время (минуты, часы или дни), деньги (доллары) или некоторая мера усилий. И они должны быть установлены пользователем.

Линейная тенденция отражает взаимосвязь между значением корреляции и стоимостью: высокоточная функция с высоким значением корреляции должна иметь высокую стоимость запроса, а низкоточная — более низкую. Два настраиваемых параметра (фиксированная стоимость и вес) позволяют гибко моделировать многие типы затрат на запросы. В следующем разделе мы узнаем, как разные типы затрат могут привести к принятию разных решений. Имея на руках модель затрат, теперь можно попробовать сбалансировать затраты и прогресс в задаче разноточной оптимизации.

Моделирование стоимости нелинейных запросов

В этой главе мы используем только линейную модель затрат. Если вам необходимо, чтобы она моделировалась нелинейным способом (например, квадратичным или экспоненциальным), вы можете реализовать свою собственную модель затрат.

Это можно сделать, расширив используемый класс `AffineFidelityCostModel` и переписав его метод `forward()`. Реализация класса `AffineFidelityCostModel` показана в официальной документации BoTorch, где можно увидеть, что метод `forward()` осуществляет линейную зависимость между стоимостью запроса и значением корреляции, как на рис. 9.9:

```
def forward(self, X: Tensor) -> Tensor:
    #Умножаем значения корреляции на вес
    lin_cost = torch.einsum(
        "...f,f", X[..., self.fidelity_dims], self.weights.to(X)
    )
    #Добавляем фиксированную стоимость
    return self.fixed_cost + lin_cost.unsqueeze(-1)
```

Затем в новом классе для пользовательской модели затрат можно переписать метод `forward()`, чтобы реализовать необходимую связь между стоимостью запроса и значением корреляции. Даже при наличии пользовательской модели затрат остальную часть кода, используемую в этой главе, не нужно изменять, что демонстрирует преимущества модульной конструкции BoTorch.

9.3.2. Оптимизация количества информации для управления оптимизацией

Вернемся к вопросу, поставленному в начале этой главы: как следует сбалансировать количество информации при запросе функции и стоимость этого запроса? В разноточной оптимизации высокоточная функция (основная истина) предлагает точную информацию о целевой функции $f(x)$ для ее оптимизации, но стоимость запроса при этом высока. С другой стороны, низкоточное приближение стоит недорого, но может предоставить неточную информацию о $f(x)$. Задача разноточной политики БО — решить, как осуществлять такую балансировку.

У нас уже есть модель из раздела 9.3.1, которая вычисляет стоимость запроса любой заданной точки данных. Что касается другого компонента, нам нужен метод количественной оценки, сколько информации мы получим о целевой функции из данного запроса.

> **Примечание**
>
> Сколько информации об $f(x)$ или, точнее, об оптимуме $f(x)$, мы получим в результате запроса, это именно то, что политика энтропийного поиска (MES) использует для ранжирования своих запросов. Она выбирает запрос, который дает больше всего информации о наивысшем значении $f(x)$ в режиме одинарной точности, где можно запрашивать только целевую функцию.

Поскольку такой способ получения сведений является общей концепцией теории информации, его также можно применять в условиях разноточности. Другими словами, MES используется в качестве базовой политики для вычисления, сколько информации об оптимальном значении $f(x)$ мы получаем с каждым запросом во время оптимизации. Теперь, когда доступны оба компонента (стоимость и количество информации), мы можем разработать способ их сбалансировать, что даст экономически обоснованную меру полезности запроса.

Возврат инвестиций

Для количественной оценки полезности запроса с учетом затрат используется общий показатель, называемый *прибылью от инвестиций* или *рентабельностью инвестиций* (**Return on Investment**, **ROI**). Он рассчитывается путем деления прибыли на инвестиционные затраты. В контексте использования MES прибыль — это объем информации, полученной в результате запроса точки данных, а затраты — это стоимость самого запроса.

Следует помнить, что оценка приобретения — это оценка, которую политика присваивает каждой точке данных в пространстве поиска, чтобы количественно оценить ее полезность, помогая оптимизировать целевую функцию. С помощью оценки приобретения ROI каждая точка данных оценивается по объему информации, которую она предоставляет об оптимуме целевой функции для каждой единицы стоимости. Этот расчет представлен на рис. 9.10.



Рис. 9.10. Формула оценки приобретения для рентабельности инвестиций в разноточной оптимизации: она количественно определяет объем информации, которую предоставляет запрос об оптимуме целевой функции для каждой единицы затрат

Из рисунка видно, что оценка приобретения ROI является подходящей мерой, которая «взвешивает» информацию от запроса в зависимости от того, сколько стоит выполнить этот запрос:

- Если два запроса имеют одинаковую стоимость, но приводят к разным информационным выгодам, следует выбрать тот, который предлагает больше информации.
- Если два запроса предоставляют одинаковое количество информации, следует выбрать тот, создание которого обходится дешевле.

Данный компромисс позволяет получить информацию об $f(x)$ из недорогих приближений с низкой точностью, если они действительно информативны. С другой стороны, когда низкоточные запросы перестают давать достаточно информации, мы переключаемся на точки данных с более высокой точностью. По сути, мы всегда выбираем оптимальное решение с учетом затрат, которое дает наибольшую отдачу от вложенных средств.

Чтобы реализовать данный экономичный вариант MES, можно использовать реализацию класса `qMultiFidelityMaxValueEntropy` от BoTorch. Она требует ряда компонентов, которые принимает в качестве аргументов:

- *Объект полезности затрат, который выполняет расчет ROI*, как показано на рис. 9.10. Он реализован с помощью класса `InverseCostWeightedUtility`, который

измеряет полезность запроса по величине, обратной его стоимости. Инициализация использует модель затрат, созданную ранее:

```
from botorch.acquisition.cost_aware import InverseCostWeightedUtility

cost_aware_utility = InverseCostWeightedUtility(cost_model=cost_model)
```

- *Последовательность Соболя, которая действует как набор кандидатов для вычисления энтропии MES.* Здесь процедура работы с последовательностью Соболя та же самая, что и в разделе 6.2.2: создаем последовательность порядка 1000 элементов из единичного куба (сегмент от 0 до 1, в нашем случае он одномерный) и масштабируем его до нашего пространства поиска. Еще одно действие, которое нужно сделать при разноточности, — это дополнить набор кандидатов дополнительным столбцом со значением корреляции 1 (соответствующим целевой функции $f(x)$), чтобы обозначить, что нужно измерить энтропию именно в целевой функции:

```
#Исправляем случайное начальное число для воспроизводимости
torch.manual_seed(0)

#Рисуем 1000 точек из последовательности Соболя внутри единичного куба
sobol = SobolEngine(1, scramble=True)
candidate_x = sobol.draw(1000)

#Масштабируем выборки на пространство поиска
candidate_x = bounds[0] + (bounds[1] - bounds[0]) * candidate_x

#Дополняем выборки индексом основной истины
candidate_x = torch.cat([candidate_x, torch.ones_like(candidate_x)], dim=1)
```

- *Вспомогательная функция, которая проецирует заданную точку данных с любой точностью до основной истины.* Этот прогноз необходим для расчета энтропии, которую политика использует при расчете оценок приобретения. Здесь BoTorch предоставляет вспомогательную функцию `project_to_target_fidelity`, которая не требует дальнейшей параметризации, если последний столбец в обучающем наборе содержит значения корреляции, а значение корреляции основной истины равно 1 (и оба они верны в нашем коде).

Используя компоненты, реализуем нашу разноточную политику MES следующим образом:

```
from botorch.acquisition.utils import project_to_target_fidelity

policy = qMultiFidelityMaxValueEntropy(
    model,
    #Выборки из последовательности Соболя
    candidate_x,
    num_fantasies=128,
    #Объект затратной полезности, который измеряет полезность
    #по обратному значению стоимости
    cost_aware_utility=cost_aware_utility,
```

```
    #Вспомогательная проецирующая функция
    project=project_to_target_fidelity,
)
```

На этом этапе можно использовать объект политики для оценки каждой точки данных с любой степенью точности по ее значению с поправкой на стоимость, что помогает нам найти оптимум цели. Последняя часть пазла — это вспомогательная функция, которая используется для оптимизации оценки приобретения, чтобы найти точку с наивысшим показателем рентабельности инвестиций на каждой итерации поиска. В предыдущих главах мы использовали `optimize_acqf()` из модуля `botorch.optim.optimize` для оптимизации оценки приобретения при одинарной точности, которая работает только в том случае, когда пространство поиска непрерывно.

> **Примечание**
>
> В нашем текущем разноточном сценарии пространство поиска для запроса местоположения по-прежнему непрерывно, но выбор функции, с помощью которой выполняется запрос, является дискретным. Другими словами, пространство поиска является смешанным. К счастью, для такого случая BoTorch предлагает аналогичную вспомогательную функцию — `optimize_acqf_mixed()`.

В дополнение к обычным аргументам, которые принимает `optimize_acqf()`, новая вспомогательная функция (`optimize_acqf_mixed()`) имеет аргумент `fixed_features_list`. Он должен представлять собой список словарей, каждый из которых сопоставляет индекс дискретного столбца в `train_x_full` с возможными значениями в нем. В нашем случае есть только один дискретный столбец — последний. Он содержит значения корреляции, поэтому мы используем `[{1: cost.item()} for cost in fidelities]` в качестве аргумента `fix_features_list`. Кроме того, переменная `bounds`, которая обычно передается вспомогательной функции, теперь также должна содержать границы значений корреляции. В целом мы оптимизируем оценку приобретения разноточной политики MES так:

```
from botorch.optim.optimize import optimize_acqf_mixed

next_x, acq_val = optimize_acqf_mixed(
    policy,
    #Границы пространства поиска, в том числе для значений корреляции
    bounds=torch.cat(
        [bounds, torch.tensor([0.5, 1.0]).unsqueeze(-1)], dim=1
    ),
    #Дискретные значения, которые может содержать столбец корреляции
    fixed_features_list=[{1: cost.item()} for cost in fidelities],
    q=1,
    num_restarts=20,
    raw_samples=50,
)
```

Эта вспомогательная функция завершает код, который нужен для использования разноточной MES. Нижняя часть рис. 9.11 визуализирует оценки приобретения, вычисленные этой политикой с помощью разноточного ГП, который был обучен

в разделе 9.2.2. На нижней части рисунка граница затемненной области (обозначает оценки низкоточных запросов) превышает границу заштрихованной области (оценки высокоточных запросов). Это означает, что запросы с низкой точностью являются более экономически эффективными, чем запросы с высокой точностью, учитывая наши текущие знания. В конечном итоге лучший запрос, который мы делаем (отмечен звездочкой), имеет значение около 3,5 в низкоточном приближении $\overline{f}(x)$.

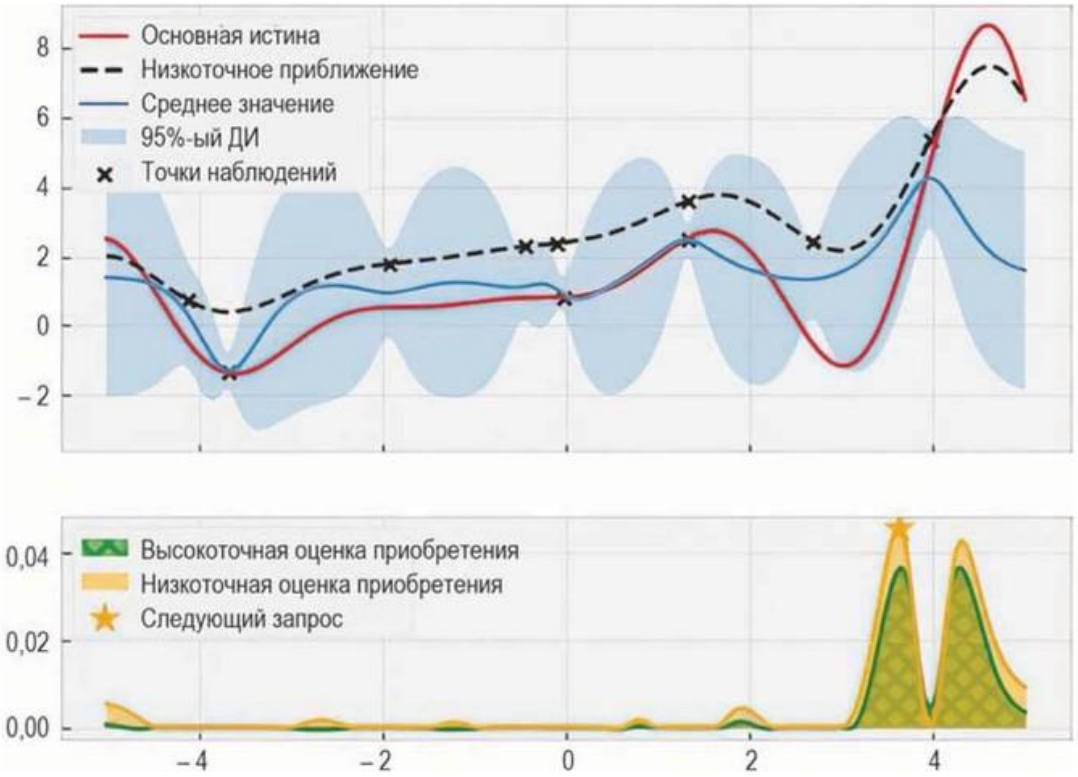


Рис. 9.11. Текущее предположение ГП о целевой функции (вверху) и оценках приобретения, рассчитанных с помощью разноточной политики MES (внизу): в данном примере низкоточные запросы предпочтительнее высокоточных из-за их низкой стоимости

Запрос с низкой точностью является оптимальным вариантом (рис. 9.11), поскольку он предлагает больше информации относительно своей стоимости, чем любой запрос с высокой точностью. Тем не менее так бывает не всегда. Изменив процедуру генерации данных (раздел 9.2.2) для создания обучающего набора из всех низкоточных наблюдений и перезапустив код, мы получим левую часть рис. 9.12, где на этот раз оптимальным решением будет запрос высокоточной функции $f(x)$. Это связано с тем, что, по мнению ГП, из низкоточных данных извлечено достаточно информации и пришло время проверить основную истину $f(x)$.

В качестве окончательного анализа поведения нашей политики можно изучить влияние запрашиваемых затрат на то, как принимаются решения. Для этого нужно изменить стоимость запроса. Таким образом, низкоточный запрос будет ненамного дешевле, чем запрос с высокой точностью, в частности, за счет увеличения фиксированной стоимости, описанной в разделе 9.3.1 (от 0 до 10). Это изменение означает, что запрос с низкой точностью теперь стоит 10,5 единицы, а запрос с высокой точностью теперь стоит 11 единиц. По сравнению с предыдущими издержками (0,5 и 1) новые затраты (10,5 и 11) гораздо ближе друг к другу, что делает знаменате-

ли на рис. 9.10 для двух вариантов почти равными. Это означает, что запрос $\overline{f}(x)$ стоит почти столько же, как и сама целевая функция $f(x)$. Учитывая такие затраты, правая часть рис. 9.12 показывает, как политика MES оценивает потенциальные запросы. На этот раз, поскольку высокоточные запросы ненамного дороже, чем низкоточные, первые более предпочтительны, поскольку они дают больше знаний об $f(x)$.

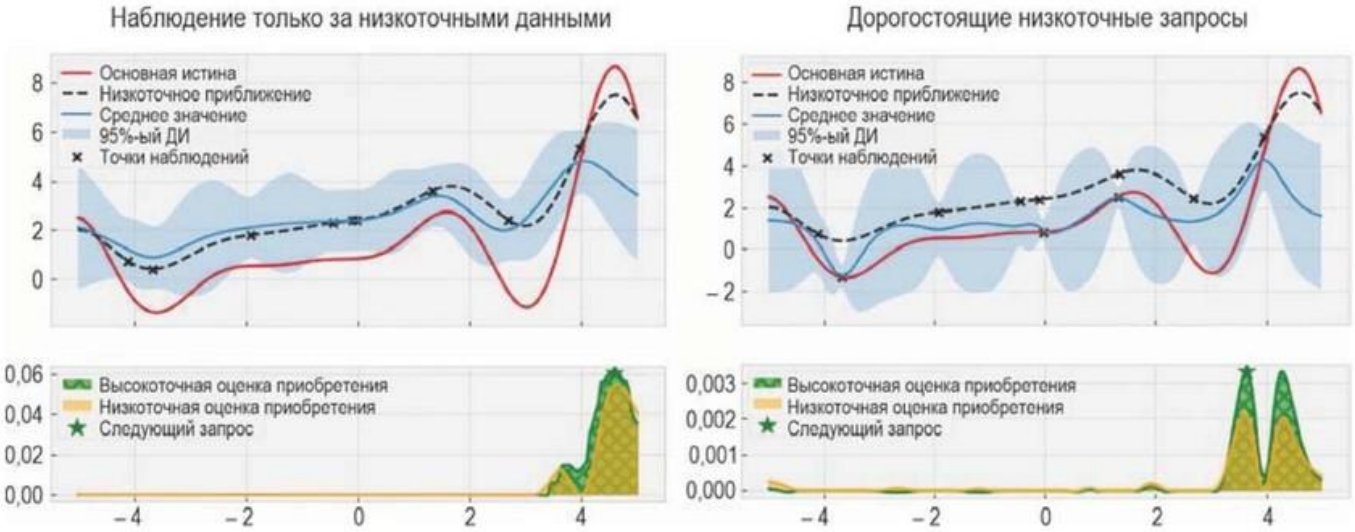


Рис. 9.12. Сценарий, когда высокоточные запросы предпочтительнее низкоточных: слева обучающий набор содержит только низкоточные наблюдения; справа — запросы с низкой точностью почти так же дороги, как высокоточные

Эти примеры показывают, что MES может определять оптимальное решение, которое соответствующим образом сбалансирует информацию и затраты. Политика присваивает более высокие оценки запросам с низкой точностью, когда их выполнение не требует больших затрат и может предоставить существенную информацию о целевой функции. С другой стороны, если запросы с высокой точностью либо значительно более информативны, либо почти не отличаются по стоимости, политика будет отдавать предпочтение им.

9.4. Измерение производительности в разноточной оптимизации

Наши рассуждения выше показывают, что разноточная MES способна принимать соответствующие решения при выборе между двумя вариантами для запроса. Но имеет ли она лучшую производительность, чем обычные политики БО, которые запрашивают только основную истину $f(x)$? И если да, то насколько она лучше? В этом разделе мы изучим, как оценить производительность политики в условиях разноточности, что требует дополнительного рассмотрения. Код к этой главе можно найти в `CH09/03 - Measuring Performance.ipynb`.

Примечание

Чтобы измерить прогресс оптимизации в предыдущих главах, мы записывали самые высокие целевые значения (они также называются *действующими значениями* или *действующими точками*), которые затем сохраняли в обучающем наборе на протя-

жении всего поиска. Если действующие значения, собранные политикой A, превышают значения, собранные политикой B, то говорят, что политика A более эффективна при оптимизации, чем B.

При разноточности запись действующих значений не работает. Во-первых, если бы нужно было записать их в обучающий набор, имеет смысл выбирать только точку данных высокой точности с меткой, имеющей самое высокое значение. Однако такая стратегия игнорирует любой вклад в изучение цели $f(x)$, который вносят запросы с низкой точностью. Рассмотрим два возможных сценария, представленных на рис. 9.13:

- В первом сценарии (слева) проведены три наблюдения с высокой точностью, и самое большое наблюдаемое значение составляет примерно 0,8. Объективно говоря, в этой ситуации нет никакого прогресса в оптимизации. Область, где x больше 0, даже не исследовалась.
- Во втором сценарии (справа) проведены только наблюдения с низкой точностью, поэтому запись действующего значения с высокой точностью непригодна. Однако можно увидеть, что здесь мы очень близки к нахождению оптимума целевой функции, поскольку запросы обнаружили пик функции возле 4,5.

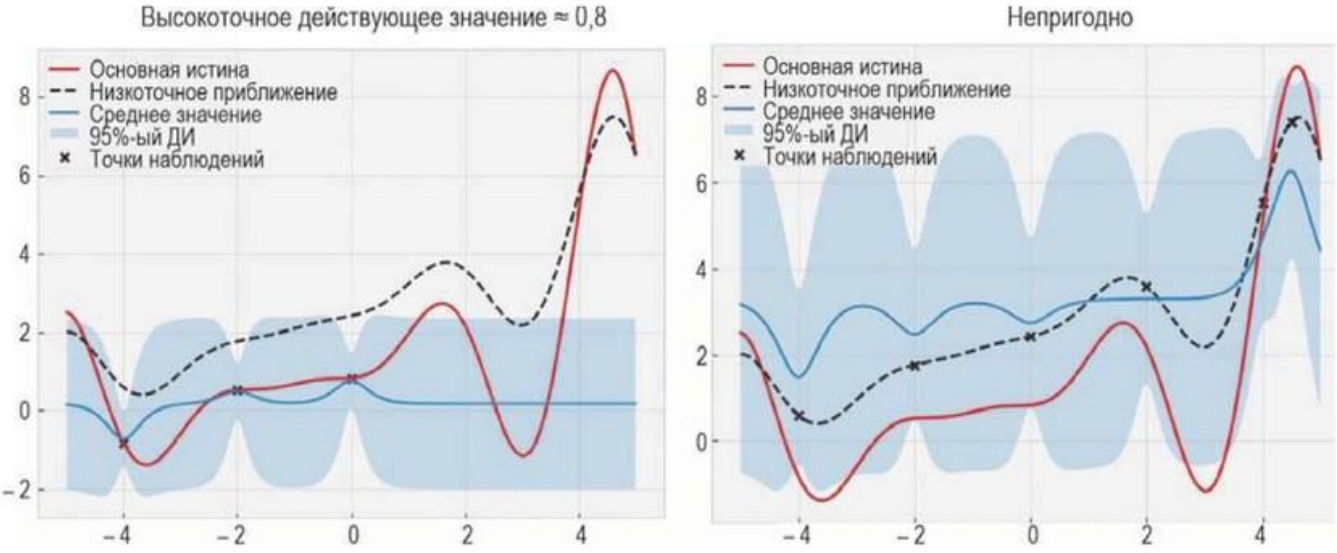


Рис. 9.13. Измерение производительности с помощью высокоточного действующего значения неприменимо при разноточной оптимизации: слева высокоточное действующее значение составляет примерно 0,8, а оптимум цели еще не обнаружен; справа мы близки к обнаружению оптимума, но нет высокоточного запроса, с помощью которого можно было бы записать действующее значение

Второй сценарий следует предпочесть первому, поскольку он показывает практический успех оптимизации, в то время как другой демонстрирует лишь незначительный прогресс. Однако использование высокоточного действующего значения в качестве меры прогресса не помогает провести различие между этими двумя сценариями. Нужен другой способ измерения прогресса.

Примечание

Распространенным показателем прогресса среди сообщества БО является значение целевой функции в той локации, которая в настоящее время дает наивысшее апостериорное среднее. Такая мера вполне отвечает на вопрос: если остановить работу БО

и рекомендовать точку в качестве решения задачи оптимизации, какую точку следует выбрать? Интуитивно следует выбрать точку, которая в наиболее вероятном сценарии (по мнению ГП) дает наибольшее значение, а именно *максимизатор* апостериорного среднего.

Максимизатор апостериорного среднего облегчает сравнение, которое было проведено на рис. 9.13, где слева он равен 0, что по-прежнему дает объективное значение 0,8 (максимизатор среднего обычно соответствует действующему значению при одноточном сценарии), а справа — около 4,5. Другими словами, такая метрика успешно помогает различить два сценария и показывает, что один предпочтительнее другого.

Чтобы реализовать максимизатор, создадим вспомогательную политику, которая использует апостериорное среднее в качестве оценки приобретения. Затем оптимизируем апостериорное среднее, так же как мы делаем с оценкой приобретения обычной политики, используя вспомогательную политику, которая состоит из двух компонентов:

- Реализация класса для политики, которая использует апостериорное среднее значение в качестве оценки приобретения. В данном случае это класс `PosteriorMean`, который можно импортировать из `botorch.acquisition`.
- *Политика-оболочка*, которая оптимизирует только высокоточные показатели. Она необходима, поскольку модель ГП является разноточной, и всегда нужно указывать, с какой точностью следует работать при передаче этой модели в политику оптимизации. Данная политика-оболочка реализована как экземпляр `FixFeatureAcquisitionFunction` из `botorch.acquisition.fixed_feature`.

В целом мы создаем метрику максимизации апостериорного среднего с помощью вспомогательной политики-оболочки, где та принимает экземпляр `PosteriorMean`, а остальные аргументы указываются следующим образом:

- *Размерность пространства поиска d = 2*. Фактическое пространство поиска является одномерным. Также есть дополнительное измерение для значений корреляции (точности запроса).
- *Индекс размерности, который должен быть зафиксирован во время оптимизации (`columns = [1]`), и его фиксированное значение (`values = [1]`)*. Поскольку нам нужно найти только максимизатор апостериорного среднего, который соответствует цели (высокоточная функция), указываем, что второй столбец (индекс `1`) всегда должен иметь значение 1:

```
from botorch.acquisition.fixed_feature import FixedFeatureAcquisitionFunction
from botorch.acquisition import PosteriorMean

post_mean_policy = FixedFeatureAcquisitionFunction(
    acq_function=PosteriorMean(model),   //Оптимизируем апостериорное среднее
    d=2,    //Число измерений пространства поиска
    columns=[1],     //Индекс фиксированного столбца
    values=[1],      //Значение фиксированного столбца
)
```

Затем используем вспомогательную функцию `optimize_acqf()`, чтобы найти точку, максимизирующую оценку приобретения, которая является апостериорным средним значением целевой функции:

```
final_x, _ = optimize_acqf(
    post_mean_policy,  //Оптимизируем апостериорное среднее
    #Остальные аргументы такие же, как и при оптимизации другой политики
    bounds=bounds,
    q=1,
    num_restarts=20,
    raw_samples=50,
)
```

Переменная `final_x` — это местоположение, которое максимизирует апостериорное среднее целевой функции. В нашем Jupyter notebook мы поместили этот код во вспомогательную функцию, которая возвращает `final_x`, дополненный значением корреляции — 1, что указывает на истинную целевую функцию:

```
def get_final_recommendation(model):
    #Делаем обертку для политики
    post_mean_policy = FixedFeatureAcquisitionFunction(...)
    final_x, _ = optimize_acqf(...)  //Оптимизируем оценку приобретения

    #Дополняем окончательную рекомендацию значением корреляции = 1
    return torch.cat([final_x, torch.ones(1, 1)], dim=1)
```

Теперь во время цикла БО вместо записи действующего значения как индикатора прогресса оптимизации мы вызываем вспомогательную функцию `get_final_recommendation()`. Кроме того, вместо максимального количества запросов, которые необходимо выполнить при каждом запуске, теперь у нас есть максимальный бюджет, который можно потратить на запросы как с низкой, так и с высокой точностью. Другими словами, мы продолжаем использовать алгоритм оптимизации до тех пор, пока совокупные затраты не превысят бюджетный лимит. «Скелет» нашего разноточного цикла БО выглядит следующим образом:

```
budget_limit = 10   //Максимальная стоимость каждого запуска оптимизации

#Для отслеживания рекомендации, максимизирующей апостериорное среднее в ходе оптимизации
recommendations = []
spent_budget = []  //Для отслеживания затрат на каждой итерации

... #Генерируем случайное начальное наблюдение

current_budget = 0

while current_budget < budget_limit:
    ... #Обучаем ГП на текущих данных

    #Обновляем запись с новейшей рекомендацией
    rec_x = get_final_recommendation(model)
```

```
    recommendations.append(evaluate_all_functions(rec_x).item())
    spent_budget.append(current_budget)

    ... #Инициализируем политику и оптимизируем ее оценку привлечения

    current_budget += cost_model(next_x).item()    //Продолжаем отслеживать затраты

    ... #Обновляем обучающие данные
```

Теперь можно запустить разноточную MES для оптимизации целевой функции Форрестера. В качестве эталона также запустим одноточную MES, которая запрашивает только основную истину $f(x)$. Для запуска одноточной политики с разноточной моделью нужна политика-оболочка из класса `FixedFeatureAcquisitionFunction`, чтобы ограничить точность, которую может запрашивать политика:

```
policy = FixedFeatureAcquisitionFunction(
    #Обернутая политика — это одноточная MES
    acq_function=qMaxValueEntropy(model, candidate_x, num_fantasies=128),
    d=2,
    #Фиксируем значение корреляции, равное 1, во втором столбце (индекс 1)
    columns=[1],
    values=[1],
)

#Оптимизируем оценку приобретения вспомогательной функцией optimize_acqf
next_x, acq_val = optimize_acqf(...)
```

Результат запуска двух политик показан на рис. 9.14, где видно, что разноточная MES значительно превосходит обычную версию политики. Экономическая эффективность также показывает преимущества баланса между информацией и затратами. Однако стоит отметить, что это результат лишь одной итерации. В первом упражнении мы будем запускать данный эксперимент несколько раз с разными исходными наборами. А также понаблюдаем среднюю эффективность этих политик.

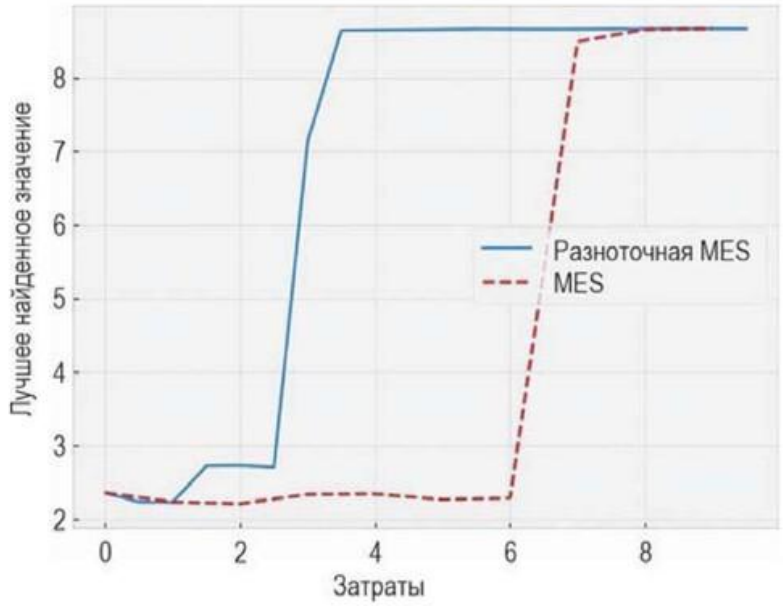


Рис. 9.14. Целевое значение максимизатора апостериорного среднего как функция затрат, потраченных двумя политиками: разноточная политика MES значительно превосходит обычную

На этом можно завершить обсуждения задачи разноточной оптимизации, в которой достигается баланс между оптимизацией целевой функции и стоимостью получения знаний о ней. Мы узнали, как реализовать модель ГП, которая может учиться на основе нескольких источников данных. Это позволило нам рассуждать о теоретической ценности запроса с точки зрения оптимального значения цели. Объединив теоретико-информационную величину со стоимостью запроса при измерении окупаемости инвестиций, мы разработали экономичный разноточный сценарий политики MES, который может автоматически сочетать знания и затраты.

9.5. Упражнение 1: Визуализация средней производительности в разноточной оптимизации

Чтобы сравнить эффективность политик, на рис. 9.14 показано целевое значение максимизатора апостериорного среднего на протяжении всей оптимизации в зависимости от суммы затраченного бюджета. Однако это результат всего лишь одного запуска, и важно было бы показать среднюю производительность каждой политики из множества экспериментов. Идея повторных экспериментов впервые была затронута в упражнении 2 к главе 4. Здесь мы запустим цикл оптимизации несколько раз и научимся брать среднюю производительность для получения более целостного сравнения. К концу упражнения будет видно, что разноточная MES хорошо балансирует между информацией и затратами и оптимизирует целевую функцию более эффективно, чем ее одноточный аналог. Решение хранится в `CH09/04 - Exercise 1.ipynb`.

Выполните следующие шаги:

1. Скопируйте настройку задачи и цикл разноточной оптимизации из `CH09/03 - Measuring Performance.ipynb` и добавьте еще одну переменную, обозначающую количество экспериментов, которые нужно провести (по умолчанию 10).
2. Чтобы облегчить повторные эксперименты, добавьте внешний цикл в код цикла оптимизации. Это должен быть цикл `for` с 10 итерациями, в котором каждый раз генерируются разные случайные наблюдения. Это можно сделать, установив для случайного начального числа номер итерации. Так можно гарантировать, что генератор случайных чисел будет возвращать одни и те же данные при разных запусках с одним и тем же начальным числом.
3. Код в `CH09/03 - Measuring Performance.ipynb` использует два списка (`recommendations` и `spent_budget`) для отслеживания хода оптимизации. Сделайте каждую из этих переменных списком, состоящим из других списков, где каждый вложенный список служит той же цели, что и соответствующий список в `CH09/03 - Measuring Performance.ipynb`. Это позволит отслеживать ход оптимизации в течение 10 экспериментов и сравнивать различные политики на последующих этапах.
4. Запустите разноточную политику MES и ее одноточную версию.
5. Поскольку стоимость запроса низкоточной функции отличается от стоимости запроса высокоточной, вполне вероятно, что списки в `spent_budget` не совсем совпадают друг с другом. Другими словами, точки кривых на рис. 9.14 не имеют

одинаковых координат x на разных прогонах. Такое несоответствие не позволяет использовать средний прогресс, хранящийся в `recommendations`, при нескольких запусках.

Чтобы решить проблему, нужно использовать линейную интерполяцию для каждой кривой прогресса, которая позволяет «заполнять» значения прогресса в обычной сетке. Именно в этой сетке мы будем усреднять производительность каждой политики через прогоны. Для линейной интерполяции используйте `np.interp` из NumPy, который принимает в качестве первого аргумента обычную сетку. Она может представлять собой массив целых чисел между 0 и `budget_limit`: `np.arange(budget_limit)`. Второй и третий аргументы — это координаты точек x и y, составляющих каждую кривую прогресса (т. е. каждый внутренний список в `spent_budget` и `recommendations`).

6. Используйте линейно интерполированные значения, чтобы построить среднюю производительность и полосы ошибок двух примененных политик, а затем сравните их производительность.
7. Из-за текущего способа, которым мы измеряем эффективность оптимизации, возможно, что список рекомендаций, которые мы отслеживаем на протяжении каждого запуска, увеличивается не монотонно. Иначе говоря, мы можем получить рекомендацию, которая будет работать хуже, чем рекомендация от предыдущей итерации. Чтобы проверить это явление, можно построить линейно интерполированные кривые, представляющие прогресс оптимизации отдельных прогонов, а также среднюю производительность и полосы ошибок. Реализуйте данную визуализацию для двух примененных политик и проверьте немонотонность полученных кривых.

9.6. Упражнение 2: Разноточная оптимизация с использованием нескольких низкоточных приближений

Подход, изученный в этой главе, можно обобщить на сценарии, где существует более одного низкоточного приближения целевой функции, которые можно запросить. Стратегия здесь аналогична: разделите объем информации, получаемый от каждого запроса, на его стоимость, а затем выберите запрос, который дает наибольшую рентабельность инвестиций. Это упражнение показывает, что наша разноточная политика MES может балансировать между несколькими низкоточными функциями. Решение хранится в `CH09/05 - Exercise 2.ipynb`.

Выполните следующие шаги:

1. В качестве целевой используйте двумерную функцию Бранина (`Branin`), которая является обычной тестовой функцией для оптимизации, подобно функции Форрестера. BoTorch имеет в своем арсенале разноточную версию Бранина, поэтому импортируем ее в наш код с помощью команды `botorch.test_functions.multi_fidelity import AugmentedBranin`. Для удобства масштабируем область определения и выходные данные этой функции, используя следующий код, который делает `objective` функцией, вызываемой при оценке запроса:

```
#Импортируем функцию Бранина из BoTorch
problem = AugmentedBranin()

#Обрабатываем ввод и вывод функции, отображая значения в удобном диапазоне
def objective(X):
    X_copy = X.detach().clone()
    X_copy[..., :-1] = X_copy[..., :-1] * 15 - 5
    X_copy[..., -2] = X_copy[..., -2] + 5
    return (-problem(X_copy) / 500 + 0.9).unsqueeze(-1)
```

2. Определите границы пространства поиска как единичный квадрат. Две нижние границы равны 0, а две верхние — 1.
3. Объявите переменную `fidelities`, которая хранит значения корреляции различных функций, доступных для запроса. Здесь мы имеем доступ к двум низкоточным приближениям со значениями корреляции 0,1 и 0,3, поэтому точность должна содержать эти два числа и 1 в качестве последнего элемента.

   Все три функции представлены на рис. 9.15, где яркие пиксели обозначают высокие целевые значения. Видно, что оба низкоточных приближения следуют общей тенденции, которая демонстрируется основной истиной, и сходство с ней увеличивается по мере роста значения точности. То есть второе приближение с точностью 0,3 (в центре) больше похоже на истинную целевую функцию (справа), чем первое приближение с точностью 0,1 (слева).

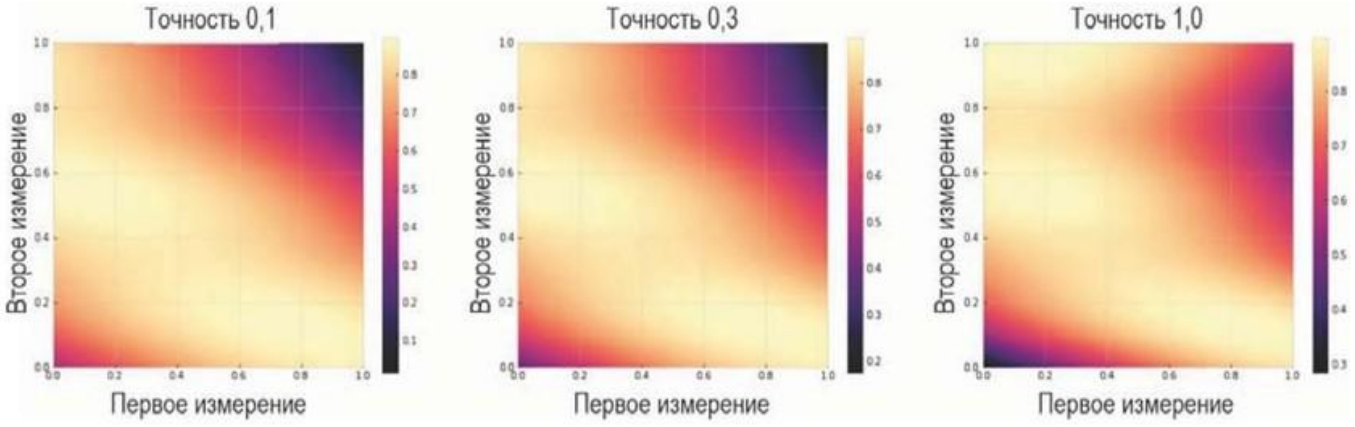


Рис. 9.15. Целевая функция Бранина (справа) и два приближения низкой точности

4. Установите фиксированную стоимость модели линейных затрат равной 0,2, а вес — 1. Это означает, что стоимость запроса низкоточной функции на рис. 9.15 (слева) равна 0,2 + 1 × 0,1 = 0,3. Аналогично стоимость запроса функции (по центру) равна 0,5, а стоимость запроса истинной целевой функции — 1,2. Установите лимит затрат в каждом эксперименте равным 10. Количество повторных экспериментов также установите равным 10.
5. Установите количество кандидатов, выбранных из последовательности Соболя, равным 5000, и используйте 100 перезапусков и 500 необработанных выборок при использовании вспомогательных функций для оптимизации оценки приобретения данной политики.

6. Переопределите вспомогательную функцию `get_final_recommendation`, которая находит максимизатор апостериорного среднего, чтобы аргументы были установлены соответствующим образом для нашей двумерной целевой функции: `d = 3` и `columns = [2]`.
7. Запустите разноточную политику MES и ее одноточную версию и постройте график среднего прогресса оптимизации и полосу ошибок для каждой политики, используя метод, описанный в упражнении 1. Обратите внимание, что при создании политики-оболочки для одноточной политики аргументы `d` и `columns` должны быть установлены так же, как и на предыдущем шаге. Убедитесь, что разноточная политика работает лучше, чем одноточная.

Заключение

- Разноточная оптимизация — это сценарий, при котором есть доступ к нескольким источникам информации, каждый из которых имеет свой уровень точности и стоимости. В этом случае необходимо сбалансировать количество информации, полученной в результате выполнения действия, и стоимость самого действия.
- При разноточной оптимизации высокоточная функция предоставляет точную информацию, но ее оценка требует больших затрат, в то время как низкоточная функция не требует дорогих запросов, но может дать неточную или недостаточную информацию. На каждой итерации цикла оптимизации необходимо решить, где и какую функцию запросить, чтобы как можно быстрее найти оптимум цели.
- Уровень информации, которую предоставляет каждое приближение, количественно определяется корреляцией между ним и основной истиной, которая представляет собой число от 0 до 1. Чем выше корреляция, тем более точно приближение соответствует основной истине. В Python значение корреляции каждой точки данных сохраняется как дополнительный столбец в матрице характеристик.
- ГП с ядром, которое может обрабатывать значения корреляции обучающих точек данных, может быть обучен на разноточном наборе данных. Для этой задачи можно использовать разноточную разновидность ядра Матерна. Неопределенность в предсказаниях ГП зависит от того, из какой функции исходит каждое наблюдение и, если оно исходит из низкоточной функции, каково значение ее корреляции.
- Для кодирования стоимости запроса каждого уровня точности, к которому есть доступ во время оптимизации, используется линейная модель. Установив параметры модели (фиксированную стоимость и вес), можно смоделировать положительно коррелированную связь между стоимостью запроса и качеством данных. Нелинейные модели затрат также можно реализовать с помощью BoTorch.
- Чтобы сбалансировать информативность и стоимость, мы используем вариант политики MES, который взвешивает объем информации, полученной в резуль-

тате каждого запроса, по величине, обратной стоимости запроса. Такая мера аналогична концепции рентабельности инвестиций в экономике и реализуется с помощью класса `InverseCostWeightedUtility` в BoTorch.

- Точный расчет информационного прироста для оптимального значения целевой функции (основная задача MES) неразрешим из-за негауссовского распределения оптимального значения. Чтобы аппроксимировать прирост информации, используется последовательность Соболя для представления всего пространства поиска, облегчая вычислительную нагрузку при расчете.
- Разноточная политика MES успешно балансирует информацию и затраты, а также отдает приоритет экономически эффективным запросам.
- Чтобы оптимизировать оценку приобретения в рамках разноточной политики, мы используем вспомогательную функцию `optimize_acqf_mixed()`. Она может работать со смешанным пространством поиска, измерения которого могут быть как непрерывными, так и дискретными.
- Чтобы точно измерить производительность в условиях разноточности, мы используем максимизатор апостериорного среднего на каждой итерации в качестве окончательной рекомендации перед завершением. Эта величина отражает то, что мы знаем о целевой функции, лучше, чем высокоточное действующее значение.
- Чтобы оптимизировать одноточную оценку приобретения в разноточном сценарии, мы используем *политику-оболочку*, которая является экземпляром класса `FixedFeatureAcquisitionFunction`. Чтобы ее инициализировать, нужно объявить, какое измерение в пространстве поиска фиксировано и в каком значении.

10 Обучение на парных сравнениях с помощью оптимизации предпочтений

Темы главы:

- Задача обучения и оптимизации предпочтений с использованием только данных парного сравнения.
- Обучение ГП на парных сравнениях.
- Политики оптимизации для парных сравнений.

Было ли такое, что вам трудно было оценить что-либо (еду, продукт, опыт и пр.) по точной шкале? Числовая оценка (рейтинг) — это обычная задача в рабочих процессах A/B-тестирования и рекомендаций по какому-либо продукту.

> **Определение**
>
> Термин «A/B-тестирование» относится к методу измерения опыта пользователя в двух средах (A и B) посредством рандомизированных экспериментов и определения, какая среда является более желательной. A/B-тестирование обычно проводится технологическими компаниями.

A/B-тестерам и специалистам по рекомендации продуктов часто приходится сталкиваться с высоким уровнем «шума» в отзывах клиентов. Под «шумом» подразумевается любой тип искажения, которому они подвержены. Примерами могут служить количество рекламы в онлайн-сервисах потокового вещания, качество службы доставки или общее настроение человека, когда он потребляет продукт. Эти и многие другие факторы влияют на то, как мы оцениваем что-то, потенциально искажая сигнал, который является истинной оценкой продукта.

Неконтролируемые внешние факторы искажают «истинную» оценку от покупателя. Поэтому часто бывает трудно оценить что-то по числовой шкале. Большой шум при обратной связи в A/B-тестировании и рекомендациях означает, что сервисная платформа не может полагаться на некоторые данные пользователей, чтобы узнать об их предпочтениях. Вместо этого им необходимо собирать больше данных от клиентов, чтобы быть более уверенными в рекомендациях.

Однако, как и в других сценариях оптимизации «черного ящика» (настройка гиперпараметров или открытие лекарств), здесь запрос целевой функции также является дорогостоящим. При рекомендации продукта каждый раз, запрашивая у пользователя оценку, мы рискуем вторгнуться в его опыт и отбить желание продолжать использовать сервисную платформу. Следовательно, существует естественное противоречие между потребностью в большом объеме данных для лучшего изучения предпочтений и навязчивостью, которая потенциально может привести к потере клиентов.

К счастью, есть способ обойти данную проблему. Исследования в области психологии показали интересный результат: люди гораздо лучше дают ответы о своих предпочтениях, если те преподносятся им в форме парных сравнений (например, «Продукт А лучше, чем продукт Б»), чем оценка по шкале (например, «Товар А — 8 из 10»).

Определение

Парное сравнение — это метод сбора данных о предпочтениях. Каждый раз, когда нужно получить информацию о пожеланиях кого-либо, мы просим его выбрать (из двух вариантов) тот продукт, который он предпочитает. Парные сравнения отличаются от числовых рейтингов тем, что в последних покупатель оценивает товар по шкале.

Причина, почему нам проще дается парное сравнение, заключается в том, что оно представляет собой менее когнитивно нагруженную задачу. Мы можем согласовываться с нашими истинными предпочтениями лучше, сравнивая два объекта, нежели оценивая продукт по шкале. На рис. 10.1 приведены два примера интерфейса интернет-магазина, который предлагает оценить гавайскую рубашку:

- Первый интерфейс предлагает оценить рубашку по шкале от 1 до 10. Это может быть затруднительным, особенно если у вас нет какой-то внутренней системы координат по этому вопросу.
- Второй интерфейс предлагает выбрать ту рубашку, которая вам больше нравится (из двух вариантов). Такую задачу выполнить проще.



Рис. 10.1. Примеры выявления предпочтений пользователя в рекомендациях продуктов: слева предлагается оценить рекомендуемый продукт по шкале; справа предлагается выбрать продукт, который ему больше нравится; второй вариант помогает лучше выявить предпочтения пользователя

Учитывая потенциал высококачественных данных, которые можно собрать с помощью парных сравнений, есть возможность применить этот метод к байесовской оптимизации предпочтений пользователя. Но при этом возникает ряд вопросов. Как мы можем обучить ML-модель на данных такого типа? Или как представить клиенту новые сравнения, чтобы лучше изучить и оптимизировать его предпочтения? В данной главе мы ответим на эти вопросы, используя модель ГП, которая может эффективно обучаться на основе парных сравнений. Также мы разработаем стратегии, которые помогут сравнивать наилучшие конкурирующие данные о продукте, найденные на текущий момент, что позволит максимально быстро оптимизировать пожелания пользователя. Иначе говоря, мы предположим, что предпочтения — это целевая функция, определенная в пространстве поиска, и ее необходимо оптимизировать.

Подобный вариант изучения и оптимизации предпочтений представляет собой уникальную задачу, которая лежит на стыке оптимизации «черного ящика» и рекомендаций продуктов, и вызывает интерес в обоих сообществах. К концу этой главы мы узнаем, как подойти к данной проблеме с помощью БО, сочетая исследование и эксплуатацию при сборе данных.

10.1. Оптимизация «черного ящика» с помощью парных сравнений

В этом разделе мы обсудим полезность парных сравнений в задаче выявления предпочтений, а затем исследуем цикл БО, модифицированный для данного сценария.

По сравнению с рейтингами, парные сравнения создают меньшую когнитивную нагрузку на пользователя и, следовательно, с большей вероятностью приводят к получению данных высокого качества (обратная связь, соответствующая истинным предпочтениям пользователя).

Парные сравнения в многоцелевой оптимизации

Ситуация, в которой парные сравнения особенно полезны, — это принятие решений, когда необходимо учитывать несколько критериев. Предположим, что вы хотите купить автомобиль и выбираете между двумя: А и Б. Чтобы принять решение, вы перечисляете различные характеристики, которые вас интересуют: внешний вид, практичность, энергоэффективность, стоимость и т. д. Затем вы оцениваете обе машины по каждому из критериев, надеясь найти явного победителя. Так вышло, что по одним критериям автомобиль А имеет более высокие баллы, чем автомобиль В. А по другим критериям — наоборот, автомобиль В имеет баллы выше, чем автомобиль А. Таким образом, между ними нет явного победителя, а объединение оценок по разным критериям в единый результат может оказаться затруднительным.

Допустим, некоторые характеристики вас волнуют больше, чем другие, поэтому они должны иметь больший вес при объединении с остальными параметрами для получения единой конечной оценки. Однако определение точных значений весов может оказаться еще более сложной задачей, чем выбор между двумя автомобилями! Гораздо проще было бы игнорировать детали, рассматривать каждую машину в целом и сравнивать две машины «лицом к лицу».

Поэтому простота использования парных сравнений делает их хорошим выбором для сценариев оптимизации, когда необходимо учитывать несколько критериев. Например, в исследовательском проекте Эдварда Эйбеля, Людмилы Михайловой и Джона Кина используются парные сравнения для решения проблемы группового принятия решений.

Само собой, парные сравнения объективно не лучше, чем численные рейтинги. Несмотря на то что первые легче получить от пользователя, они все же содержат значительно меньше информации, чем вторые. Ответ, что оранжевая рубашка вам нравится больше, чем красная (рис. 10.1), содержит ровно один бит информации (результат сравнения является двоичным: либо оранжевый, либо красный). Если бы вы оценили оранжевую рубашку на 8 баллов из 10, а красную — на 6, у нас было бы больше информации, чем просто знание, что одна оценивается выше другой.

Всегда существует компромисс при выборе метода, которым мы будем получать обратную связь от клиента. Численные оценки содержат больше информации, но они подвержены шуму и могут создать большую когнитивную нагрузку на человека. С другой стороны, парные сравнения предоставляют меньше данных, но их легче получить, а также они менее шумные". Данные плюсы и минусы суммированы на рис. 10.2.



Рис. 10.2. Различия между числовыми рейтингами и парными сравнениями по информативности и сложности получения: каждый метод выявления предпочтений имеет свои преимущества и недостатки

Другие способы выяснения предпочтений клиентов

Парные сравнения — не единственная форма уменьшения когнитивной нагрузки относительно числовых оценок. Например, онлайн-сервис потокового вещания Netflix собирает рейтинги зрителей, предлагая им выбрать один из трех вариантов: «палец вниз» означает «не нравится», «палец вверх» — «нравится» и «два пальца вверх» — «обожаю». Такой вариант представляет собой задачу порядковой классификации, в которой элементы распределяются по разным категориям и между ними существует определенный порядок. Также интересно было бы рассмотреть проблему продуктовых рекомендаций, но мы сосредоточимся на парных сравнениях.

В этой главе мы узнаем, как упростить задачу использования парных сравнений для изучения и оптимизации предпочтений клиента с помощью байесовской оптимиза-

ции. Сначала рассмотрим модифицированную версию цикла БО (рис. 1.6), которая изображена на рис. 10.3:

1. На шаге 1 ГП обучается на данных парного сравнения вместо числовых оценок. Ключевая задача состоит в том, чтобы гарантировать, что предположение ГП о целевой функции (истинной функции предпочтений пользователя) отражает информацию в наблюдаемых сравнениях.
2. На шаге 2 политика БО вычисляет оценки приобретения, чтобы количественно измерить, насколько полезен каждый новый потенциальный запрос к пользователю. Он должен быть представлен в виде пары вариантов, которые пользователь сможет сравнить. Как и в других ситуациях, политика должна сбалансировать эксплуатацию региона, в котором предпочтения высоки, и исследование других регионов, в которых мало что известно о предпочтениях пользователя.
3. На шаге 3 пользователь сравнивает два продукта, представленные политикой, и сообщает о том, который он предпочитает. Эта информация затем добавляется в обучающий набор.

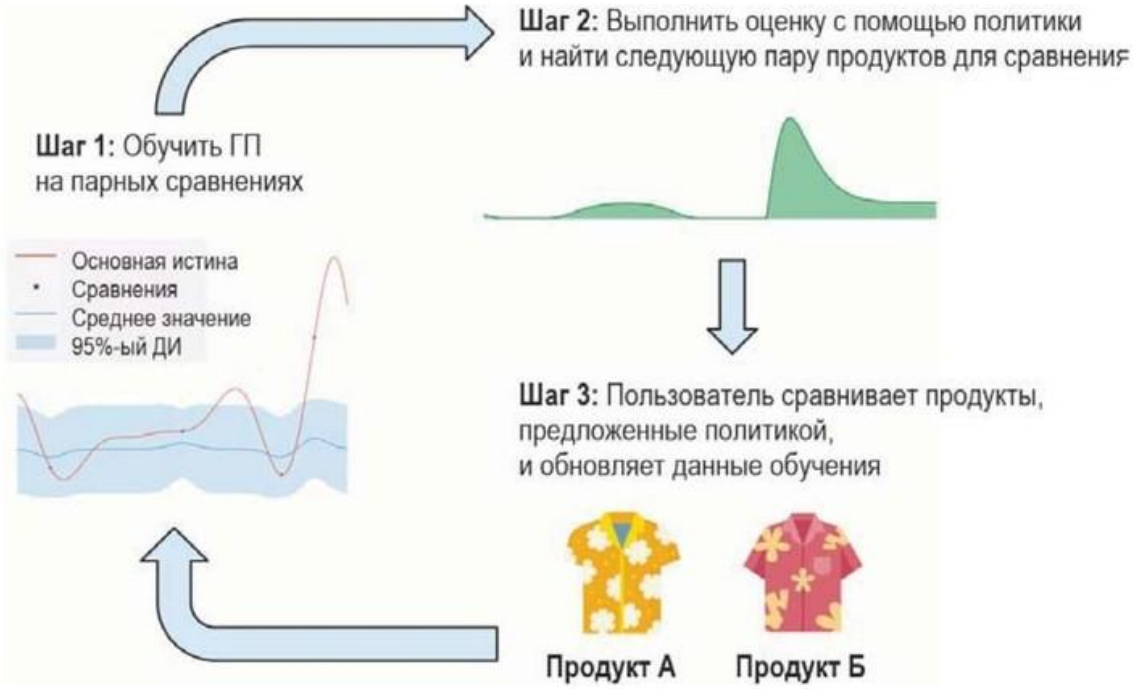


Рис. 10.3. Цикл БО с парными сравнениями для оптимизации предпочтений: ГП обучается на данных парного сравнения, а политика БО решает, какую пару точек данных следует попросить пользователя сравнить

В оставшейся части главы мы постараемся ответить на два основных вопроса:

1. Как можно обучить ГП только на парных сравнениях? ГП, обученный на рейтингах, дает вероятностные прогнозы с количественной неопределенностью, что имеет решающее значение при принятии решений. Можно ли использовать здесь ту же модель для попарного сравнения?
2. Как следует создавать новые пары продуктов для сравнения, чтобы как можно быстрее определить максимизатор предпочтений пользователя? Иначе говоря,

как лучше всего получить обратную связь от людей, используя парные сравнения, чтобы оптимизировать их предпочтения?

10.2. Формулировка задачи оптимизации предпочтений и форматирование данных парного сравнения

Прежде чем начать отвечать на эти вопросы, сначала рассмотрим задачу рекомендации продукта, которую мы будем решать на протяжении всей главы, и решим, как мы будем моделировать ее в Python. Правильная постановка поможет легче интегрировать инструменты БО, о которых мы узнаем в следующих разделах. Код, который здесь используется, включен в первую часть файла `CH10/01 - Learning from pairwise comparisons.ipynb`.

Как показано на рис. 10.1 и 10.3, текущий сценарий представляет собой задачу с рекомендацией гавайских рубашек. Представьте, что у нас есть интернет-магазин таких рубашек, и мы пытаемся определить продукт, максимизирующий предпочтения конкретного покупателя, который в настоящее время покупает рубашку.

Для простоты предположим, что после краткого опроса нам стало известно, что наиболее важным для покупателя является количество цветов на рубашке. Другие факторы, такие как стиль и цвет, тоже имеют значение, но для него самое главное в гавайской рубашке — максимальная «цветочность» (как можно больше цветков). Далее, предположим, что на складе есть много рубашек с разным количеством цветков, поэтому мы можем найти рубашку почти с любой заданной степенью их наличия. Наша цель — найти рубашку с оптимальным (неизвестным для нас) количеством цветков, учитывая предпочтения заказчика. Поиск проводится в одномерном пространстве, где нижняя его граница соответствует рубашкам без цветочных узоров, а верхняя граница содержит рубашки, полностью покрытые цветами.

На рис. 10.4 сценарий показан более подробно. В верхней части видны истинные предпочтения покупателя и то, как они меняются в зависимости от цветочности рубашки:

- Ось X — количество цветов на рубашке. На одном конце спектра находятся рубашки без цветов, а на другом — полностью покрытые цветами.
- Ось Y — предпочтения покупателя для каждой рубашки. Чем выше предпочтение, тем больше рубашка нравится заказчику.

Можно сделать вывод, что заказчику нравятся рубашки с цветочным принтом. Существует локальный оптимум по оси X выше средней точки. Глобальный оптимум функции расположен вблизи верхней границы пространства поиска. Это означает, что рубашка с большим количеством цветов, но не покрытая ими полностью, максимизирует предпочтения покупателя.

Поскольку перед нами стоит задача оптимизации «черного ящика», кривая предпочтений на рис. 10.4 фактически недоступна в реальном мире. Необходимо узнать об этой функции больше с помощью парных сравнений и оптимизировать ее как можно быстрее. Теперь рассмотрим, как можно решить данную задачу в Python.

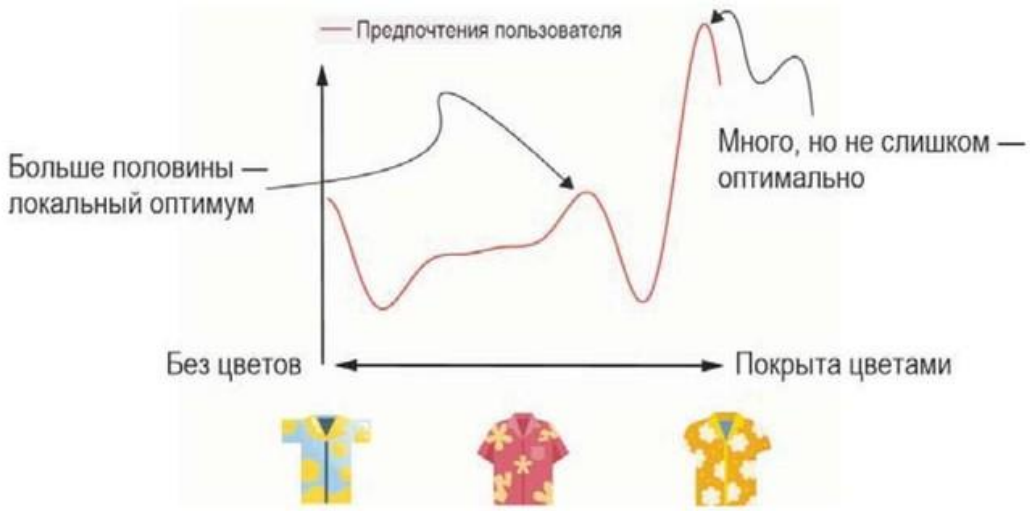


Рис. 10.4. Поиск рубашки с оптимальным количеством цветов в задаче рекомендации товара: пространство поиска одномерно, поскольку поиск выполняется только среди количества цветов на рубашке; рубашка, покрытая цветами больше чем наполовину, является локальным оптимумом, а рубашка, покрытая почти полностью, максимизирует предпочтения пользователя

Возможно, вы уже заметили, что мы используем функцию Форрестера (задействованную в предыдущих главах) для моделирования цели — истинного предпочтения клиента. Код данной функции не отличается от кода в других главах. Поэтому формула, определяющая нижнюю и верхнюю границы (от –5 до 5), будет аналогична:

```
#Целевая функция
def objective(x):
    y = -((x + 1) ** 2) * torch.sin(2 * x + 2) / 5 + 1 + x / 3
    return y

#Границы пространства поиска
lb = -5
ub = 5
bounds = torch.tensor([[lb], [ub]], dtype=torch.float)
```

Наверняка вы также помните, что метки данных имеют числовые значения, поэтому каждая точка из обучающего набора (хранящаяся в переменной `train_x`) имеет соответствующую метку в `train_y`. Текущие настройки немного отличаются. Поскольку данные представлены в форме пар, каждое наблюдение является результатом сравнения двух точек данных в `train_x`, а метка наблюдения указывает, какая точка больше ценится клиентом.

> **Примечание**
>
> Мы следуем соглашению BoTorch, чтобы кодировать результат каждого парного сравнения между двумя точками данных в `train_x` в виде тензора PyTorch из двух элементов. Первый из них — индекс точки данных, которая является предпочтительной в `train_x`, а второй — индекс точки данных, которая является предпочтительной.

Предположим, на основе двух запросов к пользователю стало известно, что он больше предпочитает $x = 0$, чем $x = 3$ (следовательно, $f(0) > f(3)$, где $f(x)$ — целевая функция). Также он больше предпочитает $x = 0$, чем $x = -4$ (следовательно, $f(0) > f(-4)$).

Можно эти две части информации представить в виде набора обучающих данных, используя переменную `train_x`, имеющую следующие значения:

```
tensor([[ 0.],  #Представляет x = 0
        [ 3.],  #Представляет x = 3
        [-4.]]) #Представляет x = -4
```

В тензоре представлены три значения x, которые используются для запроса к пользователю. При этом обучающие метки `train_comp` должны быть такими:

```
tensor([[0, 1],  #Представляет f(0) > f(3)
        [0, 2]]) #Представляет f(0) > f(-4)
```

Каждая строка в `train_comp` является двухэлементным тензором, представляющим результат парного сравнения. Первая строка `[0, 1]` означает, что точка с индексом `0` в `train_x` (которая равна `x = 0`) предпочтительнее точки с индексом `1` (которая равна `x = 3`). Аналогично вторая строка `[0, 2]` представляет сравнение $f(0) > f(-4)$.

Чтобы упростить процесс сравнения, напишем вспомогательную функцию, которая принимает целевые значения любых двух точек данных и возвращает `[0, 1]`, если первое больше второго, или `[1, 0]` в противном случае:

```
def compare(y):
    #Убеждаемся, что у нас только 2 целевых значения для сравнения
    assert y.numel() == 2

    #Если больше первое значение
    if y.flatten()[0] > y.flatten()[1]:
        return torch.tensor([[0, 1]]).long()
    #Если больше второе значение
    else:
        return torch.tensor([[1, 0]]).long()
```

Используем эту функцию для создания образца обучающего набора. Сначала случайным образом создаем две точки данных в пространстве поиска:

```
#Фиксируем случайное начальное число для воспроизводимости
torch.manual_seed(0)
#Создаем два числа между 0 и 1 и масштабируем их на наше пространство поиска
train_x = bounds[0] + (bounds[1] - bounds[0]) * torch.rand(2, 1)
```

Переменная `train_x` содержит следующие две точки:

```
tensor([[-0.0374],
        [ 2.6822]])
```

Далее получим результат сравнения между этими двумя точками, оценив функцию истинного предпочтения пользователя и вызвав `compare()`:

```
#Вычисляем фактические целевые значения, которые скрыты от нас
train_y = objective(train_x)
train_comp = compare(train_y)   //Получаем результат сравнения
```

Результат сравнения сохраняется в `train_comp`:

```
tensor([[0, 1]])
```

Он означает, что первая точка данных в `train_x` ценится клиентом выше, чем вторая.

Также напишем еще одну вспомогательную функцию с именем `observe_and_append_data()`, роль которой состоит в том, чтобы получить пару точек данных, сравнить их и добавить результат в текущий обучающий набор:

1. Функция сначала вызывает вспомогательную функцию `compare()`, чтобы получить либо `[0, 1]`, либо `[1, 0]`. Затем она корректирует значения индексов, хранящихся в двухэлементном тензоре, таким образом, чтобы они указывали на правильные местоположения точки данных в обучающем наборе:

```
def observe_and_append_data(x_next, f, x_train, comp_train, tol=1e-3):
    #Оцениваем сравнение в соответствии с предпочтениями пользователя
    x_next = x_next.to(x_train)
    y_next = f(x_next)
    comp_next = compare(y_next)

    #отслеживаем индексы
    n = x_train.shape[-2]
    new_x_train = x_train.clone()
    new_comp_next = comp_next.clone() + n
```

2. Функция также проверяет точки из набора, которые достаточно близки друг к другу, чтобы считаться одной и той же точкой (например, $x = 1$ и $x = 1{,}001$).

   Такие схожие данные могут привести к тому, что обучение ГП, основанного на предпочтениях (о котором мы узнаем в следующем разделе), станет численно нестабильным. Решение — пометить схожие точки, рассмотреть их как дубликаты и удалить одну из них:

```
n_dups = 0

    #Проверяем наличие дубликатов первой точки данных в новой паре
    dup_ind = torch.where(
        torch.all(torch.isclose(x_train, x_next[0], atol=tol), axis=1)
    )[0]
    #Если дубликатов нет, добавляем точку данных в train_x
    if dup_ind.nelement() == 0:
        new_x_train = torch.cat([x_train, x_next[0].unsqueeze(-2)])
    #Если есть хотя бы один дубликат, отслеживаем его индекс
    else:
        new_comp_next = torch.where(
            new_comp_next == n, dup_ind, new_comp_next - 1
        )
        n_dups += 1
```

```
    #Проверяем наличие дубликатов второй точки данных в новой паре
    dup_ind = torch.where(
        torch.all(torch.isclose(new_x_train, x_next[1], atol=tol), axis=1)
    )[0]
    #Если дубликатов нет, добавляем точку данных в train_x
    if dup_ind.nelement() == 0:
        new_x_train = torch.cat([new_x_train, x_next[1].unsqueeze(-2)])
    #Если есть хотя бы один дубликат, отслеживаем его индекс
    else:
        new_comp_next = torch.where(
            new_comp_next == n + 1 - n_dups, dup_ind, new_comp_next
        )

    #Возвращаем обновленный обучающий набор
    new_comp_train = torch.cat([comp_train, new_comp_next])
    return new_x_train, new_comp_train
```

Эти две вспомогательные функции мы будем использовать повторно в последующих задачах по обучению ГП и оптимизации функции предпочтений пользователя. Первую из них рассмотрим далее.

10.3. Обучение ГП, основанного на предпочтениях

Для реализации модели в этом разделе мы используем код из `CH10/01 - Learning from pairwise comparisons.ipynb`.

В разделе 2.2.2 мы узнали, что в соответствии с правилом байесовского обновления (которое позволяет обновлять убеждения в свете новых данных) можно получить точную апостериорную форму MVN-распределения, учитывая, что наблюдения касались лишь некоторых переменных. Такая способность точно вычислить апостериорное нормальное распределение является основой для обновления ГП в свете новых наблюдений. К сожалению, такое точное обновление применимо только при численных наблюдениях. Другими словами, мы можем сделать точное обновление ГП только с помощью наблюдений в форме $y = f(x)$, где x и y — действительные числа.

В нынешних условиях наблюдения осуществляются в форме парных сравнений, а апостериорная форма ГП, обусловленная этим типом данных, больше не является ГП. Это исключает большинство методов, разобранных в данной книге и основанных на том факте, что прогностическая модель является ГП. Однако это не значит, что нам следует отказаться от всего проекта.

Аппроксимация апостериорного ГП в процессе парных сравнений

Попытка приближенно решить задачу, когда нет возможности выполнить ее точно, — это распространенная тема в машинном обучении (и информатике в целом). В нашем контексте данное приближение приравнивается к поиску апостериорной формы ГП, которая дала бы наибольшую *вероятность* (**Likelihood**) для наблюдаемых парных сравнений. Более подробную информацию об этом подходе можно найти в исследова-

тельской работе Вэя Чу (**Wei Chu**) и Зубина Гахрамани (**Zoubin Ghahramani**), которые его предложили.

Конечно, распределение, которое действительно максимизирует вероятность данных, является апостериорным распределением, отличным от ГП. Но поскольку нам нужен именно он в качестве прогностической модели, позволяющей использовать изученные политики, наша цель — найти ГП с наибольшей вероятностью данных. Обратите внимание, что его поиск полностью совпадает с тем, что мы делаем, когда обучаем его. Мы находим лучшие гиперпараметры для ГП (масштаб длины, масштаб вывода и т. д.), которые максимизируют вероятность данных (впервые это подробно обсуждается в разделе 3.3.2).

С точки зрения реализации можно инициализировать и обучить ГП на парных сравнениях следующим образом:

- BoTorch предоставляет специальную реализацию класса для модели под названием `PairwiseGP`, которую можно импортировать из модуля `botorch.models.pairwise_gp`.
- Вероятность данных попарного сравнения требует иного расчета, нежели вероятность данных с реальными значениями. Для этого используем `PairwiseLaplaceMarginalLogLikelihood`, импортированный из того же модуля, что и реализация класса.
- Чтобы иметь возможность визуализировать и проверять прогнозы, зафиксируем масштаб его выходных данных так, чтобы во время обучения он сохранял значение по умолчанию, равное 1. Мы также отключим градиент с помощью `model.covar_module.raw_outputscale.requires_grad_(False)`. Этот шаг предназначен только для визуализации и поэтому не является обязательным. Далее при запуске политик оптимизации в данной главе это нам не понадобится.
- Используем вспомогательную функцию `fit_gpytorch_mll` из `botorch.fit`, чтобы получить апостериорный ГП, который максимизирует вероятность обучающих данных:

```
#Импортируем необходимые классы и вспомогательную функцию
from botorch.models.pairwise_gp import PairwiseGP,
  PairwiseLaplaceMarginalLogLikelihood
from botorch.fit import fit_gpytorch_mll

#Инициализируем модель ГП
model = PairwiseGP(train_x, train_comp)
#Фиксируем масштаб вывода для большей читаемости (необязательно)
model.covar_module.raw_outputscale.requires_grad_(False)
#Инициализируем объект правдоподобия (логарифм)
mll = PairwiseLaplaceMarginalLogLikelihood(model)
#Обучаем модель, максимизируя вероятность
fit_gpytorch_mll(mll);
```

Используя обученную модель, теперь можно делать и визуализировать прогнозы в пространстве поиска (рис. 10.5). Отметим несколько интересных моментов по поводу этих предсказаний:

- Средние прогнозы подчиняются соотношению, выраженному в обучающих данных ($f(-0{,}0374) > f(2{,}6822)$), в том смысле, что средний прогноз при $x = -0{,}0374$ больше 0, а при $x = 2{,}6822$ меньше 0.
- Неопределенность прогнозов в районе −0,0374 и 2,6822 также ниже, чем в остальных случаях. Такая разница в неопределенности отражает тот факт, что, наблюдая $f(-0{,}0374) > f(2{,}6822)$, мы получили некоторую информацию о $f(-0{,}0374)$ и $f(2{,}6822)$, и наши знания об этих двух целевых значениях должны увеличиться.

Однако неопределенность в данных точках не снижается существенно до нуля, как это происходит, когда обучение делается на числовых наблюдениях (например, на рис. 2.14). Это связано с тем, что парные сравнения не дают столько информации, сколько числовые рейтинги (раздел 10.1), поэтому сохраняется значительный уровень неопределенности. Рис. 10.5 показывает, что обученный ГП может эффективно обучаться на основе парных сравнений, где функция среднего подчиняется наблюдаемому сравнению, а неопределенность хорошо откалибрована.

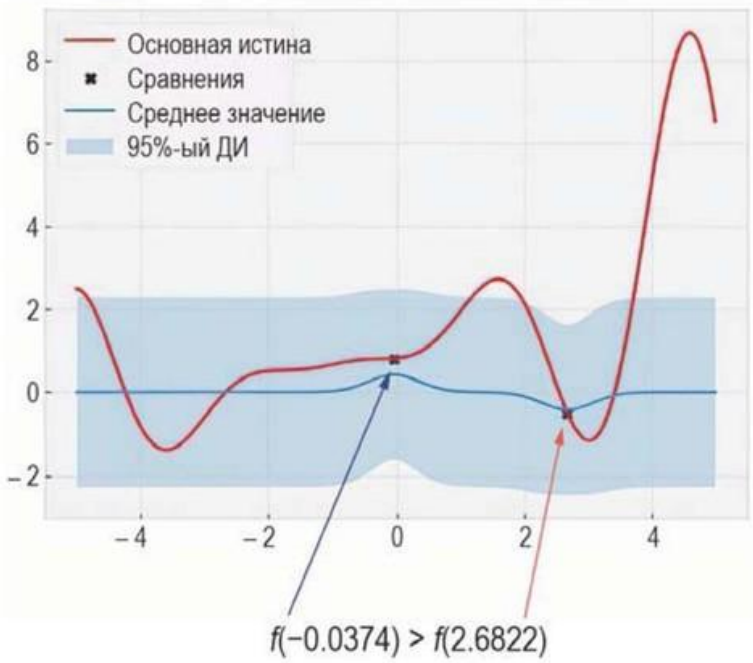


Рис. 10.5. Прогнозы ГП, обученные на парном сравнении $f(-0{,}0374) > f(2{,}6822)$: апостериорное среднее отражает результат данного сравнения, в то время как апостериорное стандартное отклонение вокруг двух точек данных немного уменьшилось по сравнению с априорным

Чтобы продолжить работу с этой моделью и посмотреть, как она может учиться на более сложных данных, создадим обучающий набор большего размера. Предположим, что мы хотели бы обучить ГП на трех отдельных сравнениях: $f(0) > f(3)$, $f(0) > f(-4)$ и $f(4) > f(-0)$. Все они верны для целевой функции, представленной на рис. 10.5. Для этого устанавливаем обучающие точки данных, хранящиеся в `train_x`, как:

```
train_x = torch.tensor([[0.], [3.], [-4.], [4.]])
```

Этот набор содержит все точки, участвовавшие в предыдущих наблюдаемых сравнениях. Далее, касательно `train_comp`, кодируем три сравнения, используя двухэлементные тензоры, как это обсуждалось в разделе 10.2:

```
train_comp = torch.tensor(
    [
        [0, 1], #[0, 1] означает f(train_x[0]) > f(train_x[1]) или f(0) > f(3)
        [0, 2], #[0, 2] означает f(train_x[0]) > f(train_x[2]) или f(0) > f(-4)
        [3, 0], #[3, 0] означает f(train_x[3]) > f(train_x[0]) или f(4) > f(0)
    ]
)
```

Теперь просто переобъявляем ГП и переопределяем его на основе новых обучающих данных:

```
#Инициализируем модель ГП
model = PairwiseGP(train_x, train_comp)
#Инициализируем объект вероятности (логарифм)
mll = PairwiseLaplaceMarginalLogLikelihood(model)
#Обучаем модель, максимизируя вероятность
fit_gpytorch_mll(mll)
```

Модель дает прогнозы, показанные на рис. 10.6, где видно, что все три результата сравнения отражаются в средних прогнозах, а неопределенность снова уменьшается вокруг обучающих точек данных.

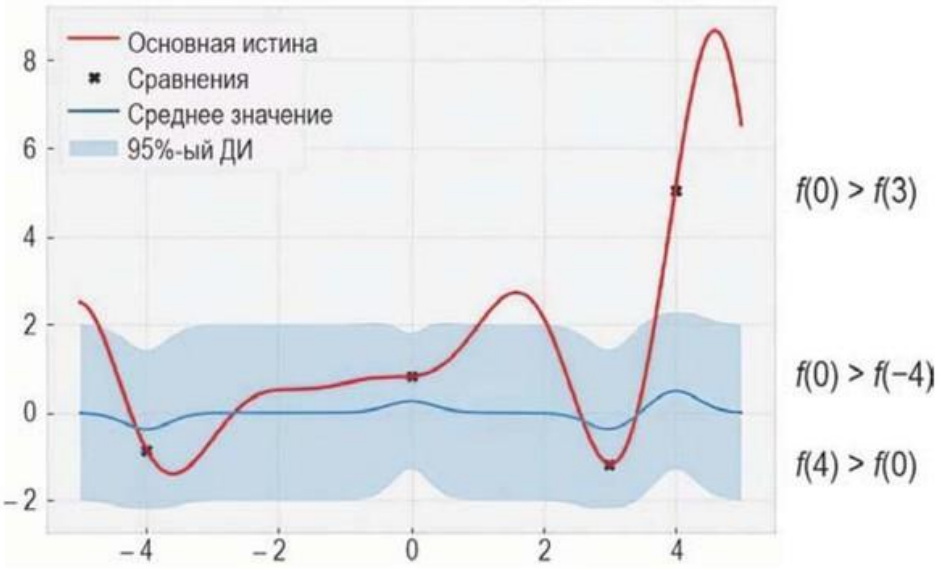


Рис. 10.6. Прогнозы ГП, обученного на парных сравнениях, показаны справа: апостериорное среднее отражает результат сравнения, в то время как апостериорное стандартное отклонение вокруг точек данных в обучающем наборе уменьшилось по сравнению с априорным

Рис. 10.6 показывает, что модель ГП может эффективно обучаться на данных парного сравнения. Теперь у нас есть возможность учиться на основе информации о предпочтениях и делать вероятностные прогнозы о целевой функции. Это подводит нас к последней теме этой главы — принятие решений при оптимизации предпочтений. То есть как следует выбирать пары данных, чтобы пользователь мог их сравнивать, а также как можно быстрее найти наиболее предпочтительную точку данных?

Диапазон целевой функции при обучении на предпочтениях

Интересным преимуществом парных сравнений относительно численных рейтингов является то, что во время обучения не нужно учитывать диапазон целевой функции. Это потому, что нас волнуют только относительные сравнения между целевыми значениями.

Другими словами, изучение $f(x)$ эквивалентно изучению $f(x) + 5$, $2f(x)$ или $f(x)/10$.

Между тем при обучении традиционного ГП с помощью числовых рейтингов крайне важно учитывать диапазон целевой функции, потому что только так можно получить модель с хорошо откалиброванной количественной оценкой неопределенности. Например, для моделирования целевой функции в диапазоне от −1 до 1 подходит масштаб вывода, равный 1, а для целевой функции в диапазоне от −10 до 10 нужен больший масштаб вывода.

10.4. Оптимизация предпочтений с помощью игры «Царь горы»

В этом разделе мы научимся применять БО для обучения на предпочтениях. Используемый код доступен в `CH10/02 - Optimizing preferences.ipynb`.

Вопрос, который нужно решить, заключается в следующем: как выбрать лучшую пару для сравнения, представить ее пользователю и узнать его предпочтения, чтобы найти наилучшую точку данных. Как и в любой политике оптимизации, стратегия должна достичь баланса между эксплуатацией (сосредоточение внимания на области в пространстве поиска, где оценка пользователя высока) и исследованием (проверка областей, о которых мало что известно).

Политики БО, которые были изучены в главах с 4 по 6, эффективно решают этот компромисс, используя различные эвристики. Поэтому мы разработаем стратегию, которая позволит переназначить их для нашей задачи. Как вы помните, в предыдущих главах политика вычисляла оценку приобретения для каждой точки данных в пространстве поиска с помощью количественной оценки ее значения, помогая нам оптимизировать целевую функцию. Найдя точку, которая максимизирует оценку приобретения, мы получим следующую точку для оценки целевой функции.

Использование политики для предложения парных сравнений

В текущем сценарии (на основе предпочтений) нам нужно предоставить пользователю пару точек данных для сравнения. На каждой итерации цикла оптимизации мы компонуем эту пару, во-первых, из точки данных, которая максимизирует оценку приобретения для политики, а во-вторых, из наилучшей точки, которую мы нашли на текущий момент.

Используемая стратегия напоминает популярную детскую игру «Царь горы». На каждой итерации мы пытаемся «побить» лучшую точку данных, найденную на данный момент (текущий царь горы), используя претендента, выбранного политикой, как показано на рис. 10.7.

Используя эту стратегию, мы передаем на аутсорсинг задачу создания пары точек данных для сравнения с обычной политикой, которая может хорошо сбалансиро-

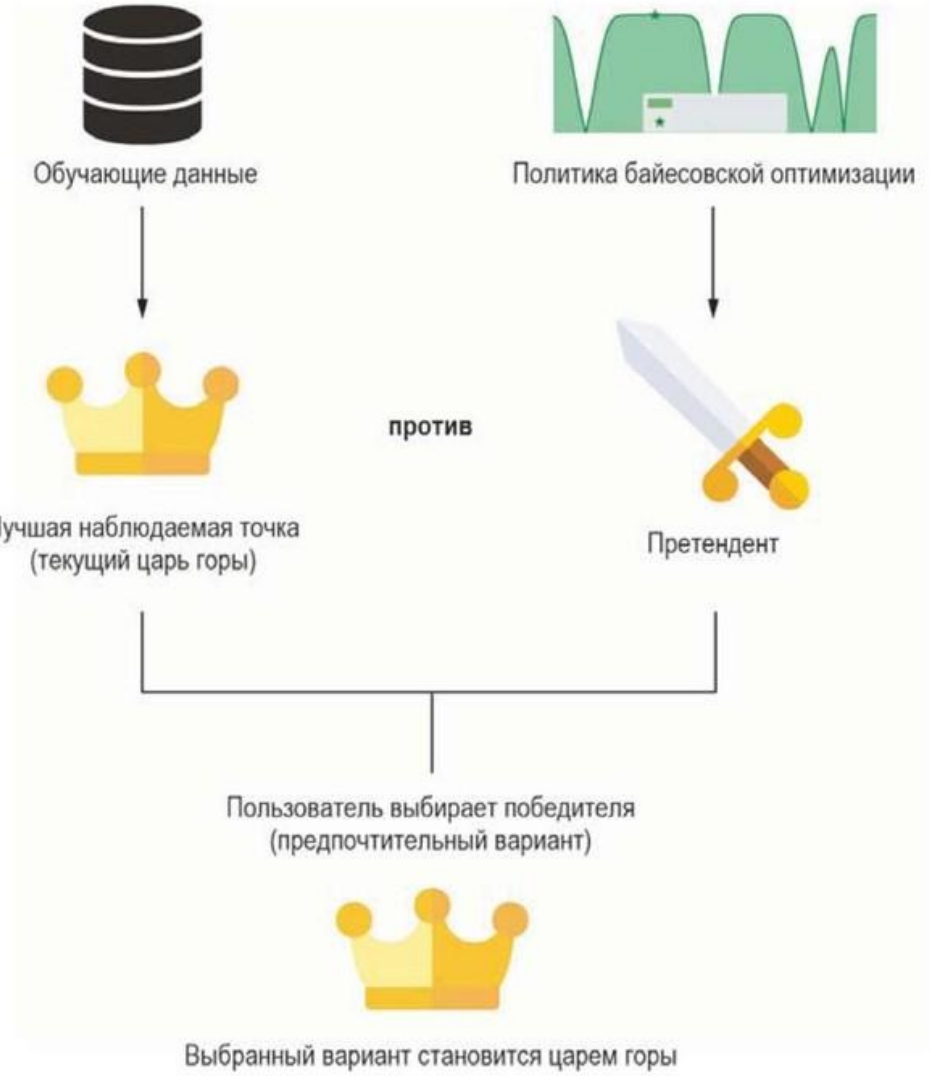


Рис. 10.7. Иллюстрация стратегии «Царь горы» в байесовской оптимизации предпочтений: лучший результат, найденный к текущему моменту, сравнивается с многообещающим кандидатом, определенным политикой

вать компромисс между эксплуатацией и исследованием, а также с которой мы уже знаем, как работать.

С точки зрения кода эту стратегию легко реализовать. Мы просто объявляем объект политики и оптимизируем его оценку приобретения с помощью вспомогательной функции `optimize_acqf()`. Например, следующий код задействует политику верхней доверительной границы (UCB, раздел 5.2). Она использует верхние границы прогностических нормальных распределений, сделанных ГП, в качестве оценок приобретения, чтобы количественно измерить ценность проверяемых точек данных:

```
#Инициализируем политику БО
policy = UpperConfidenceBound(model, beta=2)

#Находим точку данных, которая максимизирует оценку приобретения
challenger, acq_val = optimize_acqf(
    policy,
    bounds=bounds,
    q=1,
```

```
    num_restarts=50,
    raw_samples=100,
)
```

Еще одна политика, которую мы будем использовать, — это ожидаемое улучшение (EI, раздел 4.3). Атрибут EI, который делает ее подходящей для наших условий, заключается в том, что стратегия политики точно соответствует игре «Царь горы». То есть политика стремится найти точки данных, которые в среднем приводят к максимальному улучшению (с точки зрения значения целевой функции) по сравнению с лучшей точкой, наблюдаемой на данный момент. Нахождение значения, которое превышает наилучшее текущее, напрямую отражает суть стратегии «царь горы». Чтобы осуществить EI в наших условиях, используем реализацию другого класса, который может обрабатывать зашумленные наблюдения, и называется `qNoisyExpectedImprovement`.

Шумные наблюдения в БО

Термин «шумные наблюдения» относится к ситуации, когда есть подозрения, что наблюдаемые метки искажены шумом таким же образом, как описано в начале этой главы.

Как показано на рис. 10.5 и 10.6, у нас все еще есть значительная неопределенность в прогнозах, даже в местах, включенных в обучающие данные `train_x`. Здесь следует использовать шумную версию EI, поскольку она обрабатывает этот тип неопределенного прогноза лучше, чем обычная политика EI. Реализуем шумный вариант следующим образом:

```
#Инициализируем политику
policy = qNoisyExpectedImprovement(model, train_x)

#Находим точку данных, которая максимизирует оценку приобретения
challenger, acq_val = optimize_acqf(
    policy,
    bounds=bounds,
    q=1,
    num_restarts=50,
    raw_samples=100,
)
```

Для сравнения также включим стратегию равномерного случайного выбора претендента на лучшую точку в пространстве поиска:

```
#Выбираем случайную точку от 0 до 1 и масштабируем ее в соответствии
#с нашим пространством поиска
challenger = bounds[0] + (bounds[1] - bounds[0]) * torch.rand(1, 1)
```

Эта стратегия служит ориентиром для определения, могут ли имеющиеся политики работать лучше, чем случайный выбор. Теперь мы можем запустить цикл БО, чтобы оптимизировать предпочтения пользователя в нашем примере. Код этого цикла похож на тот, что использовался в предыдущих главах, за исключением шага, где мы предоставляем пару точек данных пользователю для его обратной связи с целью

добавить результат в обучающий набор. Это можно сделать с помощью вспомогательной функции `observe_and_append_data()` из раздела 10.2:

```
#Находим лучшую точку, наблюдаемую на данный момент
incumbent_ind = train_y.argmax()
#Собираем в пакет лучшую точку и точку, предложенную политикой
next_x = torch.vstack([train_x[incumbent_ind,:], challenger])

#Обновляем обучающие данные
train_x, train_comp = observe_and_append_data(
    next_x, objective, train_x, train_comp
)
train_y = objective(train_x)
```

В `CH10/02 - Optimizing preferences.ipynb` каждый запуск БО начинается со случайно сгенерированной пары точек данных и ответа от целевой функции, сравнивающей их. Затем при каждом запуске выполняется 20 парных сравнений (20 запросов к пользователю). Эксперимент повторяется 10 раз для каждой политики, чтобы можно было наблюдать совокупную эффективность каждой стратегии.

На рис. 10.8 показано среднее наилучшее значение (и полоса ошибок), полученное с помощью имеющихся стратегий оптимизации. EI работает лучше всего, последовательно находя глобальный оптимум. Возможно, большую часть этого успеха можно объяснить согласованностью между методом «царя горы» и алгоритмической мотивацией политики. Еще более удивительно, что UCB не может превзойти стратегию равномерного случайного выбора претендента. Возможно, другое значение компромиссного параметра β сможет улучшить ее производительность.

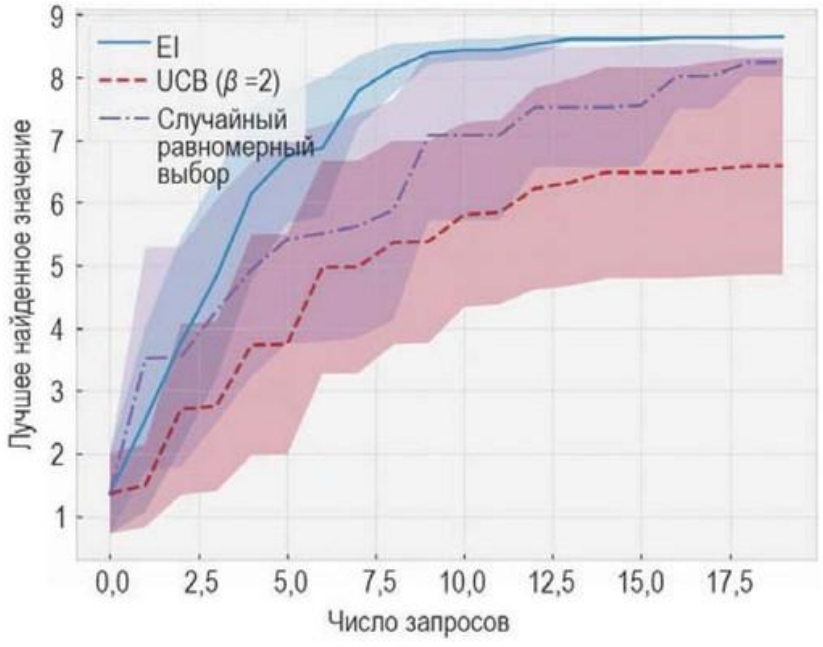


Рис. 10.8. Производительность оптимизации различных политик, совокупная по 10 экспериментам: EI работает лучше всего, последовательно находя глобальный оптимум; UCB не удалось превзойти стратегию случайного равномерного поиска претендента

Примечание

Компромиссный параметр β в UCB напрямую контролирует баланс политики между исследованием и эксплуатацией. Дополнительную информацию об этом параметре можно узнать в разделе 5.2.2.

В этой главе мы рассмотрели задачу обучения и оптимизации предпочтений с использованием парных сравнений. Мы узнали о мотивах, лежащих в основе сбора данных согласно этому методу, а также о его преимуществах перед рейтингами. Затем мы решили задачу оптимизации с помощью БО, сначала обучив ГП парным сравнениям с использованием приближенного метода. Такая модель может эффективно изучать отношения между точками в обучающем наборе, при этом предлагая хорошо откалиброванную количественную оценку неопределенности. В конце мы научились применять политики к решению данной задачи, сопоставляя наилучшую точку данных с точкой, рекомендованной конкретной политикой. В следующей главе мы узнаем о многоцелевом варианте задачи оптимизации «черного ящика», когда существует несколько конкурирующих целевых функций, которые необходимо сбалансировать во время оптимизации.

Заключение

- В приложениях с рекомендациями товаров сравнение двух элементов может помочь получить обратную связь, которая в большей степени соответствует истинным предпочтениям пользователя, чем рейтинги. Это связано с тем, что первый вариант ставит перед собой менее когнитивно нагруженную задачу.
- Парные сравнения содержат меньше информации, чем численные оценки, поэтому существует компромисс между уменьшением когнитивной нагрузки для пользователя и получением информации, когда нужно сделать выбор между двумя методами выявления предпочтений.
- ГП может быть обучен максимизировать вероятность парных сравнений в наборе данных. Эта модель аппроксимирует истинную апостериорную «не-ГП» модель, когда она обусловлена данными парного сравнения.
- ГП, обученный на парных сравнениях, дает средние прогнозы, соответствующие результатам сравнения в обучающем наборе. В частности, средние прогнозы в предпочтительных местах превышают средние прогнозы в менее предпочитаемых локациях.
- Неопределенность ГП, обученного на парных сравнениях, немного уменьшается по сравнению с априорным ГП, но не сводится к нулю, что соответствующим образом отражает неопределенность в отношении предпочтений пользователя, поскольку парные сравнения предлагают меньше информации, чем числовые оценки.
- Стратегия оптимизации предпочтений с помощью БО предполагает сопоставление наилучшей найденной точки данных и кандидата, рекомендованного поли-

тикой. Механизм данной стратегии состоит в том, чтобы постоянно пытаться превзойти наилучшую точку, найденную на данный момент.

- Результат парного сравнения представлен в BoTorch в виде двухэлементного тензора, где первый элемент — это индекс точки данных, которая является предпочтительной в обучающем наборе, а второй элемент — индекс непредпочтительной точки данных.
- Чтобы использовать EI при оптимизации с парными сравнениями, нужно использовать шумную версию политики, которая может обрабатывать высокие уровни неопределенности в обученном ГП лучше, чем обычная EI.

11 Оптимизация нескольких целевых функций одновременно

Темы главы:

- Задача одновременной оптимизации нескольких целевых функций.
- Обучение нескольких ГП работе с множеством целевых функций одновременно.
- Совместная оптимизация нескольких целей.

Каждый день мы сталкиваемся с компромиссами в области оптимизации:

- «Этот кофе вкусный, но в нем слишком много сахара».
- «Эта рубашка выглядит великолепно, но она не в моем ценовом диапазоне».
- «Нейросеть, которую я только что обучил, обладает высокой точностью, но она слишком велика и требует много времени для обучения».

Пытаясь добиться хороших результатов в чем-либо, мы жертвуем некоторыми критериями, которые важны. Любитель кофе и сладкоежка в одном лице может оптимизировать вкус своего напитка, используя нездоровое количество сахара. Покупатель будет пытаться балансировать между великолепным внешним видом предмета одежды и высокой его стоимостью. Нейросеть, созданная ML-инженером, может иметь хорошие прогностические характеристики, но при этом быть слишком большой для использования в приложениях реального времени. Сосредоточив внимание на одной задаче, мы плохо выполняем другую, которую также важно учитывать. Но можно попытаться смоделировать все целевые функции одновременно и попробовать оптимизировать их все вместе. Например, можно поискать вкусные рецепты кофе с низким содержанием сахара, модные и доступные вещи или ML-модели, которые хорошо работают и практичны в реализации. Подобный тип задач называется *многоцелевой оптимизацией* (**Multiobjective optimization**).

Определение

Задача многоцелевой оптимизации, как следует из названия, включает в себя несколько целевых функций, которые необходимо оптимизировать одновременно. Нам необходимо найти точки данных, которые достигают высоких значений по всем целям.

В любой нетривиальной задаче могут возникнуть конкурирующие цели. При этом единственный способ добиться хорошей производительности по одной из них — это пожертвовать производительностью по другой. Такой внутренний конфликт приводит к необходимости сбалансировать эти цели. Это очень похоже на компромисс между эксплуатацией и исследованием (раздел 4.1.2), которые, по сути, являются двумя конкурирующими целями оптимизации внутри цикла БО.

В этой главе мы подробно поговорим о многоцелевой оптимизации. Мы научимся успешно ее достигать, находя точки данных, производительность которых по одной цели не может быть улучшена без ущерба для другой. А также научимся применять БО к задаче, когда целевые функции требуют больших затрат на запрос. Многоцелевая оптимизация — распространенная задача во многих областях и к концу главы мы научимся решать ее с помощью байесовских методов.

11.1. Балансирование нескольких целей оптимизации с помощью БО

Приложения многоцелевой оптимизации очень распространены:

- В проектировании и производстве инженеры часто сталкиваются с необходимостью балансировать между несколькими целями: качество продукта и стоимость производства. Например, автоконцерны постоянно оптимизируют свою производственную линию, чтобы максимизировать качество и минимизировать затраты.
- При распределении ресурсов (например, денежной и медицинской помощи среди бедных групп населения или тех, кто пострадал от стихийных бедствий) лица, принимающие решения, должны сбалансировать максимальную помощь и различные логистические трудности в распределении.
- Ученые, разрабатывающие лекарства, пытаются сбалансировать максимальную эффективность препарата и минимизацию побочных эффектов на пациентов (раздел 8.1.1).
- Практическая модель машинного обучения должна обеспечивать хорошую производительность при низких затратах на обучение.

В отличие от других видов задач оптимизации, обсуждавшихся в предыдущих главах, у нас больше нет единой цели, на которой можно сосредоточиться. Во многих из них цели, которые необходимо оптимизировать, противоречат друг другу: жертвуя производительностью по одному показателю, можно улучшить другой. Один из способов понять внутренний конфликт — понять, что нам приходится «жонглировать» различными целями одновременно. Нельзя просто сосредоточиться на одних критериях, игнорируя другие. Необходимость совмещать несколько объектов одновременно показана на рис. 11.1. К счастью, тот факт, что теперь существует более одной целевой функции, не влияет на большую часть рабочего процесса БО, который разрабатывался на протяжении всей этой книги.



Рис. 11.1. Карикатура, иллюстрирующая баланс, которого мы должны достичь при многоцелевой оптимизации

Моделирование нескольких целевых функций с использованием нескольких ГП

В предыдущих главах мы обучали модель ГП на наблюдаемых данных, чтобы смоделировать предположение о единственной целевой функции, подлежащей оптимизации. В этой главе нам предстоит смоделировать несколько целевых функций. Каждую из них можно реализовать как отдельные ГП, поддерживая которые мы получим возможность рассуждать обо всех целевых функциях вероятностным образом.

На рис. 11.2 показан цикл БО, в котором необходимо оптимизировать две целевые функции. По сравнению с рис. 1.6 здесь на шаге 1 для каждой из целей есть свой

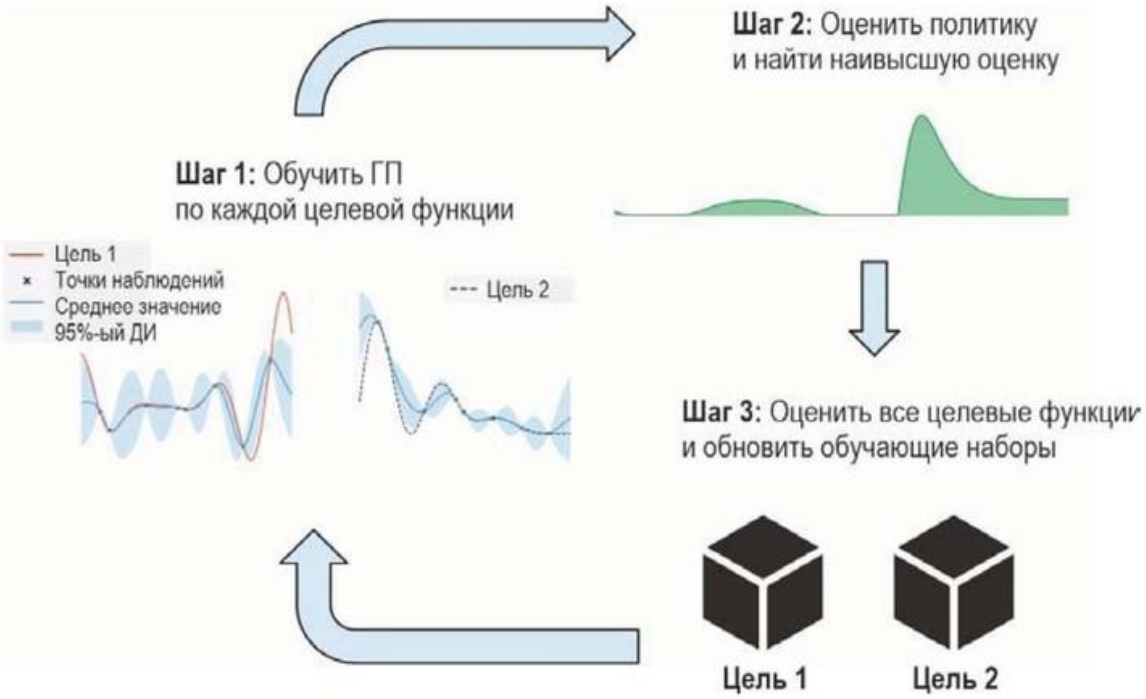


Рис. 11.2. Многоцелевой цикл БО с двумя функциями: ГП обучается на данных каждой целевой функции, а политика решает, по какой точке данных оценивать целевые функции в следующий раз

ГП, и каждая точка данных, определенная политикой, оценивается по всем целевым функциям на шаге 3.

Обучение ГП работе с данными по каждой цели осуществить несложно. Фактически это уже было сделано это для ограниченной оптимизации в главе 8, где один ГП реализует целевую функцию, а другой — функцию ограничений. Иначе говоря, нам нужно сосредоточиться только на разработке политики (шаг 2), чтобы принимать эффективные решения во время оптимизации. В оставшейся части главы мы сосредоточимся на изучении, как политика должна обеспечивать баланс между многочисленными целями, чтобы как можно быстрее найти высокопроизводительные точки данных.

11.2. Нахождение границы наиболее оптимальных точек данных

Рассмотрим более подробно математические концепции, которые обычно используются в многоцелевой оптимизации для количественной оценки прогресса, достигнутого в ходе оптимизации. Они помогают определить цель стратегии оптимизации, которая будет разработана позже в этой главе. Мы будем использовать код из `CH11/01 – Computing Hypervolume.ipynb`.

Начнем с двух целевых функций, которые необходимо оптимизировать одновременно:

- Первая цель — уже знакомая нам функция Форрестера, задействованная в предыдущих главах. Ее глобальный оптимум расположен в правой части пространства поиска. Эта функция реализована как `objective1()` в коде ниже.
- Вторая цель — еще одна функция, реализованная как `objective2()`. Она имеет другую форму и поведение. Важно отметить, что ее глобальный оптимум расположен в левой части пространства поиска. Несовпадение глобальных оптимумов моделирует ситуацию с конкурентными целями, для которых должен быть достигнут компромисс.
- Вспомогательная функция `joint_objective()` возвращает значения обеих целевых функций в тензоре PyTorch для конкретной точки входных данных `x`. Она поможет сохранить код лаконичным.
- Наконец, определим пространство поиска в диапазоне от –5 до 5:

```
#Первая целевая функция
def objective1(x):
    return -((x + 1) ** 2) * torch.sin(2 * x + 2) / 5 + 1 + x / 20

#Вторая целевая функция
def objective2(x):
    return (0.1 * objective1(x) + objective1(x - 4)) / 3 - x / 3 + 0.5
```

```
#Вспомогательная функция, которая вызывает обе целевые функции
def joint_objective(x):
    y1 = objective1(x)
    y2 = objective2(x)
    return torch.vstack([y1.flatten(), y2.flatten()]).transpose(-1, -2)

#Границы пространства поиска
lb = -5
ub = 5
bounds = torch.tensor([[lb], [ub]], dtype=torch.float)
```

На рис. 11.3 показаны обе целевые функции в нашем пространстве поиска. Как можно наблюдать, точки данных, максимизирующие обе цели, отличаются друг от друга. Сплошная кривая достигает максимума при $x = 4{,}5$, а пунктирная кривая — при $x = -4{,}5$. Такая разница означает, что мы имеем две конкурирующие цели, и их совместная оптимизация требует компромисса между их целевыми значениями.

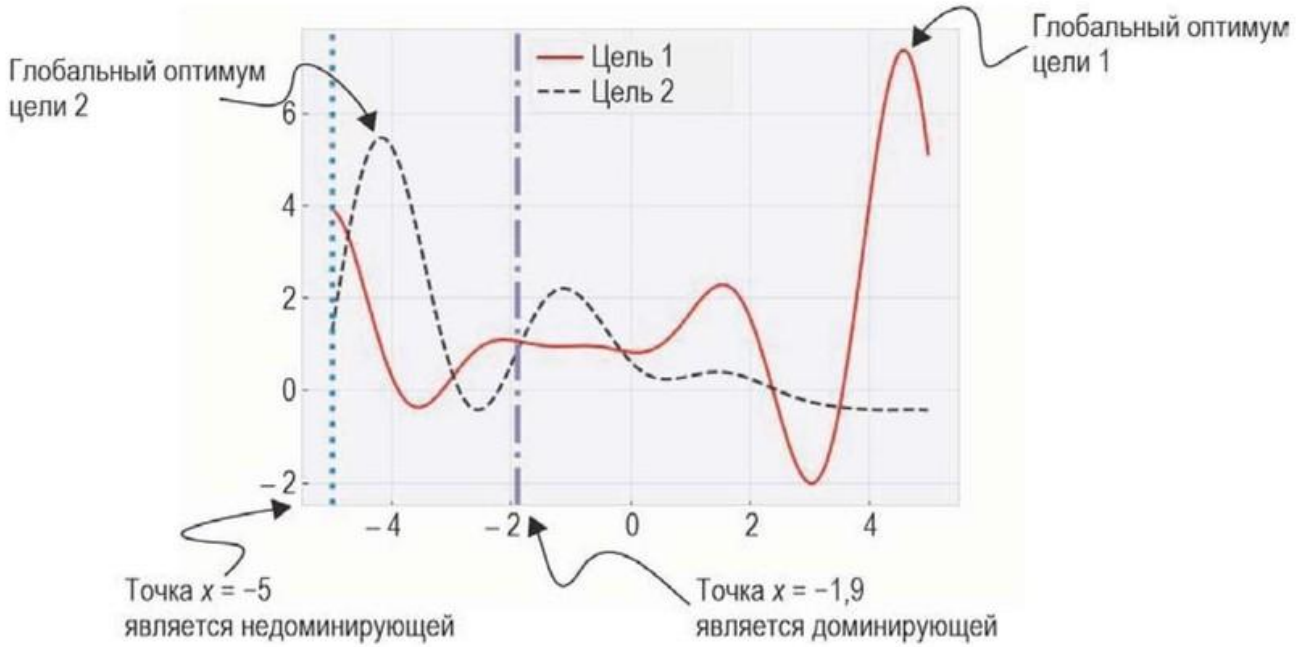


Рис. 11.3. Визуализация двух целевых функций в текущей задаче многоцелевой оптимизации: точки данных, которые максимизируют цели, отличаются друг от друга, поэтому существует компромисс при их оптимизации

Под «компромиссом» подразумевается, что в пространстве поиска существуют точки (x), значения которых не могут быть улучшены для первой цели ($f_1(x)$), если они не будут снижены для второй ($f_2(x)$).

В качестве примера рассмотрим точку $x = -5$ (рис. 11.3). Она расположена в крайнем левом углу пространства поиска: $f_1(-5)$ примерно равна 4, $f_2(-5)$ примерно равна 1,5. Если мы хотим достичь результата лучше 4 по первой цели $f_1(x)$, придется пожертвовать результатом 1,5 по $f_2(x)$. Единственный способ, чтобы значение $f_1(x)$ было выше 4, — это выполнить запрос в самой правой части пространства поиска, где $x > 4$. Здесь значения $f_2(x)$ падают ниже 0.

Это работает и в обратную сторону. Область справа ($x > 4$) также является местом, где существует напряжение между $f_1(x)$ и $f_2(x)$. Чтобы увеличить значение $f_2(x)$, нужно уйти в левую часть пространства, но в этом случае пострадает значение $f_1(x)$.

Определение

Точка данных, значение которой для одной цели не может быть превышено, пока для другой оно не уменьшится, называется *недоминирующей* (**Nondominated**). Точка, противоположная ей, называется *доминирующей* (**Dominated**). Это такая точка x_1, при которой существует другая точка x_2, все целевые значения которой превышают значения x_1. Недоминирующую точку также можно назвать *оптимальной по Парето* (**Pareto optimal**), *эффективной по Парето* (**Pareto efficient**) или *неуступающей* (**Non-inferior**).

Таким образом, точка $x = -5$ является недоминирующей, как и некоторые другие, где $x > 4$. Примером доминирующей точки является $x = -1{,}9$ (рис. 11.3), что дает нам $f_1(-1{,}9) \approx f_2(-1{,}9) \approx 1$. В этой точке доминирует $x = -5$, поскольку первые целевые значения ниже, чем вторые: $f_1(-1{,}9) < f_1(-5)$ и $f_2(-1{,}9) < f_2(-5)$.

Во многих случаях мы имеем бесконечное количество недоминирующих точек. На рис. 11.4 они показаны в виде заштрихованных областей. О том, как найти недоминирующие точки, мы поговорим позже в этом разделе, а сейчас сосредоточимся на их поведении:

- На рис. 11.4 видно, что $x = -5$ действительно является недоминирующей точкой наряду со многими другими вокруг той области, которая дает высокие значения для второй цели $f_2(x)$. Точки за пределами этой области не дают более высоких

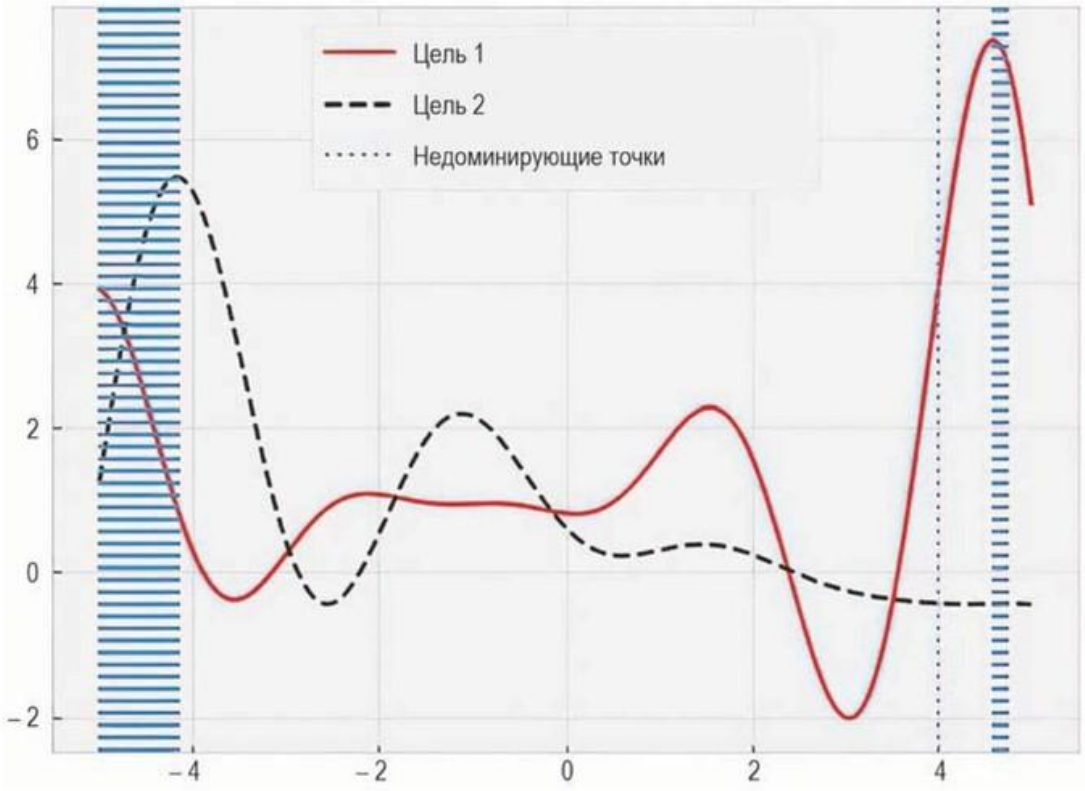


Рис. 11.4. Две целевые функции и недоминирующие точки: в задаче многоцелевой оптимизации существует бесконечно много недоминирующих точек

значений для $f_2(x)$, поэтому точки внутри нее являются недоминирующими. Назовем этот набор точек *группой 1*.

- Меньшая область справа, которая дает высокие значения для первой цели $f_1(x)$, также дает недоминирующие точки. Этот набор назовем *группой 2*.
- Существует третья, наименьшая область вокруг точки $x = 4$ (которая также является недоминирующей). Недоминирующие точки не превышают значения точек в этой области для $f_1(x)$ в крайнем левом углу пространства поиска. Несмотря на то что этот регион не содержит глобального оптимума ни одной из целевых функций, он находится в балансе между их значениями и, следовательно, не является доминирующим. Назовем эти точки *группой 3*.

Недоминирующие точки очень полезны при многоцелевой оптимизации, поскольку они сами по себе являются решениями задачи в силу того, что их нельзя улучшить, не пожертвовав при этом хотя бы одной целью. Кроме того, изучая их взаимоотношения и то, как они распределены в пространстве поиска, можно лучше понять компромисс между несколькими целями в задаче оптимизации. Следовательно, разумная цель многоцелевой оптимизации — найти множество недоминирующих точек.

Однако не сразу понятно, как именно следует количественно определять недоминирующие точки. Стремиться выявить как можно большее их количество бесполезно, поскольку их может быть (часто так и есть) бесконечно много. Вместо этого лучше использовать величину, которую проще представлять, если визуализировать точки данных в другом пространстве.

На рис. 11.3 и 11.4 ось X соответствует самим точкам данных, а ось Y — их объективным значениям. Чтобы изучить компромисс между двумя конфликтующими целями, также можно использовать точечный график, где x-координата определенной точки данных x — это значение для первой цели $f_1(x)$, а y-координата — это значение для второй цели $f_2(x)$.

На рис. 11.5 показан график для каждой точки в плотной сетке из 201 точки, расположенной на равном расстоянии от –5 до 5, где доминирующие точки обозначены «точками», а недоминирующие — «звездочками». Как видно из рисунка, в этом пространстве легче определить, является ли точка доминирующей или нет. Если для каждой точки данных x_1 есть другая точка данных x_2, расположенная выше и правее от x_1, то x_1 является доминирующей. И наоборот, если нет точки x_2, которая одновременно находилась бы выше и правее точки x_1, то x_1 не является доминирующей. Также на рис. 11.5 видно три группы недоминирующих точек, которые соответствуют тем, что обсуждались в отношении рис. 11.4.

Исходя из множества недоминирующих точек, визуализируемых в пространстве целевых значений, теперь можно ввести еще одно понятие: *фронт Парето* (**Pareto frontier**). На рис. 11.6 представлен фронт Парето текущей задачи оптимизации.

Определение

Кривая, проходящая через недоминирующие точки, называется *фронтом Парето*. Он назван именно так, поскольку, когда все точки данных рассматриваются как набор, эта кривая составляет «границу» или «фронт», за пределами которого не лежит ни одной точки данных.

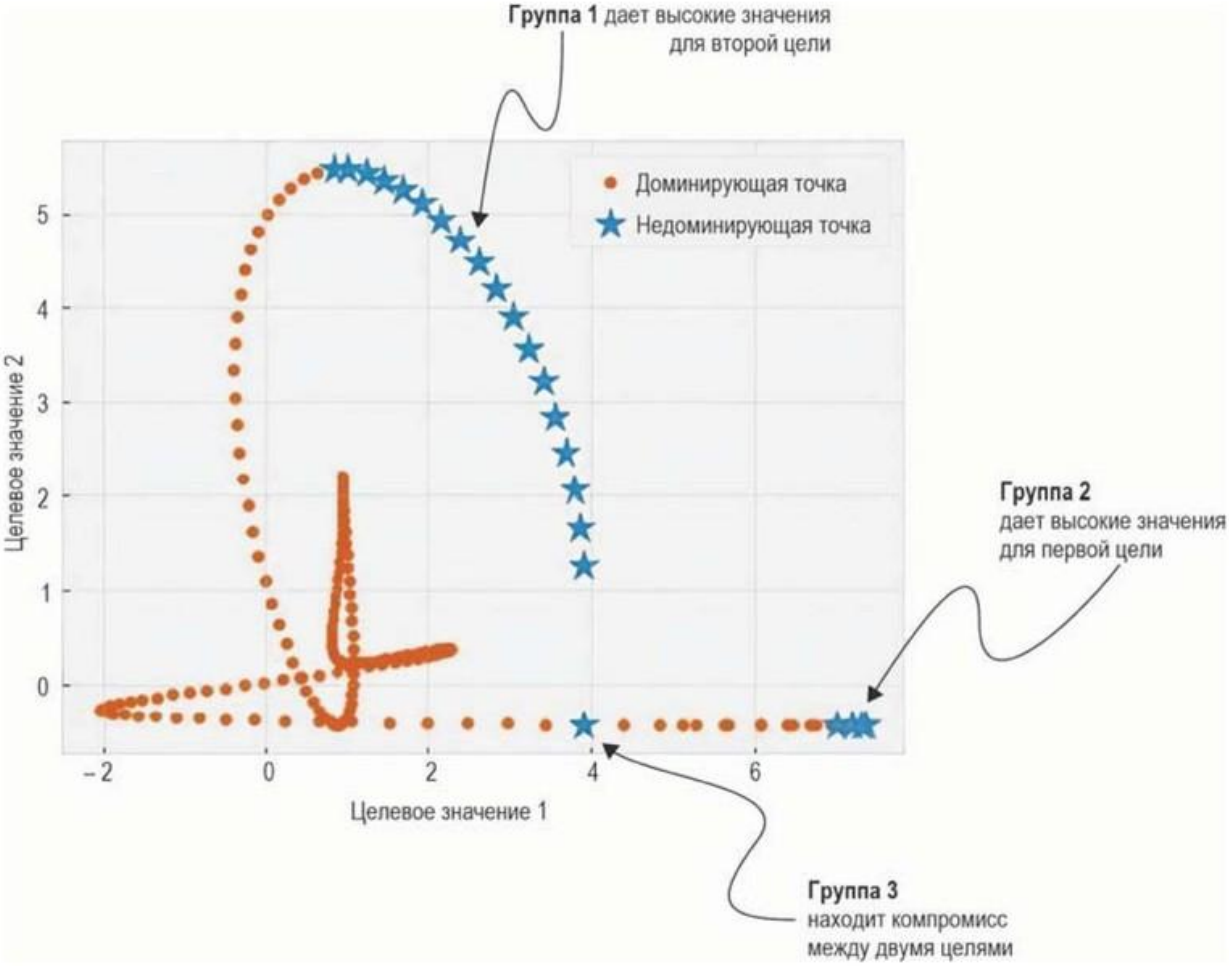


Рис. 11.5. График, основанный на значениях точек данных для двух целевых функций: доминирующие точки обозначены «точками», недоминирующие — «звездочками»; три группы недоминирующих точек соответствуют тем точкам, которые обсуждались на рис. 11.4

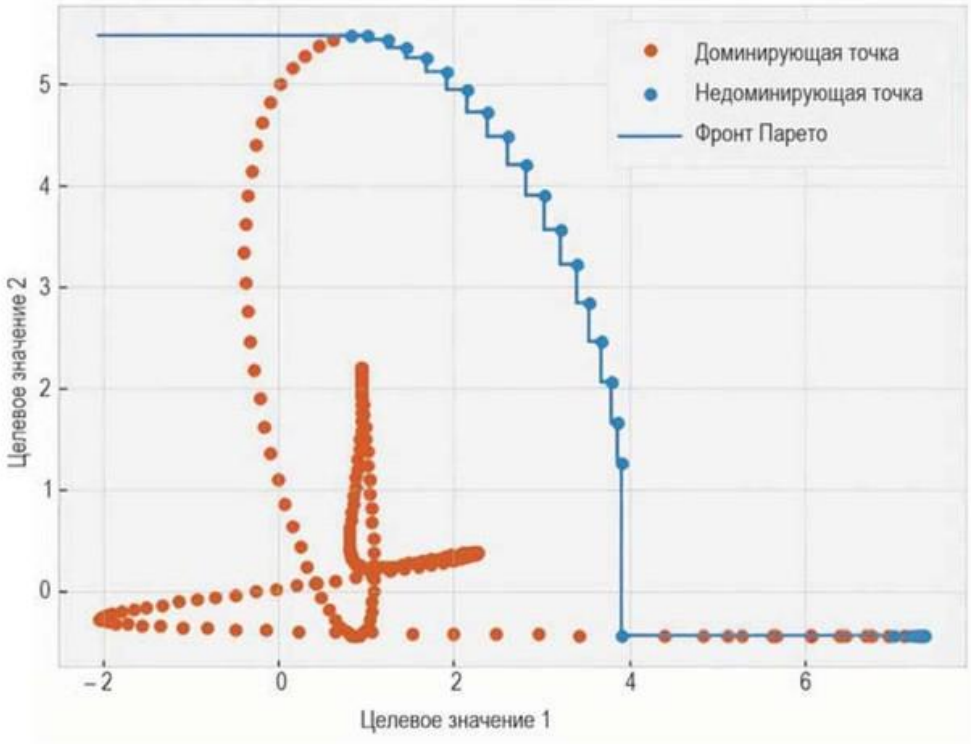


Рис. 11.6. Фронт Парето, проходящий через недоминирующие точки: за пределами фронта нет ни одной точки данных (выше и правее)

Концепция фронта Парето имеет решающее значение в многоцелевой оптимизации, поскольку она напрямую ведет к метрике, которая может количественно оценить прогресс в данной задаче. В частности, мы сосредоточиваемся на том, какую часть пространства (которое определяется совокупностью объективных значений от нескольких целей) охватывает фронт, а именно область внутри (ниже и левее) него. Эта область заштрихована на левой части рис. 11.7.

Определение

Мы используем термин «доминирующий гиперобъем» (**Dominated Hypervolume**) или иногда просто «гиперобъем», чтобы обозначить, какая часть пространства покрыта фронтом Парето. Когда в задаче есть две целевые функции, пространство является двумерным, а гиперобъем — это площадь доминирующей области. Когда целей больше двух, гиперобъем измеряет ту же величину, но в более высоких измерениях.

Разбросанные точки, показанные в левой части рис. 11.7, генерируются с использованием плотной сетки из 201 точки, чтобы можно было детально изучить поведение фронта и ее гиперобъема. Другими словами, эта плотная сетка представляет собой результат исчерпывающего поиска в пространстве для полного отображения фронта Парето.

Для сравнения: правая часть рис. 11.7 демонстрирует результат для набора из 20 точек, которые равномерно выбраны случайным образом между –5 и 5. Из них мы снова находим те, над которыми не доминируют какие-либо другие точки из этого же набора, и рисуем для него фронт Парето.

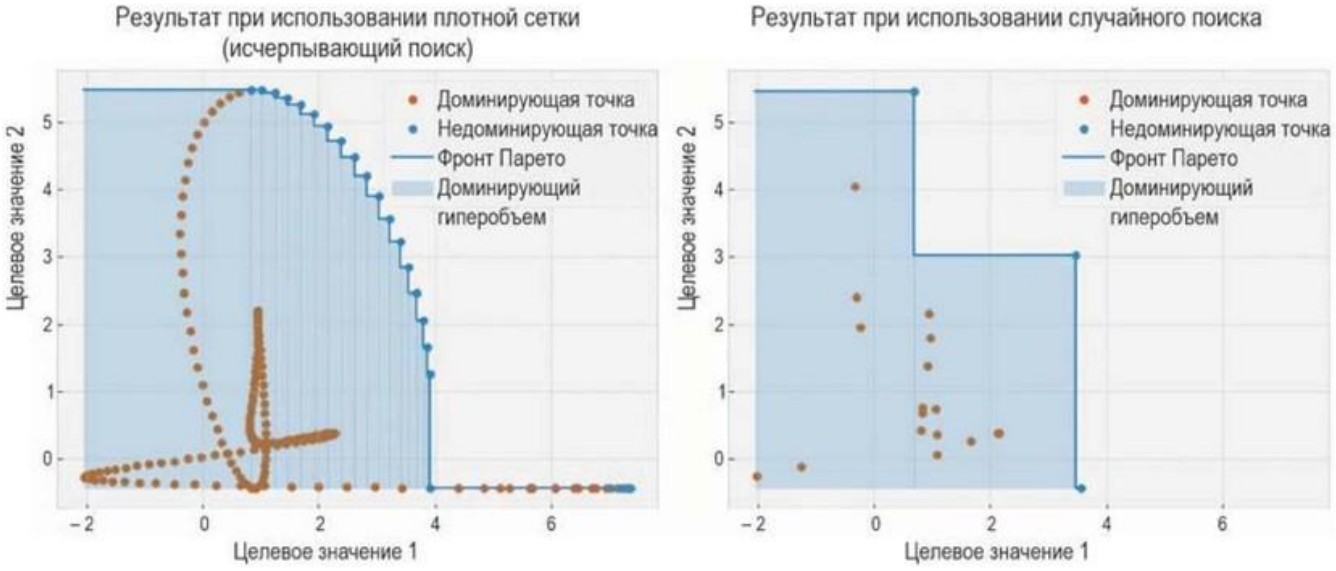


Рис. 11.7. Доминирующий гиперобъем плотной сетки (слева), который эквивалентен исчерпывающему поиску, и случайный поиск (справа) с помощью 20 точек данных, выбранных случайным образом: первый набор имеет больший доминирующий объем и, следовательно, лучше справляется с многоцелевой оптимизацией, чем второй

В отличие от сценария слева, где пространство поиска покрывается полностью, небольшой набор данных справа имеет только четыре недоминирующие точки. В контексте многоцелевой оптимизации первый набор (исчерпывающий поиск) помогает добиться большего прогресса, чем второй (случайный поиск).

Четыре недоминирующие точки из набора справа составляют менее ровный фронт Парето, который, в свою очередь, имеет гораздо меньший доминирующий гиперобъем. Другими словами, мера доминирующего гиперобъема может быть использована для количественной оценки прогресса в задаче многоцелевой оптимизации.

Примечание

В задаче многоцелевой оптимизации мы измеряем прогресс по гиперобъему доминирующей области, который получен на основе текущих собранных данных. Чем он больше, тем больший прогресс в задаче оптимизации достигнут.

Согласно метрике гиперобъема рис. 11.7 показывает, что исчерпывающий поиск дает лучшие результаты, чем случайный (с меньшим количеством запросов), что является ожидаемым результатом. Но чтобы количественно оценить, *насколько* первая стратегия поиска лучше, нужен способ вычислить гиперобъем. Для этого нам необходима контрольная точка. Она будет действовать как конечная точка для доминирующего региона, устанавливая нижнюю левую его границу. Можно считать ее как наихудший возможный результат, который наблюдается в условиях многоцелевой оптимизации. Поэтому гиперобъем области между этой контрольной точкой и фронтом Парето количественно определяет, насколько произошло улучшение по сравнению с наихудшим возможным результатом. Для каждой целевой функции он (если он не известен) может быть установлен на конкретное значение, которое является минимально возможным для достижения в запросе.

Примечание

В многоцелевой оптимизации общей контрольной точкой является *массив*, каждый из элементов которого соответствует наименьшему значению целевой функции, требующей максимизации.

Например, контрольной точкой (ориентиром) для текущей задачи является [−2,0292, −0,4444], поскольку первый элемент (−2,0292) — это минимальное значение первой цели (сплошная кривая на рис. 11.3), а второй элемент (−0,4444) — это минимальное значение второй цели (пунктирная кривая на рис. 11.3). Данная контрольная точка представлена на рис. 11.8 в виде звезды, которая устанавливает нижнюю границу доминирующего пространства.

С помощью контрольной точки можно вычислить гиперобъем доминирующей области для набора данных, собранного с помощью политики многоцелевой оптимизации. Алгоритм для этого вычисления включает в себя разделение доминирующей области на несколько непересекающихся гиперпрямоугольников, которые в совокупности составляют доминирующую область. Отсюда можно легко вычислить гиперобъем каждого гиперпрямоугольника и просуммировать их, чтобы получить гиперобъем всей области. Более подробно этот алгоритм описан в исследовательской работе Рено Лакура, Катрин Клэмрот и Карлоса М. Фонсеки.

С помощью BoTorch можно импортировать и запускать этот алгоритм без необходимости реализации деталей низкого уровня. Например, предположим, что собранные метки, найденные во время оптимизации, сохранены в переменной `train_y`. Поскольку в текущем примере есть две целевые функции, `train_y` должен иметь вид

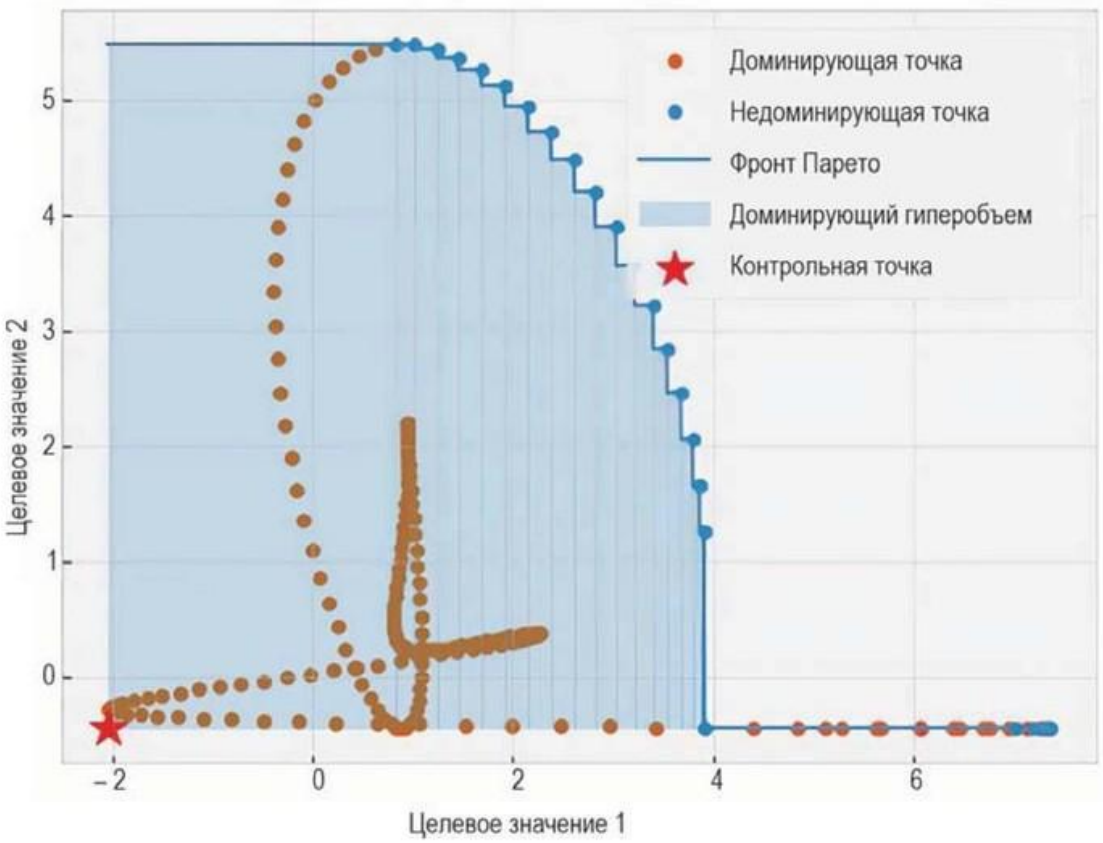


Рис. 11.8. Контрольная точка (ориентир) в задаче многоцелевой оптимизации, которая устанавливает нижнюю границу доминирующего пространства: гиперобъем вычисляется как объем области между контрольной точкой и фронтом Парето

n×2, где n — количество точек данных в собранном наборе. Затем можно использовать код ниже для вычисления гиперобъема, где:

- Класс `DominationPartitioning` реализует разделение доминирующей области. Чтобы инициализировать этот объект, нужно передать контрольную точку и собранные метки в `train_y`.
- Затем нужно вызвать метод `compute_hypervolume()` объекта на доминирующую область, чтобы вычислить ее гиперобъем:

```
#Импортируем реализацию класса доминирующего региона
from botorch.utils.multi_objective.box_decompositions.dominated import
  DominatedPartitioning


#Вычисляем гиперобъем доминирующей области по отношению к контрольной точке
dominated_part = DominatedPartitioning(ref_point, train_y)
volume = dominated_part.compute_hypervolume().item()
```

Используя этот метод, можно вычислить гиперобъемы для исчерпывающего и случайного поиска, как показано на левой и средней частях рис. 11.9. Можно убедиться, что исчерпывающий поиск действительно обеспечивает более высокий гиперобъем (31,49), чем случайный поиск (25,72).

В правой части рис. 11.9 также можно увидеть результат, достигнутый с помощью стратегии БО, которая использует всего 20 точек данных. О ней более подробно

речь пойдет в следующем разделе. Имея лишь 1/10 часть бюджета (20 против 201), БО достигает почти такого же гиперобъема, что и исчерпывающий поиск. А по сравнению со случайным поиском с тем же бюджетом БО способна более полно определить истинный фронт Парето и достичь гораздо большего гиперобъема.

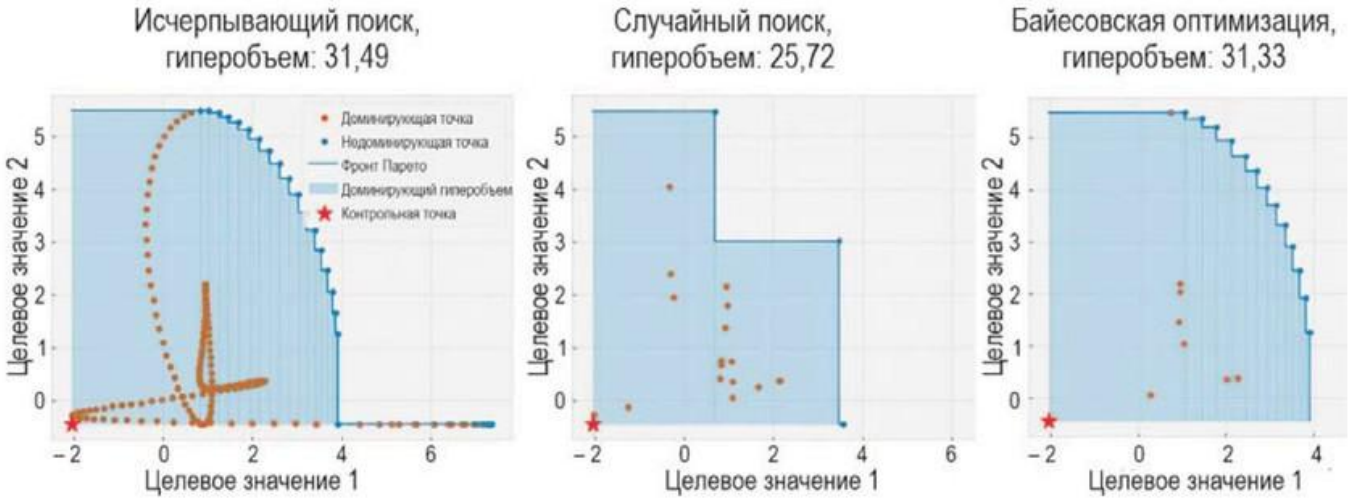


Рис. 11.9. Результаты многоцелевой оптимизации различных стратегий поиска и соответствующих гиперобъемов: БО обеспечивает почти такой же гиперобъем, как и исчерпывающий поиск, но со значительно меньшим количеством запросов

11.3. Стремление к улучшению оптимальной границы данных

Каким образом можно направить политику, чтобы она максимизировала гиперобъем доминирующей области в рамках собранных данных? Простая стратегия состоит в том, чтобы чередовать оптимизацию каждой из целей. На первой итерации цикла мы стремимся максимизировать первую цель $f_1(x)$, на следующей итерации — вторую цель $f_2(x)$ и т. д. Таким образом, во время каждой итерации есть конкретная цель, которую нужно оптимизировать. Этого можно достичь, используя различные политики, изученные в главах с 4 по 6. В оставшейся части главы мы будем использовать ожидаемое улучшение из раздела 4.3. EI — это политика, которая широко используется на практике, благодаря ее алгоритмической простоте и стабильной производительности.

Предположим, что в текущей задаче многоцелевой оптимизации мы выполнили наблюдения в точках, которые обозначены крестиками на верхних частях рис. 11.10. Обучая ГП на наборе данных, принадлежащем каждой из целевых функций, можно получить прогнозы для первой (верхняя левая часть) и второй целей (верхняя правая часть).

На нижних частях рис. 11.10 показаны оценки приобретения отдельных политик EI по соответствующим целевым функциям. Нижний левый EI стремится максимизировать первую цель $f_1(x)$, а нижний правый EI ищет оптимум $f_2(x)$. Также можно убедиться, что конфликт между двумя целями здесь очевиден. Первый EI фокуси-

руется на правой области пространства поиска, где максимизируется $f_1(x)$, а второй EI смотрит на левую область, где максимизируется $f_2(x)$.

Примечание

Оценка приобретения точки данных, рассчитанная с помощью политики, количественно определяет, насколько ценна эта точка для текущего поиска оптимума. Чем больше оценка приобретения, тем выше ценность точки. В свою очередь, точка, дающая наивысшую оценку приобретения, является той, которую политика рекомендует запрашивать.

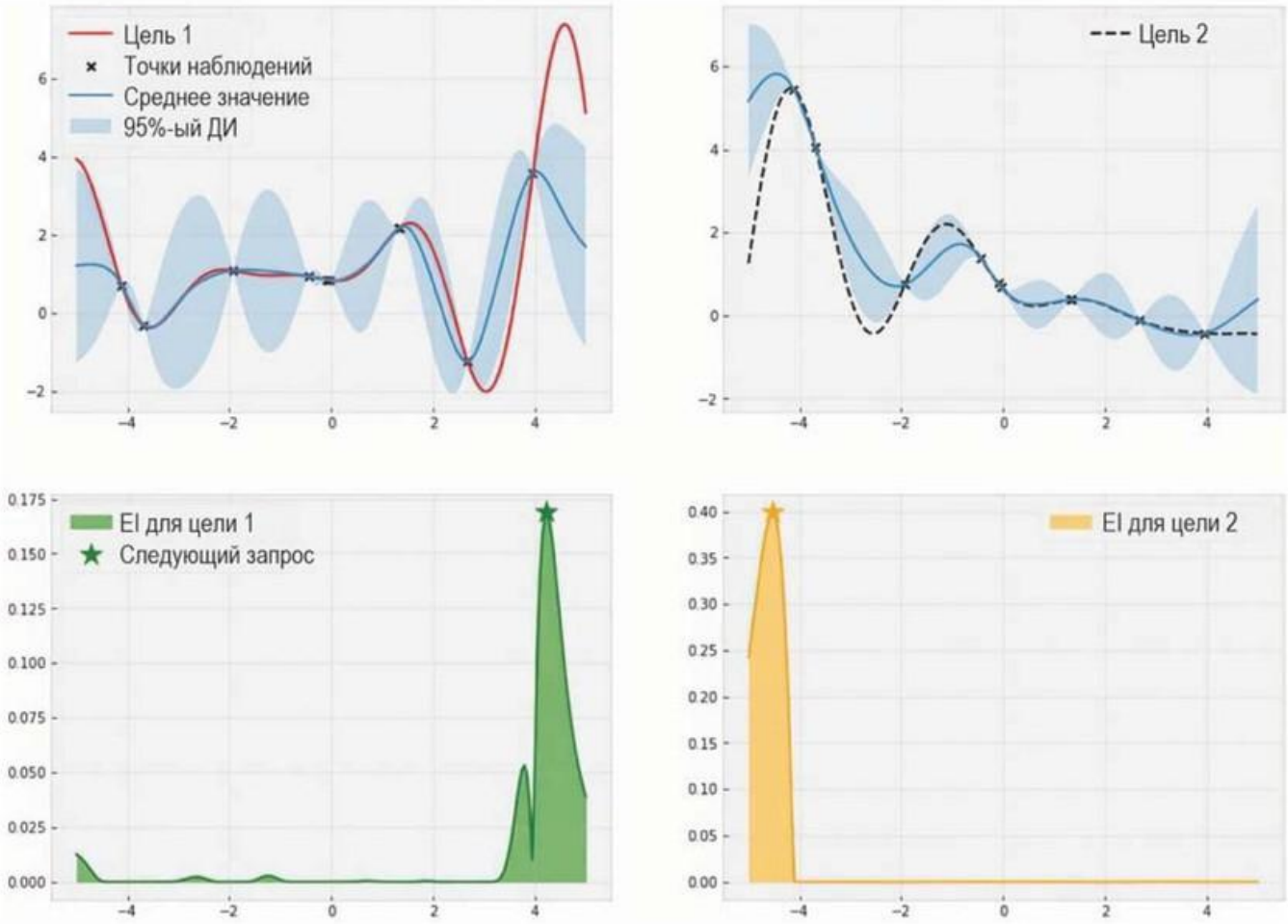


Рис. 11.10. Текущее предположение ГП о двух целевых функциях (вверху) и их соответствующих оценках приобретения (внизу): каждая политика EI направлена на оптимизацию собственной целевой функции и фокусируется на своих регионах

На рис. 11.10 продемонстрирована стратегия, при которой мы либо следуем первой политике и запрашиваем точку возле $x = 4{,}5$, либо второй и запрашиваем точку возле $x = -4{,}5$, в зависимости от того, чья сейчас очередь на оптимизацию: $f_1(x)$ или $f_2(x)$. Данный подход используется в качестве основы для сравнения нашего окончательного решения.

Каким оно должно быть, чтобы можно было добиться результата лучше, чем просто чередовать целевые функции? Стоит заметить, что при наличии ГП для моделирования целей можно вероятностно рассуждать о ценности, которую новый потенциальный запрос может дать для каждой из них *одновременно*. Это значение

(ценности) соответствует уже знакомому нам нормальному распределению, которое является прогнозом о значении запроса. А он, в свою очередь, позволяет рассуждать, является ли каждый новый потенциальный запрос недоминирующей точкой.

И если это так, то насколько увеличится гиперобъем доминирующей области? Каждая вновь наблюдаемая недоминирующая точка расширяет границу этой области (фронт Парето) и, следовательно, увеличивает доминирующий гиперобъем. Таким образом, можно использовать его рост, к которому в среднем приводит каждый новый запрос, в качестве оценки приобретения. Это позволит количественно оценить, насколько запрос ценен. Чем большего увеличения гиперобъема можно ожидать от запроса, тем больше это поможет добиться прогресса в оптимизации.

Конечно, нельзя знать наверняка, какое увеличение гиперобъема получится в результате запроса, пока он не будет сделан к целевым функциям. Однако можно рассуждать об этом увеличении вероятностным образом. А именно — мы можем вычислить *ожидаемое значение* (**Expected value**) для увеличения гиперобъема в результате потенциального запроса.

Подобно алгоритму, определяющему гиперобъем доминирующей области, такой расчет включает в себя деление доминирующей области на гиперпрямоугольники и является довольно сложным. Здесь мы не будем вдаваться в математические детали, но для более подробного знакомства можно обратиться к исследовательской работе Кайфэна Янга, Майкла Эммериха, Андре Дойца и Томаса Бэка. В ней предлагается соответствующая политика БО, которая называется *ожидаемое улучшение гиперобъема*.

Определение

Ожидаемое улучшение гиперобъема (**Expected Hypervolume Improvement**, **EHVI**) использует ожидаемое значение увеличения гиперобъема доминирующей области, к которому приведет новая точка данных, в качестве оценки приобретения этой точки. Политика представляет собой обобщенный вариант EI для многоцелевого сценария, где мы стремимся максимизировать доминирующий гиперобъем.

На рис. 11.11 показаны оценки EHVI (нижняя правая часть) для того же набора данных, что и на рис. 11.10. Можно увидеть, что по сравнению с отдельными политиками EI EHVI прекрасно уравновешивает обе цели, присваивая высокие оценки приобретения нескольким регионам, которые расширяют фронт Парето. Левая область пространства поиска имеет самую высокую оценку.

Мы можем убедиться, что такая стратегия действительно дает преимущество в многоцелевой оптимизации. Для этого реализуем и запустим в текущей задаче. Используемый код включен в `CH11/02 - Multi-objective BayesOptloop.ipynb`.

Для начала нам нужна реализация класса модели и вспомогательная функция `fit_gp_model()`, которая облегчит обучение каждого ГП на наблюдаемых данных. Поскольку эти компоненты уже реализованы в предыдущих главах, их код здесь можно пропустить (его реализацию можно найти в разделе 4.1.1). На каждом этапе цикла БО вызывается вспомогательная функция для инициализации и обучения ГП на данных имеющихся целевых функций. В текущем сценарии мы задействуем две

цели, поэтому вспомогательная функция вызывается дважды: либо с `train_y[:, 0]` (метки $f_1(x)$), либо с `train_y[:, 1]` (метки $f_2(x)$):

```
model1, likelihood1 = fit_gp_model(train_x, train_y[:, 0])
model2, likelihood2 = fit_gp_model(train_x, train_y[:, 1])
```

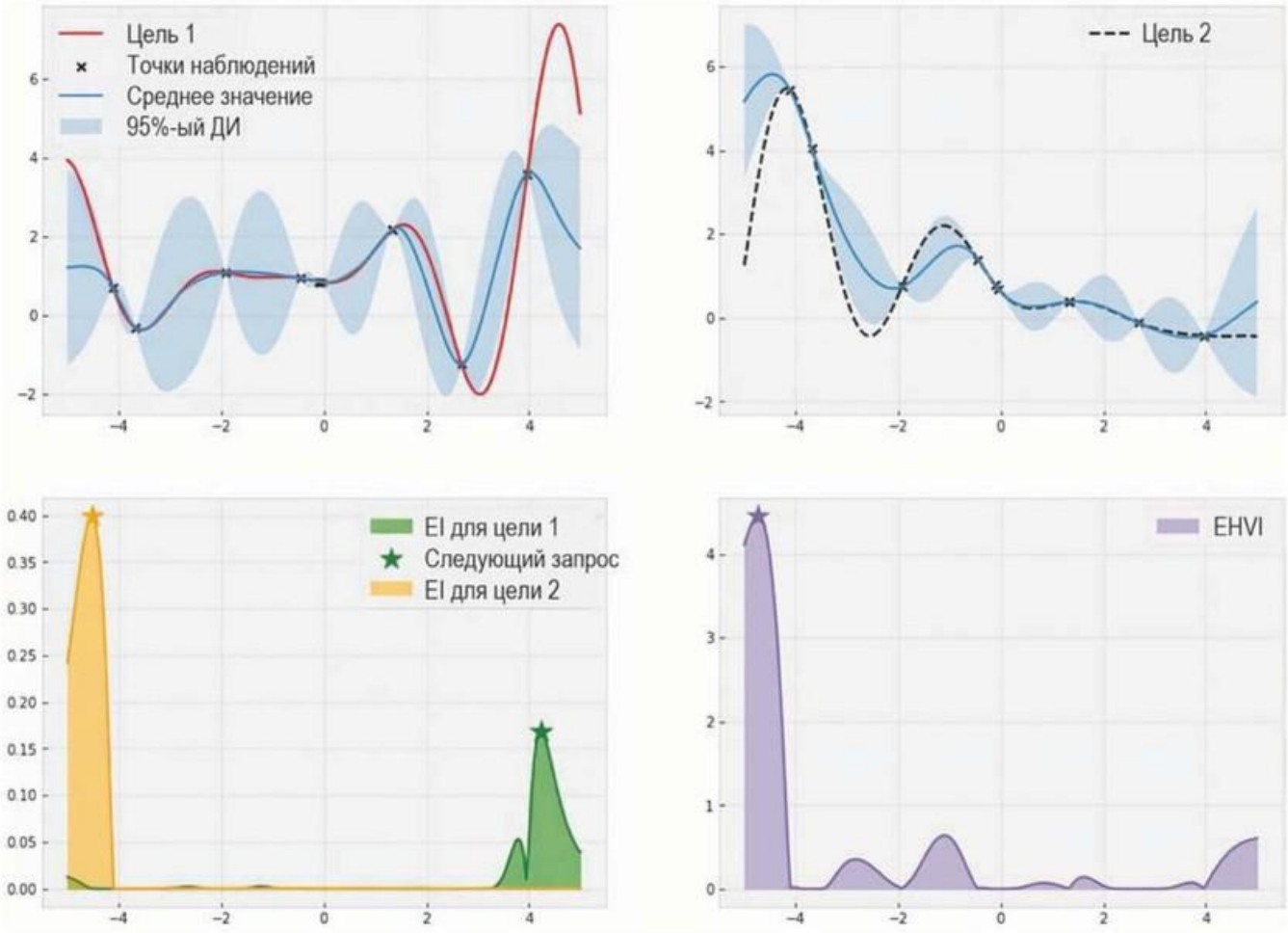


Рис. 11.11. Текущее предположение ГП о каждой из двух целевых функций (вверху), соответствующие оценки приобретения политики EI (внизу слева) и оценки приобретения политики EHVI (внизу справа): EHVI уравновешивает обе цели, присваивая высокие оценки нескольким регионам, которые расширяют фронт Парето

Далее реализуем политику EHVI, используя класс `ExpectedHypervolumeImprovement` из модуля `botorch.acquisition.multi_objective.analytic`. Чтобы инициализировать объект политики, установим следующие аргументы:

- Аргумент `model` принимает список ГП, каждый из которых моделирует целевую функцию. Данный список реализован как экземпляр класса `ModelListGP`, который включает отдельные объекты ГП (`model1`, `model2`).
- Аргумент `ref_point` принимает контрольную точку, которая необходима для расчета гиперобъема и его потенциального увеличения.
- Аргумент `partitioning` использует экземпляр класса `FastNondominationPartitioning`, который облегчает расчет увеличения гиперобъема. Инициализация этого объекта аналогично объекту `DominationPartitioning`, который мы наблюдали ранее, принимает контрольную точку и наблюдаемые метки `train_y`:

```
#Импортируем необходимые классы
from botorch.acquisition.multi_objective.analytic import
  ExpectedHypervolumeImprovement
from botorch.utils.multi_objective.box_decompositions.non_dominated import
  FastNondominatedPartitioning
from botorch.models.model_list_gp_regression import ModelListGP

policy = ExpectedHypervolumeImprovement(
    #Список моделей ГП, по одному для каждой целевой функции
    model=ModelListGP(model1, model2),
    #Контрольная точка
    ref_point=ref_point,
    #Недоминирующий объект partitioning для вычисления увеличения гиперобъема
    partitioning=FastNondominatedPartitioning(ref_point, train_y)
)
```

Используя объект политики для EHVI, можно затем вычислить оценку приобретения, обозначающую ожидаемое увеличение гиперобъема, которое является результатом потенциального нового наблюдения. Далее мы можем найти точку данных, дающую наивысшую оценку, используя вспомогательную функцию `optimize_acqf()`:

```
next_x, acq_val = optimize_acqf(
    policy,
    bounds=bounds,
    q=1,
    num_restarts=20,
    raw_samples=50
)
```

Переменная `next_x` хранит местоположение запроса, с помощью которого мы будем выполнять наши целевые функции: `next_y = joint_objective(next_x)`.

Теперь мы имеем все, что нужно, чтобы запустить EHVI для решения текущей задачи. Для сравнения мы также протестируем стратегию попеременной оптимизации (обсуждалась ранее), в которой будет использована обычная EI при поочередной оптимизации целевых функций. Поскольку целей в текущей задаче две, мы просто будем переключаться между ними (здесь `num_queries` — это общее количество оценок, которые можно выполнить при запуске БО):

```
for i in range(num_queries):
    #Если номер итерации четный, оптимизируем первую цель
    if i % 2 == 0:
        model = model1
        best_f = train_y[:, 0].max()
    #Если номер итерации нечетный, оптимизируем вторую цель
    else:
        model = model2
        best_f = train_y[:, 1].max()

    #Создаем необходимую политику EI согласно номеру итерации
    policy = ExpectedImprovement(model=model, best_f=best_f)
```

Наконец, чтобы количественно оценить прогресс оптимизации, запишем гиперобъем доминирующей области, полученный на основе собранного в ходе поиска текущего набора данных. Запись выполняется с помощью тензора `hypervolumes`, который сохраняет текущий доминирующий гиперобъем на каждом этапе через все множество экспериментов. В целом цикл БО выглядит следующим образом: для каждой политики несколько раз запускается эксперимент с исходным набором данных, выбираемым равномерно случайным образом:

```
#История гиперобъемов, обнаруженных во время оптимизации
hypervolumes = torch.zeros((num_repeats, num_queries))

#Инициализируем начальный обучающий набор, созданный случайный образом
for trial in range(num_repeats):
    torch.manual_seed(trial)
    train_x = bounds[0] + (bounds[1] - bounds[0]) * torch.rand(1, 1)
    train_y = joint_objective(train_x)

#Записываем текущий гиперобъем
for i in range(num_queries):
    dominated_part = DominatedPartitioning(ref_point,train_y)
    hypervolumes[trial, i] = dominated_part.compute_hypervolume().item()

    ... #Переобучаем модели, инициализируем политику, находим следующий запрос
```

В `CH11/02 – Multi-objective BayesOpt loop.ipynb` используются две политики для 10 экспериментов, каждый из которых имеет бюджет в 20 запросов. На рис. 11.12 показаны средние показатели гиперобъема и полосы ошибок в зависимости от количества запросов, выполненных обеими политиками. Можно увидеть, как EHVI постоянно

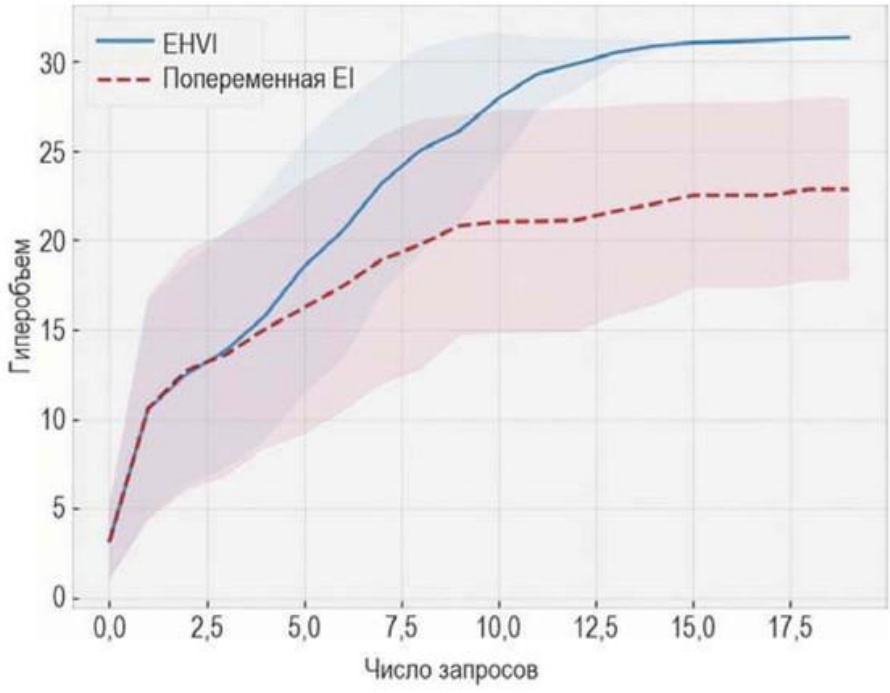


Рис. 11.12. Средний гиперобъем и полосы ошибок в зависимости от количества запросов, выполненных двумя политиками: EHVI неизменно превосходит альтернативную политику EI

превосходит попеременный вариант EI. Это иллюстрирует преимущества подхода, основанного на гиперобъеме.

В этой главе мы узнали о задаче многоцелевой оптимизации и о том, как подойти к ней с помощью БО. Обсудили концепцию гиперобъема в качестве меры эффективности, позволяющей количественно оценить прогресс. Используя вариацию политики EI для оптимизации увеличения гиперобъема, мы получили политику EHVI, которая обеспечивает высокую производительность.

К сожалению, есть и другие аспекты многоцелевого алгоритма БО, которые невозможно охватить в данной главе. В частности, помимо EHVI, можно рассмотреть и другие политики оптимизации. Одним из распространенных методов является *секуляризация* (**Scalarization**), которая объединяет несколько конкурирующих целей в одну с помощью *взвешенной суммы* (**Weighted sum**). Эта стратегия является обобщением попеременного варианта политики EI, которую можно рассматривать с точки зрения присвоения веса 1 для одной цели и веса 0 для другой на каждой итерации. Более подробную информацию о различных политиках многоцелевой оптимизации, предлагаемых BoTorch, можно найти в документации к данной библиотеке.

11.4. Упражнение: Многоцелевая оптимизация для конструирования самолета

В этом упражнении мы применим изученные методы многоцелевой оптимизации к созданию авиастроительной конструкции самолета. Эта задача впервые была представлена в упражнении 2 к главе 7, а также модифицирована как задача с ограничением стоимости в упражнении 2 к главе 8. Здесь мы будем повторно использовать код из главы 8. Это упражнение позволяет нам наблюдать за эффективностью политики ожидаемого улучшения гиперобъема (EHVI) в многомерной задаче. Решение включено в `CH11/03 - Exercise 1.ipynb`.

Выполните следующие шаги:

1. Скопируйте код целевых функций `flight_utility()` и `flight_cost()` из упражнения 2 главы 8. Инвертируйте знак возвращаемого значения второй функции `flight_cost()`. Обе функции будут использоваться в качестве целей для текущей задачи многоцелевой оптимизации.
2. Напишите вспомогательную функцию, которая принимает входные данные `x` (которые могут содержать несколько точек данных) и возвращает значения, вычисленные для двух целевых функций. Возвращаемое значение должно быть тензором размера $n \times 2$, где n — количество точек данных в `x`.
3. Объявите пространство поиска четырехмерным единичным квадратом. То есть четыре нижние границы равны 0, а четыре верхние границы — 1.
4. Чтобы вычислить гиперобъем набора данных, собранного алгоритмом оптимизации, нам нужна контрольная точка. Объявите контрольную точку равной [–1,5, –2], что является соответствующими наименьшими значениями двух целевых функций.

5. Реализуйте класс модели, который должен иметь постоянное среднее значение и четырехмерное ядро Матерна 2.5 с автоматическим определением релевантности (ARD, раздел 3.4.2), и вспомогательную функцию `fit_gp_model()`, которая инициализирует и обучает ГП. Подробную информацию о реализации этих компонентов можно найти в разделе 4.1.1.
6. Установите количество экспериментов, которые необходимо провести, равное 10, а бюджет (количество запросов) в каждом эксперименте — 50.
7. Запустите политику EHVI, чтобы оптимизировать обе целевые функции, а также альтернативную стратегию EI (раздел 11.3). Постройте график среднего гиперобъема и полос ошибок, достигнутых этими политиками (аналогично рис. 11.2), и сравните их производительность.

Заключение

- Задача многоцелевой оптимизации возникает, когда существует несколько потенциально конфликтующих целей, которые необходимо оптимизировать одновременно. Такая задача распространена в реальном мире, поскольку во многих задачах мы часто сталкиваемся с несколькими противоположными целями.
- При использовании БО для многоцелевой оптимизации мы используем несколько ГП для моделирования прогнозов о целевых функциях (одна модель для каждой цели). Мы можем использовать эти ГП для одновременного рассуждения о них вероятностным способом.
- Недоминирующая точка достигает целевых значений, которые невозможно улучшить, если не пожертвовать производительностью хотя бы по одной цели. Обнаружение недоминирующих точек данных является смыслом многоцелевой оптимизации, поскольку они позволяют изучить компромисс между целевыми функциями.
- Недоминирующие точки данных составляют фронт Парето, который устанавливает границу, представляющую оптимальность в многоцелевой оптимизации. За фронтом Парето из всех недоминирующих точек не лежит ни одной точки данных.
- Гиперобъем доминирующего пространства (области, охватываемой фронтом Парето) измеряет эффективность оптимизации набора данных, собранного алгоритмом. Чем больше гиперобъем, тем выше производительность алгоритма. Гиперобъем набора данных можно вычислить, вызвав метод `compute_hypervolume()` в экземпляре класса `DominationPartitioning` из BoTorch.
- Чтобы вычислить гиперобъем набора данных, нужна контрольная точка, которая будет служить конечной точкой доминирующего пространства. Обычно она устанавливается как наименьшие значения целевых функций, подлежащих оптимизации.
- Поскольку ГП позволяют делать прогнозы относительно целевых функций, можно попытаться улучшить гиперобъем текущего набора данных. Эта стратегия соответствует политике EHVI, которая является вариацией EI. Она успешно уравновешивает конкурирующие цели.

Часть IV

Специальные модели гауссовского процесса

Гауссовские процессы вне контекста байесовской оптимизации сами по себе являются мощным классом моделей машинного обучения. Несмотря на то что основная тема этой книги — БО, большой ошибкой будет не уделить ГП больше внимания. В этой части речь пойдет о том, как расширить ГП и сделать их более практичными в различных ML-задачах, сохранив при этом их самую ценную особенность — количественную оценку неопределенности в прогнозах.

В главе 12 мы рассмотрим, как ускорить обучение ГП и масштабировать его для работы с большими наборами данных. Мы также узнаем, как устранить один из самых больших их недостатков — затраты на обучение.

В главе 13 мы узнаем, как вывести гибкость гауссовских процессов на новый уровень, объединив их с нейросетями. Такая комбинация позволит взять лучшее с обеих сторон: способность нейросетей аппроксимировать любую функцию и количественную оценку неопределенности, которую способен дать ГП. Эта глава также поможет по-настоящему оценить наличие оптимизированной экосистемы программного обеспечения в PyTorch, GPyTorch и BoTorch, которая делает совместную работу с нейросетями и ГП беспроблемной.

12 Масштабирование гауссовского процесса для больших наборов данных

Темы главы:

- Обучение ГП на большом наборе данных.
- Использование мини-пакетного градиентного спуска при обучении ГП.
- Использование продвинутого градиентного спуска для более быстрого обучения ГП.

На текущий момент мы можем быть уверены, что ГП способны предложить большую гибкость прогнозирования. В главе 3 мы убедились, что можем моделировать как тенденции высокого уровня, используя функцию среднего ГП, так и изменчивость, используя функцию ковариации. ГП также обеспечивает калиброванную количественную оценку неопределенности. То есть прогнозы для точек, находящихся рядом с наблюдениями в наборе обучающих данных, имеют меньшую неопределенность, чем прогнозы для точек, находящихся далеко. Такая гибкость отличает ГП от других моделей машинного обучения, которые производят только точечные оценки (например, нейросети). Однако за это приходится платить скоростью.

Обучение и составление прогнозов с помощью ГП (в частности, вычисление обратной ковариационной матрицы) масштабируются в кубической пропорции в зависимости от размера обучающих данных. Например, если их размер увеличится вдвое, ГП потребуется в восемь раз больше времени для обучения и прогнозирования. Если набор данных увеличится в десять раз, для ГП это займет в 1000 раз больше времени. Это создает проблему масштабирования ГП на большие наборы данных, что часто встречается во многих приложениях:

- Если нужно смоделировать цены на жилье по всей стране (например, в США), где каждая точка данных представляет цену одного дома в определенный момент времени, тогда размер набора данных будет содержать сотни миллионов точек. В качестве иллюстрации можно привести онлайн-базу данных Statista, которая отслеживает количество единиц жилья в Соединенных Штатах с 1975

по 2021 год. Доступ к этому отчету можно получить на их веб-сайте. Очевидно, что число точек неуклонно растет с 1975 года, превысив 100 миллионов в 1990 году, а сейчас составляет уже более 140 миллионов.

- В приложениях для поиска лекарств, которые обсуждались в разделе 1.1.3, база данных возможных молекул, которые потенциально могут быть синтезированы в лечебные препараты, может содержать миллиарды записей.
- При прогнозировании погоды устройства мониторинга позволяют легко собирать данные о погоде в больших масштабах. Набор данных может содержать поминутные измерения за несколько лет.

Учитывая, что время работы обычной модели увеличивается кубически, невозможно обучить ее на данных такого масштаба. В этой главе мы узнаем, как использовать такой класс моделей ГП, как *вариационный гауссовский процесс* (**Variational Gaussian Process, VGP**), для решения проблемы обучения на больших данных.

> **Определение**
>
> *Вариационный гауссовский процесс* выбирает небольшое подмножество данных, которое хорошо представляет весь набор. Он делает это, стремясь свести к минимуму разницу между собой и обычным ГП, обучающимся на полных данных. Термин *вариационный* относится к разделу математики, изучающему оптимизацию функционалов.

Идея обучения только на небольшом подмножестве репрезентативных точек вполне естественна и интуитивна. На рис. 12.1 показан VGP в действии, где, обучаясь на основе нескольких выборочных точек, модель дает почти такие же прогнозы, как и обычный ГП.

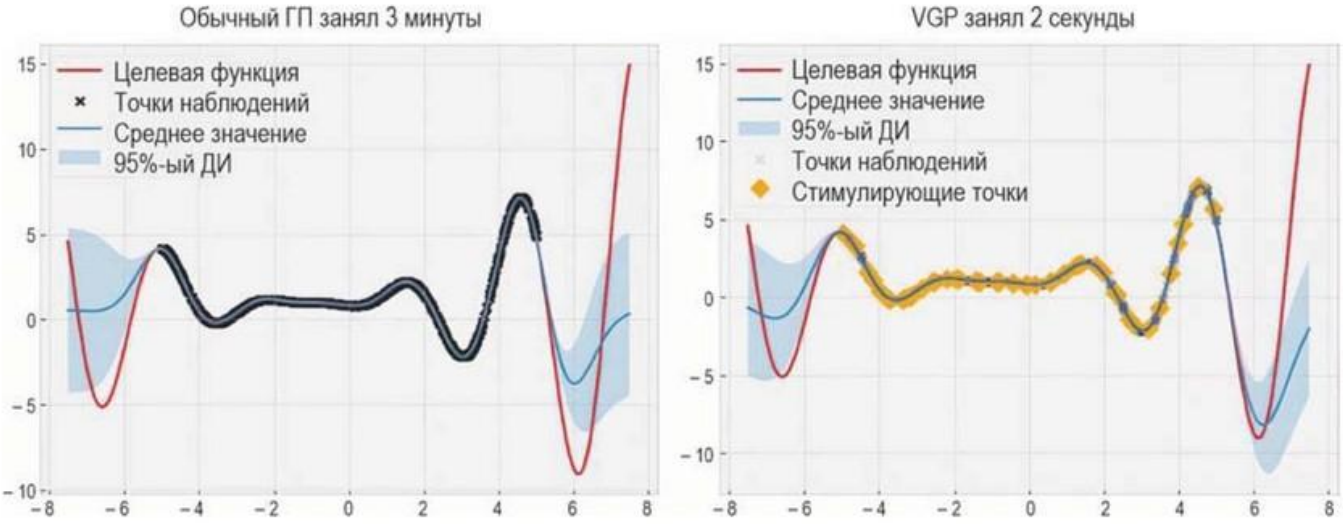


Рис. 12.1. Прогнозы обычного ГП и VGP: VGP дает почти такие же прогнозы, как и ГП, но требует значительно меньше времени на обучение

В этой главе мы расскажем, как реализовать такую модель, а также рассмотрим ее вычислительные преимущества. Далее при работе с VGP мы задействуем более продвинутую версию градиентного спуска, которая (как мы видели в разделе 3.3.2) используется для оптимизации гиперпараметров ГП. Мы учимся использовать эту версию алгоритма для более быстрого и эффективного обучения и в конечном ито-

ге масштабируем имеющиеся ГП для больших наборов данных. Код, сопровождающий эту главу, можно найти в `CH11/01 – Approximate Gaussian Process inference.ipynb`.

12.1. Обучение ГП на большом наборе данных

В этом разделе, чтобы наглядно убедиться, насколько сложной задачей является обучение ГП на большом наборе данных, мы попытаемся применить модель, которая использовалась в главах 2 и 3, к набору данных среднего размера из 1000 точек. Эта задача покажет, что использование обычного ГП в данном сценарии неосуществимо, а также то, что для подобных случаев лучше подойдут вариационные ГП.

12.1.1. Постановка задачи обучения

Сначала создадим набор данных, используя одномерную целевую функцию Форрестера (главы 2 и 3). Реализуем это следующим образом:

```
def forrester_1d(x):
    y = -((x + 1) ** 2) * torch.sin(2 * x + 2) / 5 + 1
    return y.squeeze(-1)
```

Также создадим вспомогательную функцию (аналогично разделу 3.3), которая принимает модель и визуализирует ее прогнозы по всей предметной области. Функция принимает три параметра (модель ГП — `model`, соответствующую функцию вероятности — `likelihood` и логический флаг, обозначающий, является ли модель VGP — `variational=False`) и имеет следующий заголовок:

```
def visualize_gp_belief(model,likelihood,variational=False):
```

Логика вспомогательной функции показана на рис. 12.2, который состоит из четырех основных этапов: вычисление прогнозов, построение основной истины и обучающих данных, построение прогнозов и, наконец, построение стимулирующих точек, если модель является VGP.

> **Определение**
>
> *Стимулирующие точки* (**Inducing point**) — это небольшое подмножество точек, выбранных моделью VGP для представления всего обучающего набора данных. Как следует из названия, эти точки *стимулируют* получение знаний обо всех данных.

Далее рассмотрим эти этапы более подробно. На первом шаге мы вычисляем для целевой функции прогнозы для среднего и ДИ с помощью ГП:

```
with torch.no_grad():
    predictive_distribution = likelihood(model(xs))
    predictive_mean = predictive_distribution.mean
    predictive_upper, predictive_lower =
      predictive_distribution.confidence_region()
```

На втором шаге создаем график с помощью Matplotlib и демонстрируем истинную функцию, хранящуюся в `xs` и `ys` (которые мы вскоре сгенерируем), а также обучающие данные `train_x` и `train_y`:



Рис. 12.2. Блок-схема вспомогательной функции, визуализирующая прогнозы ГП: функция также показывает стимулирующие точки, если передано условие, что модель является VGP

```
plt.figure(figsize=(8, 6))

#Рисуем истинную целевую функцию
plt.plot(xs, ys, label="objective", c="r")
#Создаем точечный график для обучающих данных
plt.scatter(
    train_x,
    train_y,
    marker="x",
    c="k",
    alpha=0.1 if variational else 1,
    label="observations",
 )
```

Если модель представляет собой VGP (для параметра `variational` установлено значение `True`), мы отображаем обучающие данные с меньшей «непрозрачностью» (устанавливая `alpha = 0.1`), что делает их более понятными. Это сделано, чтобы можно было более четко изобразить репрезентативные точки, полученные с помощью VGP позже.

Прогнозы ГП затем отображаются сплошной линией (среднее значение) и заштрихованной областью (95%-ный ДИ) на третьем шаге:

```
plt.plot(xs, predictive_mean, label="mean")
plt.fill_between(
    xs.flatten(),
    predictive_upper,
    predictive_lower,
```

```
    alpha=0.3,
    label="95% CI"
)
```

Наконец, мы наносим на график репрезентативные точки, выбранные VGP, извлекая `model.variational_strategy.inducing_points`:

```
if variational:
    inducing_points =
      model.variational_strategy.inducing_points.detach().clone()
    with torch.no_grad():
        inducing_mean = model(inducing_points).mean

    #Рассеиваем стимулирующие точки
    plt.scatter(
        inducing_points.squeeze(-1),
        inducing_mean,
        marker="D",
        c="orange",
        s=100,
        label="inducing pts"
    )
```

Теперь, чтобы сгенерировать обучающий набор, нужно случайным образом выбрать 1000 точек в области от –5 до 5 и вычислить значения функции в них:

```
torch.manual_seed(0)
train_x = torch.rand(size=(1000, 1)) * 10 - 5
train_y = forrester_1d(train_x)
```

Для этого вычисляем плотную сетку между –7,5 и 7,5, используя функцию `torch.linspace()`. Данный тестовый набор включает в себя значения: –7.5, 7,4, –7.3, ..., 7,5:

```
xs = torch.linspace(-7.5, 7.5, 151).unsqueeze(1)
ys = forrester_1d(xs)
```

Чтобы визуализировать обучающий набор, можно еще раз построить график:

```
plt.figure(figsize=(8, 6))
plt.scatter(
    train_x,
    train_y,
    c="k",
    marker="x",
    s=10,
    label="observations"
)
plt.legend();
```

Данный код создаст рис. 12.3, где черные точки обозначают отдельные точки данных в обучающем наборе.

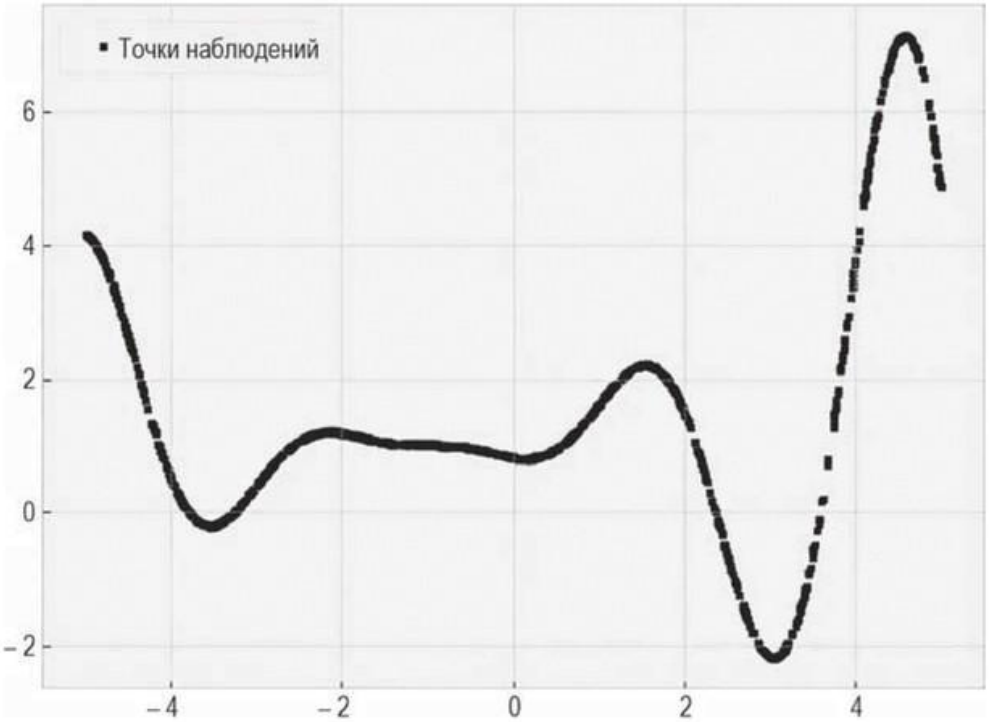


Рис. 12.3. Набор обучающих данных, содержащий 1000 точек: обучение обычного ГП на этом наборе занимает значительное время

12.1.2. Обучение обычного ГП

Теперь мы можем создать и обучить модель на наборе данных. Сначала реализуем класс модели, который имеет постоянную функцию (экземпляр `gpytorch.means.ConstantMean`) в качестве функции среднего значения и ядро RBF с масштабом вывода (реализованный с помощью `gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())`) в качестве ковариационной функции:

```
class GPModel(gpytorch.models.ExactGP):
    #Постоянная функция среднего значения
    def __init__(self, train_x, train_y, likelihood):
        super().__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ConstantMean()
        #Ядро RBF с масштабированным выводом
        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.RBFKernel()
        )

    #Создаем MVN-распределение (нормальное) в качестве прогноза
    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

Далее инициализируем модель с помощью обучающих данных и объекта `GaussianLikelihood`:

```
likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = GPModel(train_x, train_y, likelihood)
```

Затем обучаем ГП, запуская градиентный спуск, чтобы минимизировать функцию потерь, определяемую вероятностью данных. В конце обучения мы получим гиперпараметры модели (например, среднюю константу, масштаб длины и масштаб вывода), которые дают низкое значение потерь. Градиентный спуск реализуем с помощью оптимизатора Adam (`torch.optim.Adam`), который является одним из наиболее часто используемых алгоритмов:

```
#Алгоритм Adam для градиентного спуска
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
#Функция потерь, которая вычисляет вероятность данных на основе гиперпараметров
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

#Включаем обучающий режим
model.train()
likelihood.train()

#Запускаем 500 итераций градиентного спуска
for i in tqdm(range(500)):
    optimizer.zero_grad()

    output = model(train_x)
    loss = -mll(output, train_y)

    loss.backward()
    optimizer.step()

#Включаем режим прогнозирования
model.eval()
likelihood.eval()
```

Следует помнить, что при обучении ГП необходимо включить режим обучения как для модели, так и для вероятности (используя `model.train()` и `likelihood.train()`). А после обучения и перед прогнозами нужно включить режим предсказания (с помощью `model.eval()` и `likelihood.eval()`).

Использование GPU для обучения ГП

Метод масштабирования ГП для больших наборов данных заключается в использовании графических процессоров (GPU). Они часто задействуются для распараллеливания матричных умножений и ускорения обучения нейросетей.

Здесь применяется тот же принцип. GPyTorch упрощает обучение ГП на графических процессорах, следуя синтаксису PyTorch для передачи объектов в GPU (с помощью вызова для объектов метода `cuda()`). В частности, мы вызываем `train_x = train_x.cuda()` и `train_y = train_y.cuda()`, чтобы поместить данные в графический процессор, а также `model = model.cuda()` и `likelihood = likelihood.cuda()`, чтобы туда же поместить модель и ее вероятность.

Более подробную информацию по этой теме можно найти в документации GPyTorch.

Мы выполнили градиентный спуск в 500 итераций, но поскольку текущий набор данных значительно больше, этот цикл может занять некоторое время (так что пока ждете, можете выпить кофе!). После завершения обучения мы вызываем вспомогательную функцию `visualize_gp_belief()`, которая была написана ранее, чтобы показать прогнозы, сделанные обученным ГП (рис. 12.4):



Рис. 12.4. Прогнозы обычного ГП: они хорошо соответствуют обучающим данным, но обучение занимает много времени

На рисунке видно, что прогнозы хорошо соответствуют точкам обучающих данных. Это обнадеживающий признак, что модель успешно обучилась. Однако есть несколько проблем с этой процедурой.

12.1.3. Проблемы с обучением обычного ГП

В этом разделе мы обсудим некоторые проблемы обучения ГП на большом наборе данных. Первая из них, как уже упоминалось, заключается в том, что обучение занимает довольно много времени. На моем MacBook 500 итераций градиентного спуска могут занять до 45 секунд, что значительно дольше, чем мы наблюдали в главах 2 и 3. Это прямой результат того, что время работы ГП увеличивается кубически и становится чрезмерным по мере роста набора данных, как показано в табл. 12.1.

Вторая и, возможно, более серьезная проблема связана с тем, что вычисление функции потерь (предельной логарифмической вероятности обучающих данных), используемой при градиентном спуске, становится все более сложным по мере увеличения размера данных. На это указывают предупреждающие сообщения, которые GPyTorch выдает во время обучения:

```
NumericalWarning: CG terminated in 1000 iterations with average
  residual norm...
```

Таблица 12.1. Расчетное время обучения ГП с учетом размера набора обучающих данных

Размер обучающего набора данных	Время обучения
500 точек	45 секунд
2000 точек	48 минут
3000 точек	2.7 часа
5000 точек	12.5 часа
10 000 точек	4 дня

Эти сообщения говорят, что мы сталкиваемся с численной нестабильностью при расчете потерь.

> **Примечание**
>
> Расчет потерь по многим точкам данных является вычислительно нестабильной операцией.

Численная нестабильность не позволяет правильно их рассчитать и, следовательно, эффективно минимизировать. *Как* потери меняются на протяжении 500 итераций градиентного спуска, продемонстрировано на рис. 12.5.

В отличие от того, что мы наблюдали в главах 2 и 3, потери здесь хаотично скачут вверх и вниз, указывая на то, что градиентный спуск не справляется с задачей их

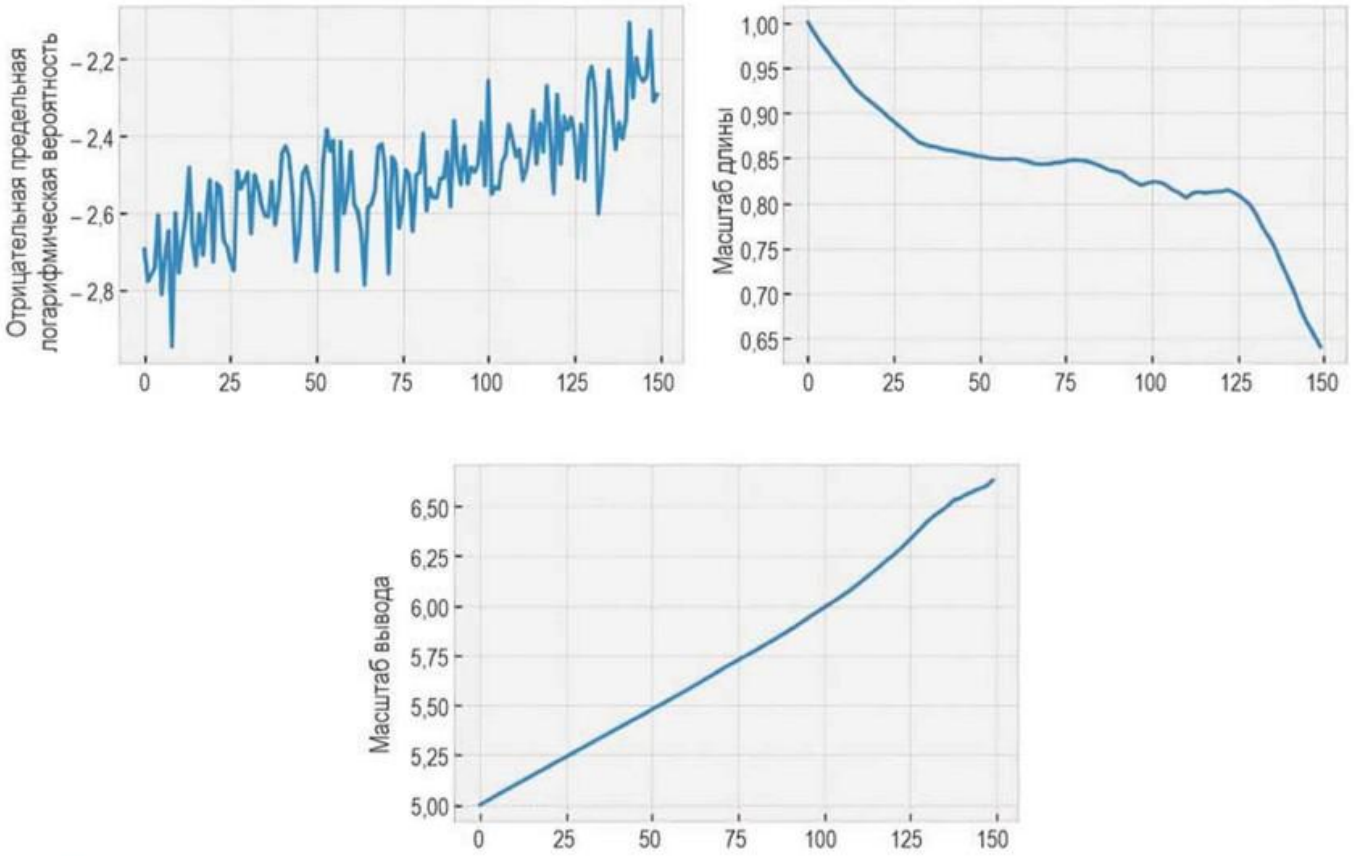


Рис. 12.5. Прогрессирующие потери обычного ГП во время градиентного спуска: из-за численной нестабильности кривая потерь является неровной и не может эффективно минимизироваться

минимизации. Фактически по мере прохода все большего числа итераций потери увеличиваются. Это означает, что модель не оптимальна. Такое явление вполне объяснимо: если неправильно рассчитать потери модели, а затем использовать этот расчет для управления обучением при градиентном спуске, мы, скорее всего, получим неоптимальное решение.

Возможно, привычнее будет провести аналогию между градиентным спуском и спуском с горы, как показано на рис. 12.6. Допустим, вы находитесь на вершине и у вас завязаны глаза. Вы хотите оказаться внизу. На каждом шаге вы находите направление движения, которое позволит вам добраться до более низкого места (т. е. спуститься). В конце концов, сделав достаточно шагов, вы достигнете подножия. Аналогично при градиентном спуске мы начинаем с относительно высоких потерь и, корректируя гиперпараметры модели на каждой итерации, постепенно уменьшаем их. При достаточном количестве итераций мы приходим к оптимальной модели.



Рис. 12.6. Выполнение градиентного спуска с численно нестабильным расчетом потерь похоже на спуск с горы с завязанными глазами

Примечание

Более подробную информацию о градиентном спуске, а также почему он аналогичен спуску с горы, можно найти в книге Луиса Серрано *«Grokking Machine Learning»* (*«Обучение машины Гроккинга»*).

Этот процесс работает хорошо только в том случае, когда можно точно вычислить потери, а именно — если есть возможность точно увидеть, в каком направлении нужно шагать, чтобы попасть на место пониже. Однако если такие вычисления подвержены ошибкам, мы не сможем эффективно минимизировать потери модели.

Это похоже на попытку спуститься с горы с завязанными глазами. Как видно из рис. 12.5, мы фактически оказались выше (потери превышают значение до градиентного спуска).

В целом обучение обычного ГП на большом наборе данных — это не лучший подход. Затраченное время не только увеличивается кубически в зависимости от размера данных, но и вычисление потерь, которое необходимо оптимизировать, также является нестабильным. В оставшейся части этой главы мы рассмотрим вариационные ГП (VGP) в качестве решения данной проблемы.

12.2. Автоматический выбор репрезентативных точек из большого набора данных

Идея VGP состоит в том, чтобы выбрать несколько точек, которые репрезентативны для всего набора, и обучить на них ГП. Мы уже научились хорошо делать это с обычным ГП на небольшом наборе данных. Есть вероятность, что меньшее подмножество репрезентативных точек отражает общую тенденцию всего набора, поэтому при обучении на нем потери информации минимальны.

Такой подход вполне естествен. Большой набор обычно содержит избыточную информацию, поэтому, если есть возможность учиться только на наиболее информативных точках, можно избежать необходимости обрабатывать такую избыточность. В разделе 2.2 отмечалось, что ГП, как и любая ML-модель, работает при предположении, что схожие точки данных создают схожие метки. Когда большой набор содержит много похожих точек, для ГП достаточно сосредоточиться только на одной из них, чтобы узнать об их общей тенденции. Например, несмотря на наличие поминутных данных о погоде, прогностическая модель может вполне эффективно учиться на основе почасовых измерений. В этом разделе мы узнаем, как сделать это автоматически, убедившись, что обучение на небольшом подмножестве приводит к минимальным потерям информации по сравнению с обучением на большом наборе. А также реализуем эту модель с помощью GPyTorch.

12.2.1. Минимизация различий между двумя ГП

Как лучше всего выбрать подмножество, чтобы окончательная модель могла получить наибольшее количество информации из исходного набора данных? В этом разделе мы обсудим общую идею, как это сделать с помощью VGP. Данный процесс можно приравнять к поиску подмножества стимулирующих точек, которые при обучении на меньшем подмножестве будут стимулировать апостериорный ГП.

Углубляясь в некоторые математические детали, при обучении VGP мы стремимся минимизировать разницу между двумя апостериорными ГП: один обучен на стимулирующих точках, а другой — на всем наборе данных. Для этого требуется измерить разницу между двумя распределениями (двумя ГП). Используемая для этого мера — *дивергенция Кульбака — Лейблера* (**Kullback — Leibler divergence**, **KL-расхождение**).

Определение

Дивергенция Кульбака — Лейблера представляет собой статистическое расстояние, которое измеряет расхождение между двумя распределениями. Другими словами, KL-расхождение вычисляет, насколько одно распределение вероятностей отличается от другого.

Дополнительный материал по KL-расхождению

Интуитивное объяснение KL-расхождения можно найти в превосходной записи из блога Уилла Курта «Объяснение дивергенции Кульбака — Лейблера». Также дополнительную информации с математической точки зрения можно найти в главе 2 книги Дэвида Маккея *«Information Theory, Inference, and Learning Algorithms»* (*«Теория информации, выводы и алгоритмы обучения»*, издательство Cambridge University Press, 2003).

Аналогично тому, как евклидово расстояние (длина отрезка, соединяющего две точки) измеряет, насколько далеко друг от друга находятся точки A и B, KL-расхождение измеряет, насколько далеко друг от друга находятся два заданных распределения в пространстве вероятностных распределений. Иначе говоря, насколько они отличаются друг от друга. Это показано на рис. 12.7.

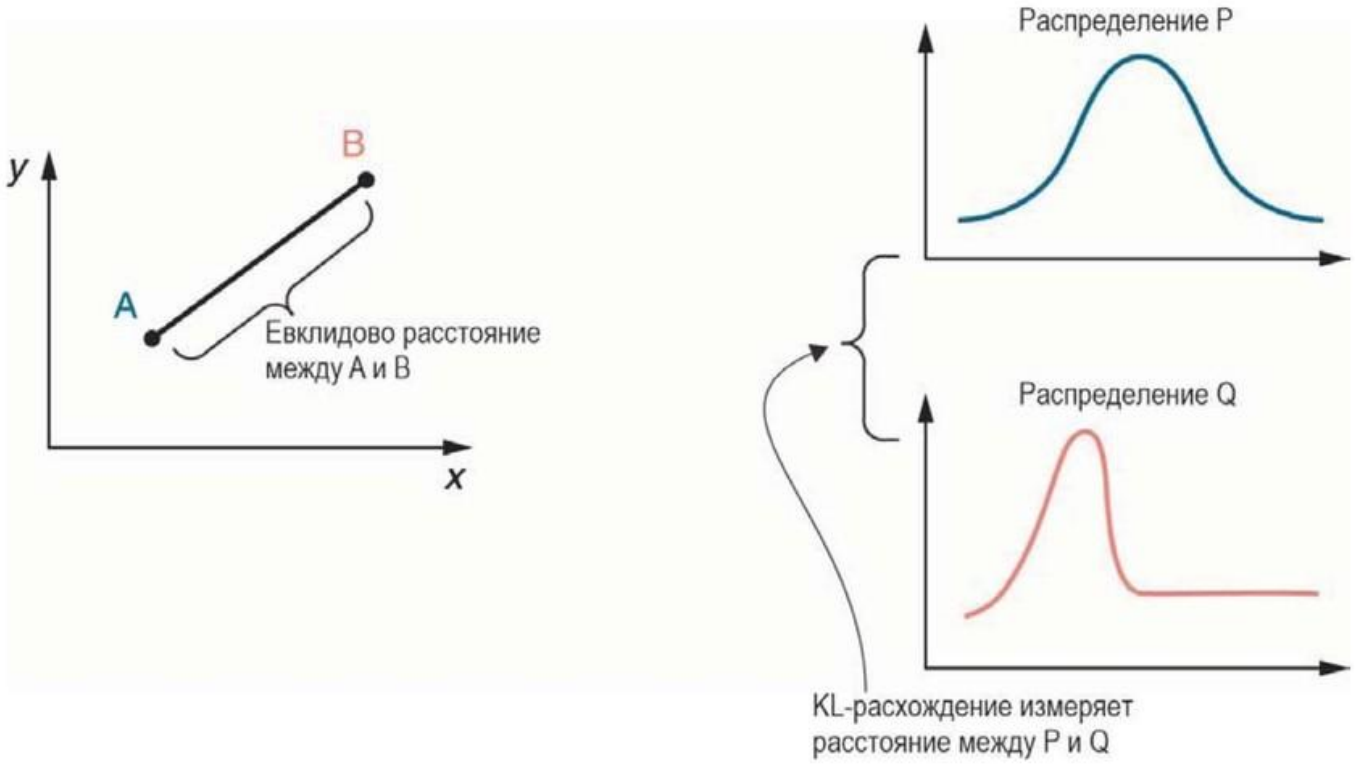


Рис. 12.7. Евклидово расстояние измеряет расстояние между двумя точками на плоскости, а KL-расхождение измеряет расстояние между двумя распределениями вероятностей

Примечание

В качестве математически обоснованной меры KL-расхождение является неотрицательным. Другими словами, расстояние между двумя любыми распределениями не может быть меньше нуля. А когда оно равно нулю, два распределения точно совпадают.

Для вычисления KL-расхождения между двумя апостериорными ГП следует выбирать стимулирующие точки, которые сводят это расхождение к нулю. Как и нахож-

дение предельной логарифмической вероятности, расчет KL-расхождения также вычислительно нестабилен. Однако благодаря его математическим свойствам можно переписать KL-расхождение в виде разницы между двумя величинами, как показано на рис. 12.8.



Рис. 12.8. KL-расхождение можно разложить на разницу между предельной логарифмической вероятностью и ELBO: ELBO легко вычислить, поэтому он выбран в качестве метрики, подлежащей оптимизации

Третий член в этом уравнении — *нижняя граница доказательств* (**Evidence Lower Bound**, **ELBO**). Он представляет собой разницу между предельной логарифмической вероятностью и KL-расхождением. Несмотря на то что эти два термина трудно вычислить, ELBO имеет простую форму и может быть легко посчитан. По этой причине вместо минимизации KL-расхождения можно максимизировать ELBO как способ косвенно максимизировать предельную логарифмическую вероятность, чтобы апостериорный ГП, обученный на стимулирующих точках, был как можно ближе к обученному на полном наборе данных.

ELBO = marginal log likelihood − KL divergence

ELBO ≤ marginal log likelihood

Поскольку KL-расхождение всегда больше или равно 0, предельная логарифмическая вероятность больше или равна ELBO. Ее называют нижней границей доказательства, поскольку она является нижней границей логарифмической вероятности модели.

Подведем итог. Чтобы найти набор стимулирующих точек, которые дадут апостериорный ГП, максимально похожий на ГП, который возможно получить при обучении на полном наборе данных, мы стремимся минимизировать KL-расхождение между двумя этими ГП. Однако данное расхождение трудно вычислить, поэтому проще оптимизировать нижнюю границу доказательства (ELBO) модели, поскольку ее легче посчитать. В следующем разделе мы увидим, что GPyTorch может предоставить удобную функцию потерь, которая вычисляет ELBO за нас. Прежде чем рассмотреть данную реализацию, сначала обсудим еще одну вещь: как учесть все точки данных в большом обучающем наборе при максимизации члена ELBO.

12.2.2. Обучение модели небольшими пакетами

Поскольку мы стремимся найти подмножество стимулирующих точек, которое лучше всего представляет весь набор обучающих данных, нам по-прежнему необходимо включить все точки в обучающий набор при вычислении ELBO. Но ранее мы говорили, что вычисление предельной логарифмической вероятности для многих точек численно нестабильно, поэтому градиентный спуск становится неэффективным. Сталкиваемся ли мы здесь с той же проблемой? В этом разделе мы уви-

дим, что при обучении VGP путем оптимизации члена ELBO можно избежать проблемы числовой нестабильности, используя модифицированную версию градиентного спуска, которая наиболее подходит для больших наборов данных.

Задача вычисления функции потерь для ML-модели по многим точкам данных не уникальна для ГП. Например, нейросети обычно обучаются на тысячах и миллионах точек, и вычислить функцию потерь для всех них также невозможно. Решение данной проблемы как для нейросетей, так и для VGP состоит в том, чтобы аппроксимировать истинное значение потерь для всех точек данных, используя значение потерь для случайного подмножества точек. Например, следующий фрагмент кода взят из официальной документации PyTorch и показывает, как обучить нейросеть на наборе данных изображений. Здесь внутренний цикл перебирает небольшие подмножества обучающих данных и выполняет градиентный спуск по значениям потерь, вычисленным для этих подмножеств:

```
#Проходим по набору данных многократно
for epoch in range(2):

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        #Входные данные в виде списка [inputs, labels]
        inputs, labels = data

        #Обнуляем градиенты параметров
        optimizer.zero_grad()

        #Шаг вперед + шаг назад + оптимизация
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

При расчете потерь по небольшому количеству точек вычисления могут выполняться стабильно и эффективно. Кроме того, повторяя такое приближение много раз, можно хорошо аппроксимировать истинные потери. Затем остается запустить градиентный спуск по этим аппроксимированным потерям, что также минимизирует истинные потери во всех точках данных.

> **Определение**
>
> Технику градиентного спуска с учетом потерь, вычисленных со случайным подмножеством данных, иногда называют *мини-пакетным градиентным спуском* (**Mini-batch Gradient descent**). На практике вместо случайного выбора на каждой итерации алгоритма обучающий набор разбивается на небольшие подмножества, а затем итеративно вычисляются приблизительные потери, используя каждое из небольших подмножеств.

Например, если обучающий набор содержит 1000 точек, можно разделить его на 10 небольших подмножеств по 100 точек. Затем вычислить потери для каждого из них при градиентном спуске и повторить процедуру для всех 10 подмножеств

(именно это мы будем делать в примере кода позже). Несмотря на то что приблизительные потери, вычисленные на основе подмножества данных, не совсем равны истинным потерям, при градиентном спуске такое приближение повторяется много раз, что указывает на правильное направление спуска.

Разница между градиентным спуском, минимизирующим истинные потери, и мини-пакетным градиентным спуском, минимизирующим приблизительные потери, показана на рис. 12.9. По сравнению с обычным градиентным спуском (который, опять же, невозможно использовать с большими данными) мини-пакетная версия может не указывать на наиболее эффективное направление спуска, но, повторяя аппроксимацию несколько раз, мы все равно можем достичь цели.



Рис. 12.9. Иллюстрация обычного и мини-пакетного градиентных спусков в «долине потерь», где центр долины дает наименьшие потери: обычный градиентный спуск, если его возможно вычислить, ведет прямо к цели, в то время как мини-пакетная версия идет в неоптимальных направлениях, но в конце все равно достигает цели

Если бы рассуждения касались аналогии «спуска с горы с завязанными глазами», мини-пакетный градиентный спуск был бы похож на то, как будто глаза завязаны тонкой тканью, сквозь которую можно частично видеть. Не всегда есть гарантия, что с каждым нашим шагом мы окажемся ниже, но при достаточном количестве времени мы сможем успешно спуститься.

> **Примечание**
>
> Не все функции потерь можно аппроксимировать потерями на подмножестве данных. Проще говоря, не все функции потерь можно минимизировать с помощью мини-пакетного градиентного спуска. Примером может служить отрицательная предельная логарифмическая вероятность ГП. Иначе мы могли бы запустить мини-пакетный градиентный спуск и для этой функции. Но, к счастью, он вполне применим к ELBO для VGP.

Подведем итог. Обучение VGP происходит примерно так же, как и обучение обычного ГП, в котором используется мини-пакетная версия градиентного спуска, чтобы минимизировать соответствующие потери модели. В табл. 12.2 суммированы ключевые различия между двумя классами моделей: обычный ГП следует обучать на небольших наборах данных с помощью градиентного спуска, чтобы минимизировать точную отрицательную предельную логарифмическую вероятность, тогда как VGP можно обучать на больших наборах данных с помощью мини-пакетного

градиентного спуска для оптимизации ELBO, который является аппроксимацией истинной логарифмической вероятности.

Таблица 12.2. Обучение ГП и VGP: на высоком уровне процедуры аналогичны, заменяются только определенные компоненты и настройки

Процедура обучения	ГП	VGP
Размер обучающих данных	Маленький	От среднего до большого
Тип обучения	Точное обучение	Приблизительное обучение
Функция потерь	Отрицательная предельная логарифмическая вероятность	Нижняя граница доказательства (ELBO)
Оптимизация	Градиентный спуск	Мини-пакетный градиентный спуск

12.2.3. Реализация приближенной модели

Теперь мы можем реализовать вариационный ГП в GPyTorch. План состоит в том, чтобы написать класс модели VGP, который похож на уже известные нам классы модели обычного ГП, и минимизировать ELBO с помощью мини-пакетного градиентного спуска. Различия в рабочем процессе, описанном в табл. 12.2, отражены в коде для этого раздела. В табл. 12.3 показаны компоненты, необходимые при реализации ГП и VGP в GPyTorch. Помимо модулей среднего значения и ковариации, VGP требует также два дополнительных компонента:

- *Вариационное распределение* (**Variational distribution**) — определяет распределение по стимулирующим точкам для VGP. Как известно из предыдущего раздела, это распределение необходимо оптимизировать так, чтобы VGP напоминал обычный ГП, обученный на полном наборе данных.
- *Вариационная стратегия* (**Variational strategy**) — определяет, как создаются прогнозы на основе стимулирующих точек. Из раздела 2.2 известно, что многомерное нормальное распределение (MVN) может быть обновлено по мере наблюдения. Вариационная стратегия обеспечивает такое же обновление вариационного распределения.

Таблица 12.3. Необходимые компоненты при реализации ГП и VGP в PyTorch

Компоненты	ГП	VGP
Модуль среднего значения	Да	Да
Модуль ковариационной функции	Да	Да
Вариационное распределение	Нет	Да
Вариационная стратегия	Нет	Да

Имея на руках эти компоненты, можно реализовать класс модели VGP, который будет называться `ApproximateGPModel`. Обучающие данные и функция вероятности

больше не будут использоваться в методе `__init__()`. Вместо этого мы возьмем набор стимулирующих точек, которые будут использоваться для представления всего набора данных. Остальная часть метода `__init__()` будет состоять из объявления конвейера обучения, который будет использоваться для определения, какой набор стимулирующих точек является лучшим:

- Переменная `variation_distribution` — это экземпляр класса `CholeskyVariationalDistribution`, который принимает количество стимулирующих точек во время инициализации. Вариационное распределение является основой VGP.
- Переменная `variational_strategy` — экземпляр класса `VariationalStrategy`. Он принимает набор стимулирующих точек, а также вариационное распределение. Мы устанавливаем `learn_inducing_locations = True`, чтобы можно было изучить лучшие места для стимулирующих точек во время обучения. Если для этой переменной установлено значение `False`, точки, переданные в `__init__()` (хранящиеся в `inducing`), будут использоваться в качестве стимулирующих точек:

```
#Вместо объекта ExactGP наш VGP является приблизительным ГП
class ApproximateGPModel(gpytorch.models.ApproximateGP):
    #Принимает набор начальных стимулирующих точек
    def __init__(self, inducing_points):
        #Устанавливаем вариационные параметры, необходимые для обучения
        variational_distribution =
          gpytorch.variational.CholeskyVariationalDistribution(
            inducing_points.size(0)
        )
        variational_strategy = gpytorch.variational.VariationalStrategy(
            self,
            inducing_points,
            variational_distribution,
            learn_inducing_locations=True,
        )
        super().__init__(variational_strategy)

        ... #Продолжение следует
```

На последнем этапе метода `__init__()` объявим функции среднего и ковариации для VGP. Они должны быть такими, какими мы хотели бы их видеть в обычном ГП, если бы он обучался на полных данных. В этом случае мы будем использовать постоянное среднее значение и ядро RBF с масштабом вывода:

```
class ApproximateGPModel(gpytorch.models.ApproximateGP):
    def __init__(self, inducing_points):
        ...

        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(
          gpytorch.kernels.RBFKernel()
        )
```

Также мы объявили метод `forward()` (который здесь не показан) как и в обычном ГП. Далее инициализируем модель, используя первые 50 точек данных из обучающего набора в качестве стимулирующих точек:

```
#Срез тензора train_x[:50, :] дает первые 50 точек данных в train_x
model = ApproximateGPModlel(train_x[:50, :])
likelihood = gpytorch.likelihoods.GaussianLikelihood()
```

В первых 50 точках данных нет ничего особенного. Их значения, хранящиеся внутри модели VGP, будут изменены во время обучения. Самая важная часть данной инициализации — мы указываем, что модель должна использовать 50 стимулирующих точек. Если нужно использовать 100 точек, можно передать `train_x[:100, :]`.

Трудно сказать точно, какого количества стимулирующих точек будет достаточно для VGP. Чем меньше их используется, тем быстрее обучается модель, но тем менее эффективно эти точки будут представлять весь набор. По мере увеличения количества стимулирующих точек у VGP появляется больше свободы в распределении их на весь набор, но обучение при этом становится медленнее.

> **Примечание**
>
> Главное правило — не превышать 1000 стимулирующих точек, 50 вполне достаточно, чтобы с высокой точностью аппроксимировать обученный ГП из предыдущего раздела.

Чтобы настроить мини-пакетный градиентный спуск, нам нужен оптимизатор. Мы снова задействуем Adam:

```
optimizer = torch.optim.Adam(
    [
        {"params": model.parameters()},
        #Оптимизируем параметры вероятности вместе с параметрами ГП
        {"params": likelihood.parameters()}
    ],
    lr=0.01
)
```

Параметры для оптимизации

Раньше нам нужно было передавать алгоритму Adam только `model.parameters()`. Здесь вероятность не связана с моделью VGP. Обычный ГП инициализируется с помощью вероятности, а VGP — нет. В этом случае необходимо передать в Adam `likelihood.parameters()`.

Для функции потерь мы будем использовать класс `gpytorch.mlls.VariationalELBO`. Он реализует величину ELBO, которую мы стремимся оптимизировать с помощью VGP. Во время инициализации экземпляр этого класса принимает функцию вероятности, модель VGP и размер полного обучающего набора (доступ к которому можно получить с помощью `train_y.size(0)`). При этом этот объект объявляется следующим образом:

```
mll = gpytorch.mlls.VariationalELBO(
    likelihood,
    model,
    num_data=train_y.size(0) #Размер обучающих данных
)
```

После настройки модели, оптимизатора и функции потерь нужно запустить мини-пакетный градиентный спуск. Для этого набор обучающих данных разбит на пакеты (по 100 точек в каждом) с помощью классов PyTorch — `TensorDataset` и `DataLoader`:

```
train_dataset = torch.utils.data.TensorDataset(train_x, train_y)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=100)
```

Объект `train_loader` позволяет аккуратно перебирать получившиеся мини-пакеты при выполнении градиентного спуска. Потери (ELBO) вычисляются с использованием следующего синтаксиса:

```
output = model(x_batch)
loss = -mll(output, y_batch)
```

Здесь `x_batch` и `y_batch` — очередная партия (небольшое подмножество) от полного обучающего набора. В целом градиентный спуск реализуется следующим образом:

```
#Включаем режим обучения
model.train()
likelihood.train()

#Выполняем проход по всему набору обучающих данных 50 раз
for i in tqdm(range(50)):
    #На каждой итерации перебираем мини-пакеты в train_loader
    for x_batch, y_batch in train_loader:
        #Мини-пакетный градиентный спуск на партиях точек
        optimizer.zero_grad()

        output = model(x_batch)
        loss = -mll(output, y_batch)

        loss.backward()
        optimizer.step()

#Включаем режим прогнозирования
model.eval()
likelihood.eval()
```

Выполняя данный цикл, можно заметить, что он работает значительно быстрее, чем цикл с обычным ГП (на том же MacBook данный процесс занял менее одной секунды, что является существенным улучшением скорости).

Скорость VGP

Вы можете подумать, что сравнение 500 итераций градиентного спуска в обычном ГП и 50 итераций мини-пакетной версии в VGP не является справедливым. Но стоит помнить, что в каждой итерации внешнего цикла `for` для мини-пакетного градиентного спус-

ка мы также итерируем более 10 мини-пакетов в `train_loader`. Поэтому в сумме мы получаем 500 итераций градиента. Более того, даже если их выполнить в мини-пакетном спуске, это займет меньше 1 секунды, умноженной на 10, что все равно будет в 4 раза быстрее по сравнению с 45 секундами.

Таким образом, с помощью мини-пакетного градиентного спуска модель VGP может обучаться гораздо эффективнее. А что насчет качества обучения? Левая часть рис. 12.10 визуализирует прогрессирующую потерю ELBO во время алгоритма. По сравнению с рис. 12.5, несмотря на то, что потери не уменьшились последовательно на каждом шаге (имеется зигзагообразный тренд), они были эффективно минимизированы на протяжении всей процедуры.

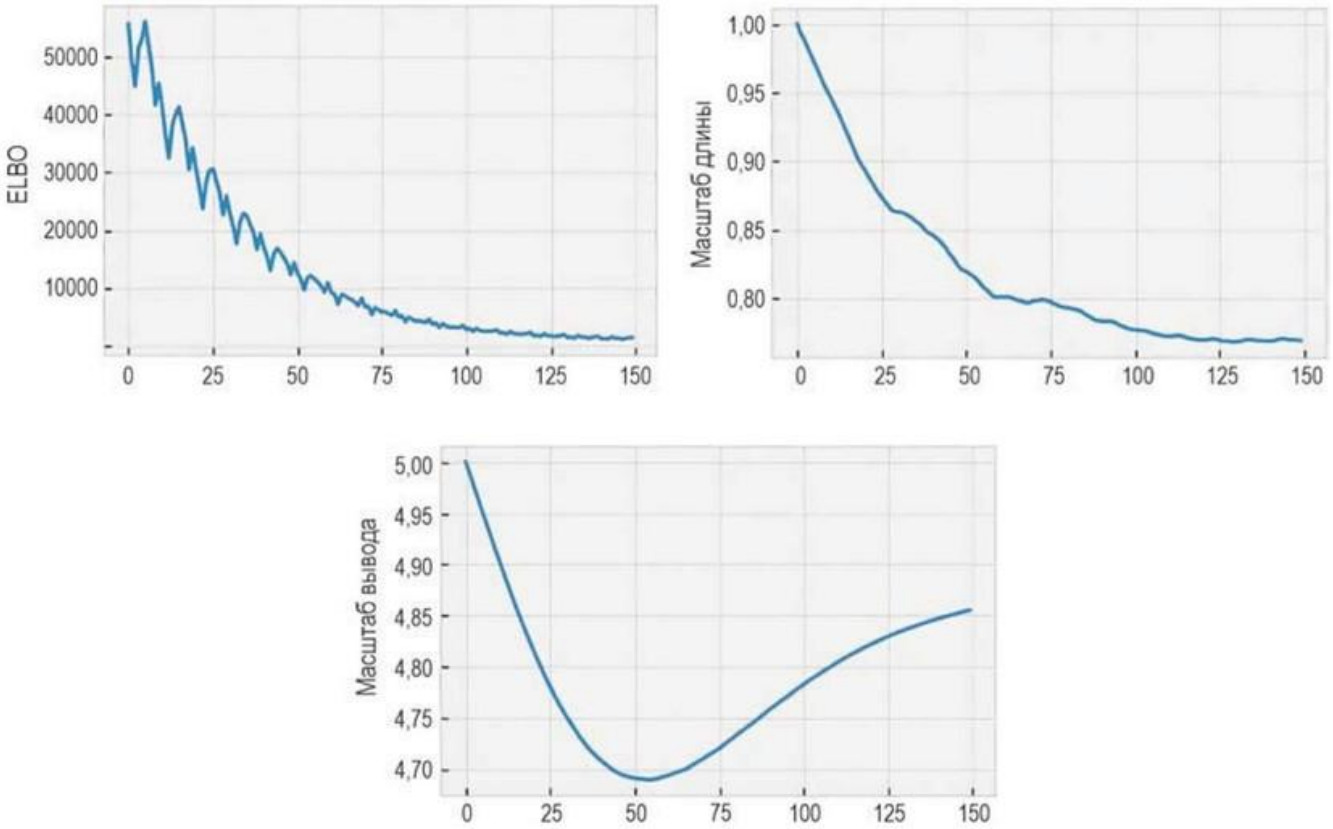


Рис. 12.10. Прогресс уменьшения потерь и соответствующие масштабы длины и вывода VGP во время мини-пакетного градиентного спуска

Это показывает, что каждый шаг при оптимизации может быть не лучшим направлением для минимизации потерь, но мини-пакетный градиентный спуск действительно эффективен в этом. Более наглядно это показано на рис. 12.11.

Далее визуализируем прогнозы модели VGP, чтобы увидеть, дает ли она разумные результаты. Используя вспомогательную функцию `visualize_gp_belief()`, получим рис. 12.12, на котором показано, что в результате у нас получилась высококачественная аппроксимация ГП, обученная на истинных потерях и за небольшие временные затраты.

Чтобы закончить обсуждение VPG, визуализируем расположение стимулирующих точек, изученных моделью. Уже говорилось, что стимулирующие точки должны быть репрезентативными для всего набора данных и хорошо отражать его тенден-

цию. Чтобы построить их, нужно получить доступ к их местоположению с помощью `model.variational_strategy.inducing_points.detach()` и построить их как разбросанные точки вдоль среднего прогноза. Во вспомогательной функции `visualize_gp_belief()` это уже реализовано, и единственное, что нужно сделать, это установить `variational = True` при вызове данной функции:

```
visualize_gp_belief(model, likelihood, variational=True)
```

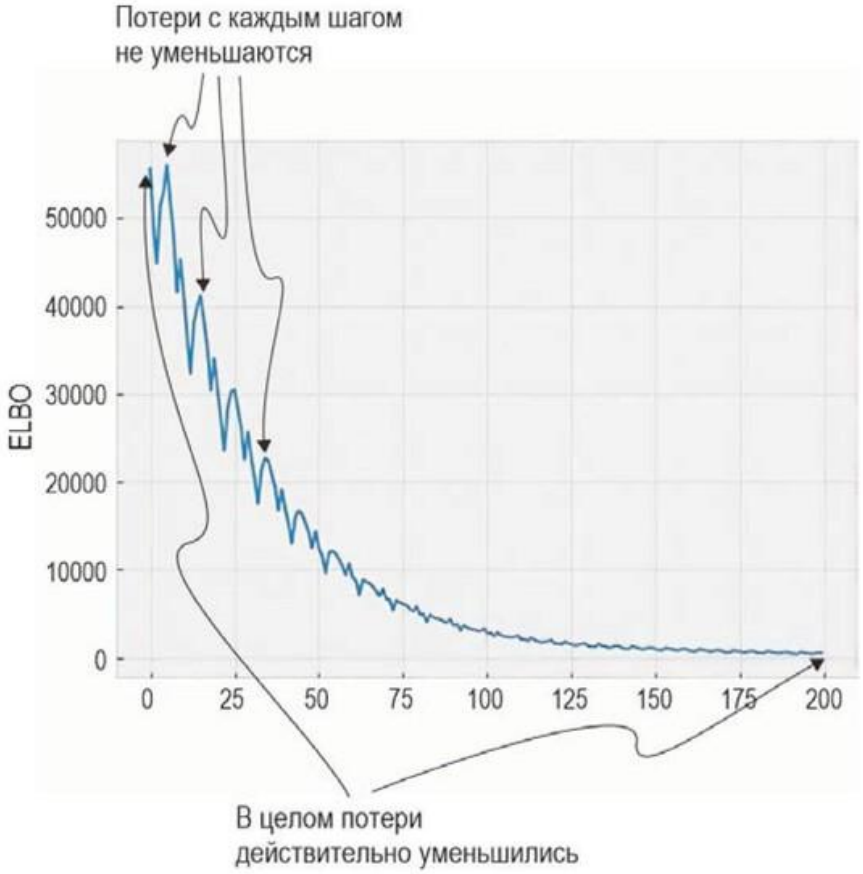


Рис. 12.11. Прогресс уменьшения потерь VGP во время мини-пакетного градиентного спуска: несмотря на некоторое непостоянство, потери эффективно минимизируются

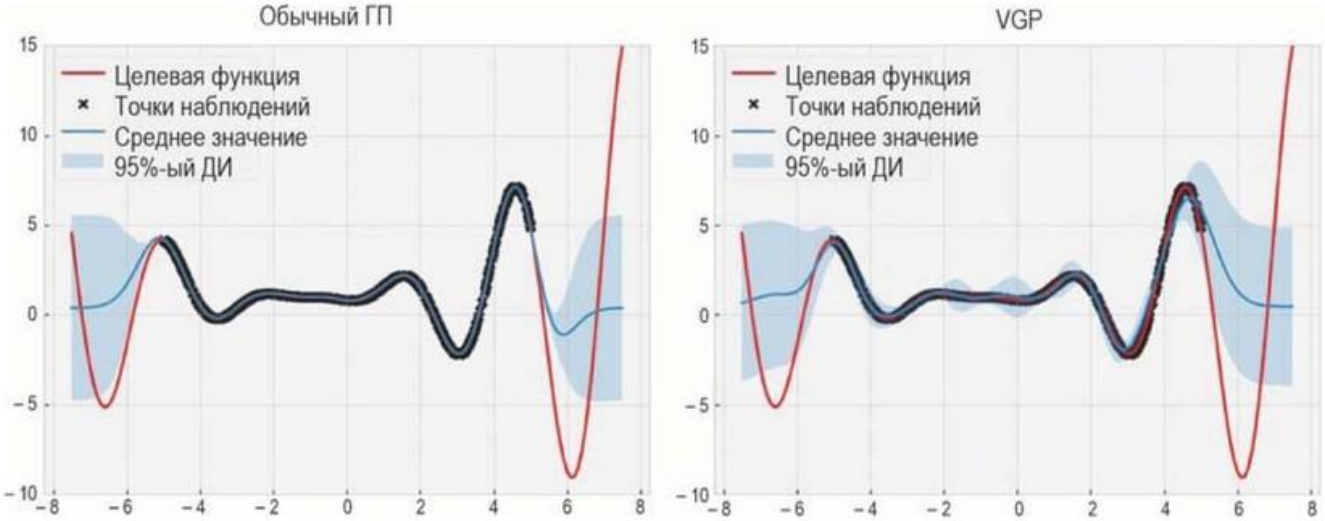


Рис. 12.12. Прогнозы обычного ГП и VGP примерно совпадают

В результате получается рис. 12.13, на котором видно очень интересное поведение стимулирующих точек. Они распределены по данным обучения неравномерно. Напротив, они группируются вокруг разных частей данных, где целевая функция изгибается вверх или вниз или демонстрирует какое-то нетривиальное поведение. Распределяя стимулирующие точки по этим местам, VGP может улавливать наиболее важные тенденции, которые имеет также и большой набор обучающих данных.

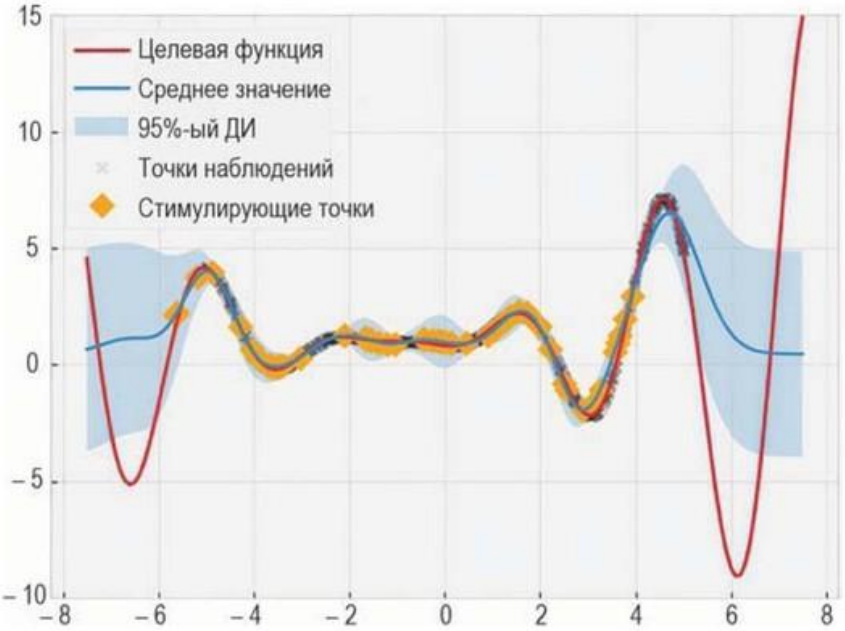


Рис. 12.13. Стимулирующие точки VGP расположены так, что представляют все данные и отражают наиболее важные тенденции

Теперь мы научились, как обучать VGP с помощью мини-пакетного градиентного спуска, и увидели, что это помогает аппроксимировать обычный ГП (обучение которого невозможно) с высокой точностью и гораздо меньшими затратами. В следующем разделе мы узнаем о другом алгоритме градиентного спуска, который может еще более эффективно обучить VGP.

12.3. Достижение лучшей оптимизации с учетом геометрии поверхности потерь

В этом разделе мы познакомимся с алгоритмом *естественного градиентного спуска* (**Natural Gradient Descent**, **NGD**). Он представляет собой еще одну версию обычного алгоритма, которая более тщательно учитывает геометрию функции потерь при вычислении каждого шага при спуске. Это позволит быстро снизить функцию потерь, что в конечном итоге приведет к более эффективной оптимизации с меньшим количеством итераций (т. е. к более быстрой сходимости).

Чтобы понять мотивацию естественного градиентного спуска и почему он работает лучше изученных алгоритмов, рассмотрим два типа параметров VGP:

- Стандартные параметры ГП, например средняя константа, масштаб длины или масштаб вывода для ковариационной функции. Они принимают обычные числовые значения, существующие в евклидовом пространстве.

- *Вариационные* параметры, которые есть только у VGP. Они связаны со стимулирующими точками и различными компонентами, необходимыми для облегчения аппроксимации вариационным распределением. Они связаны с распределениями вероятностей и имеют значения, которые не могут быть хорошо представлены в евклидовом пространстве.

> **Примечание**
>
> Разница между этими двумя типами параметров в чем-то похожа (хоть и не совсем аналогична) на то, как в евклидовом пространстве измеряется расстояние между двумя точками. Но при этом оно не позволяет измерить разницу между двумя распределениями вероятностей.

Несмотря на то что мини-пакетный градиентный спуск работает достаточно хорошо, он предполагает, что все параметры существуют в евклидовом пространстве. Например, с точки зрения этого алгоритма разница между масштабом длины 1 и масштабом длины 2 аналогична разнице между средним значением 1 и средним значением 2 для стимулирующей точки. Однако это не так: переход от масштаба длины для 1 к масштабу длины для 2 повлияет на модель VGP совсем иначе, нежели переход от среднего значения 1 к среднему значению 2. Это показано на рис. 12.14, где поведение потерь по отношению к масштабу длины совершенно иное, чем по отношению к стимулирующему среднему.



Рис. 12.14. Пример, как потери могут вести себя по-разному по отношению к обычному и вариационному параметрам: это порождает необходимость учета геометрии потерь

Такое различие в поведении существует потому, что геометрия функции потерь по отношению к обычным параметрам VGP принципиально отличается от геометрии функции потерь по отношению к вариационным параметрам. Если бы мини-пакетный градиентный спуск мог учитывать эту геометрическую разницу при вычислении направления спуска, он был бы более эффективным при минимизации потерь. Здесь на помощь приходит естественный градиентный спуск.

> **Определение**
>
> *Естественный градиентный спуск* (NGD) использует информацию о геометрии функции потерь по отношению к вариационным параметрам для расчета лучших направлений спуска.

Выбирая лучшие направления, NGD может помочь более эффективно и быстро оптимизировать модель VGP. Конечным результатом будет то, что мы придем

к окончательной модели за меньшее количество шагов. Продолжая двумерную иллюстрацию различных алгоритмов, на рис. 12.15 приведен пример, как учет геометрических различий помогает естественному градиентному спуску достичь цели быстрее, чем мини-пакетному варианту. Иначе говоря, ему нужно меньше шагов при обучении для достижения тех же потерь, что и мини-пакетному спуску. В текущей аналогии «спуска с горы» при NGD мы по-прежнему завязываем глаза тонкой тканью, но теперь у нас есть специальная походная обувь, которая помогает более эффективно перемещаться по местности.



Рис. 12.15. Демонстрация обычного, мини-пакетного и естественного градиентных спусков в «долине потерь», где центр долины дает наименьшие потери: учитывая геометрию функции потерь, естественный градиентный спуск достигает минимума потерь быстрее, чем мини-пакетный вариант

Дополнительный материал по NGD

Для более сложного математического объяснения естественного градиентного спуска я рекомендую превосходную публикацию в блоге «Natural Gradient Descent» Августина Кристиади.

Примечание

Важно отметить, что алгоритм NGD оптимизирует только вариационные параметры VGP. Обычные параметры, такие как масштаб длины и масштаб вывода, по-прежнему можно оптимизировать с помощью обычного мини-пакетного алгоритма. Мы увидим это, когда в следующий раз реализуем новую процедуру обучения.

Применим NGD для обучения модели VGP. Используя ту же одномерную целевую функцию, что и в предыдущих разделах, реализуем модель, которая будет работать с NGD. Класс модели в этом случае аналогичен `ApproximateGPModel`, который был реализован для мини-пакетного градиентного спуска в предыдущем разделе, поскольку он:

- все еще расширяет `gpytorch.models.ApproximateGP`;
- требует наличия вариационной стратегии для управления процессом обучения;
- имеет функцию среднего значения, функцию ковариации и метод `forward()`, как в обычной модели ГП.

Единственная разница здесь заключается в том, что вариационное распределение должно быть экземпляром `gpytorch.variational.NaturalVariationalDistribution`, чтобы можно было использовать NGD при обучении модели. Весь класс модели реализован следующим образом:

```
class NaturalGradientGPModel(gpytorch.models.ApproximateGP):
    def __init__(self, inducing_points):
        #Вариационное распределение должно быть естественным,
        #чтобы оно могло работать с естественным градиентным спуском
        variational_distribution =
          gpytorch.variational.NaturalVariationalDistribution(
            inducing_points.size(0)
        )

        #Объявление остальной части вариационной стратегии такое же,
        #как и раньше

        variational_strategy = gpytorch.variational.VariationalStrategy(
            self,
            inducing_points,
            variational_distribution,
            learn_inducing_locations=True,
        )
        super().__init__(variational_strategy)
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.RBFKernel()
        )

    def forward(self, x):
        ... #Метод forward() такой же, как и раньше
```

Еще раз инициализируем модель VGP с 50 стимулирующими точками:

```
model = NaturalGradientGPModel(train_x[:50, :]) #50 стимулирующих точек
likelihood = gpytorch.likelihoods.GaussianLikelihood()
```

Далее следует важная часть, в которой мы объявим оптимизаторы для обучения. Следует помнить, что мы используем алгоритм естественного градиентного спуска (NGD) для оптимизации вариационных параметров модели. Однако другие параметры, такие как масштаб длины и масштаб вывода, по-прежнему должны быть оптимизированы с помощью алгоритма Adam. Таким образом, код будет следующим:

```
#Естественный градиентный спуск принимает вариационные параметры VGP —
# — model.variational_parameters().
ngd_optimizer = gpytorch.optim.NGD(
    model.variational_parameters(), num_data=train_y.size(0), lr=0.1
)

#Adam принимает другие параметры VGP:
# model.parameters() и likelihood.parameters()
hyperparam_optimizer = torch.optim.Adam(
    [{"params": model.parameters()}, {"params":likelihood.parameters()}],
    lr=0.01
)
```

```
mll = gpytorch.mlls.VariationalELBO(
    likelihood, model, num_data=train_y.size(0)
)
```

Теперь во время обучения мы по-прежнему можем вычислять потери, используя:

```
output = model(x_batch)
loss = -mll(output, y_batch)
```

Для этого мы просматриваем мини-пакеты (`x_batch` и `y_batch`) обучающих данных. Однако теперь одновременно работают два оптимизатора. Ими нужно управлять, вызывая для каждого из них и на каждой итерации обучения методы `zero_grad()`, чтобы очистить градиенты с предыдущего шага, и `step()`, чтобы сделать шаг спуска:

```
#Включаем обучающий режим
model.train()
likelihood.train()

for i in tqdm(range(50)):
    for x_batch, y_batch in train_loader:
        #Удаляем градиенты с предыдущего шага
        ngd_optimizer.zero_grad()
        hyperparam_optimizer.zero_grad()

        output = model(x_batch)
        loss = -mll(output, y_batch)

        loss.backward()

        #Делаем шаг спуска с каждым оптимизатором
        ngd_optimizer.step()
        hyperparam_optimizer.step()

#Включаем режим прогнозирования
model.eval()
likelihood.eval()
```

> **Примечание**
>
> Как и всегда, нам нужно вызвать `model.train()` и `likelihood.train()` перед градиентным спуском, а также `model.eval()` и `likelihood.eval()` после завершения обучения.

Обратите внимание, что мы вызываем `zero_grad()` и `step()` как для оптимизатора естественного градиентного спуска, так и для Adam, чтобы оптимизировать соответствующие параметры модели VGP. Цикл обучения, опять же, занимает очень короткое время. Обученный VGP выдает прогнозы, показанные на рис. 12.16. На нем видны очень похожие прогнозы, как у обычного ГП на рис. 12.4, так и у VGP, обученного с помощью мини-пакетного градиентного спуска, на рис. 12.13.

Во время обучения мы также можем дополнительно проверить прогрессирующую потерю ELBO. Это показано на левой части рис. 12.17.

Примечательно, что потери ELBO упали до низкого значения почти сразу во время обучения. Это доказывает, что NGD смог помочь быстро прийти к хорошей модели. Рисунок иллюстрирует преимущества градиентного спуска при обучении VGP.



Рис. 12.16. Прогнозы VGP и стимулирующие точки, обученные с помощью естественного градиентного спуска: качество прогнозов превосходное

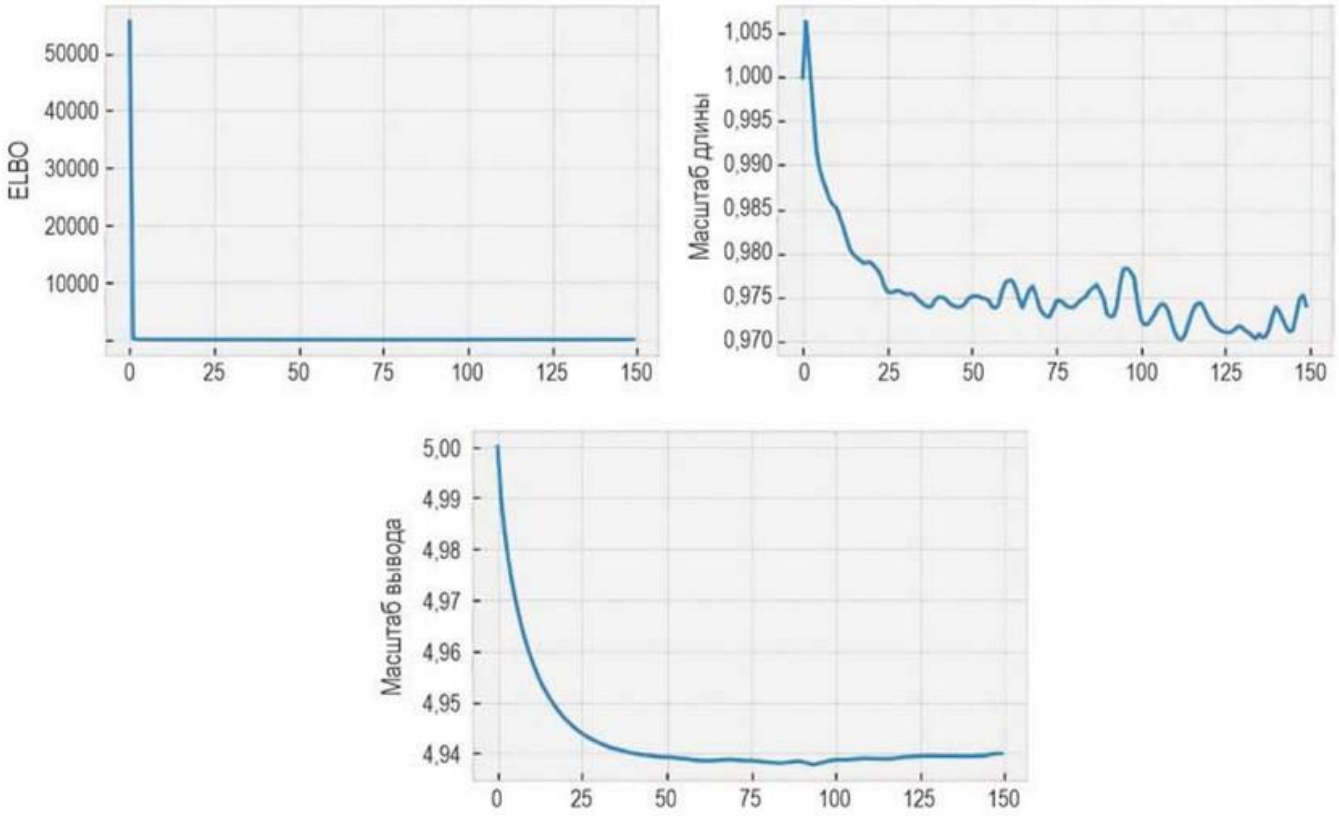


Рис. 12.17. Прогресс потерь VGP во время естественного градиентного спуска: потери были эффективно минимизированы после нескольких итераций

Мы подошли к концу главы 12. В ней мы научились масштабировать модели ГП до больших наборов данных с помощью стимулирующих точек, которые представляют собой набор репрезентативных точек. Их цель — уловить тенденции, демонстрируемые большим обучающим набором. Полученная модель называется VGP. Она допускает мини-пакетный градиентный спуск и, следовательно, может обучаться без расчета потерь модели по всем точкам данных. Мы также рассмотрели естест-

венный градиентный спуск как более эффективную версию мини-пакетного алгоритма, позволяющую оптимизировать за меньшее количество шагов. В главе 13 мы рассмотрим еще одно расширенное использование гауссовских процессов через объединение их с нейросетями для моделирования сложных структурированных данных.

12.4. Упражнение

Это упражнение демонстрирует повышение эффективности при переходе от обычной модели ГП к модели VGP на реальном наборе данных, где представлены цены на жилье в Калифорнии. Цель — оценить вычислительные преимущества VGP, на этот раз в реальных условиях.

Выполните следующие шаги:

1. Используйте функцию `read_csv()` из библиотеки Pandas для чтения набора, который хранится в электронной таблице с именем `data/housing.csv`. Он получен из данных о ценах на жилье в Калифорнии на платформе Kaggle под лицензией Creative Common Public Domain. После считывания фрейм данных Pandas должен выглядеть примерно так, как на рис. 12.18.

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
0	-117.08	32.70	37	2176	418.0	1301	375	2.8750	98900
1	-117.91	34.11	20	3158	684.0	2396	713	3.5250	153000
2	-117.10	32.75	11	2393	726.0	1905	711	1.3448	91300
3	-117.22	32.74	52	1260	202.0	555	209	7.2758	345200
4	-121.99	37.29	32	2930	481.0	1336	481	6.4631	344100
...	...	...	...	...	...	...	...	...	...
4995	-118.41	34.19	42	779	145.0	450	148	3.9792	193800
4996	-122.96	38.42	50	2530	524.0	940	361	2.9375	122900
4997	-118.00	33.77	24	1324	267.0	687	264	3.4327	192800
4998	-122.81	38.54	12	2289	611.0	919	540	1.1553	139300
4999	-118.08	33.84	25	3696	953.0	2827	860	3.3438	153300

Рис. 12.18. Набор данных о ценах на жилье, показанный в виде фрейма данных Pandas: это тренировочный набор для данного упражнения

2. Визуализируйте столбец `median_house_value`, который является целью прогнозирования, на диаграмме. Оси X и Y на ней соответствуют столбцам долготы и широты, расположение точки соответствует местоположению дома, а цвет точки соответствует цене. Визуализация должна выглядеть примерно так, как на рис. 12.19.
3. Извлеките все столбцы, кроме последнего (`median_house_value`), и сохраните их как тензор PyTorch. Он будет использоваться в качестве функций обучения `train_x`.
4. Извлеките столбец `median_house_value` и сохраните его логарифмическое преобразование как еще один тензор PyTorch. Это и есть цель обучения (`train_y`).
5. Нормализуйте обучающие метки `train_y`, вычтя среднее значение и разделив их на стандартное отклонение. Это сделает обучение более стабильным.

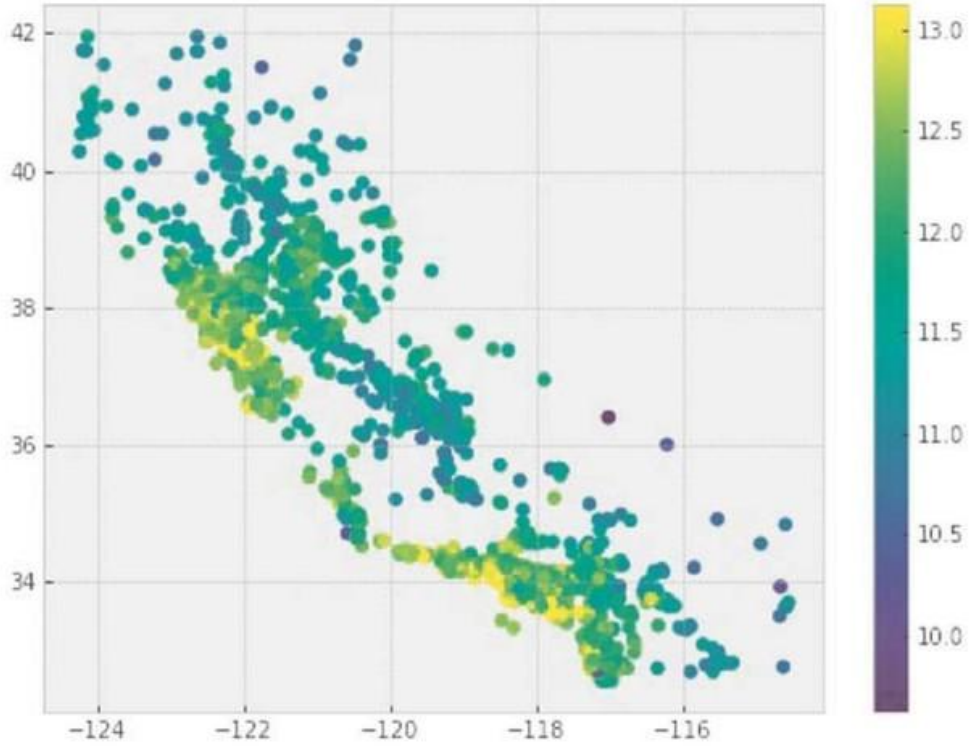


Рис. 12.19. Набор данных о ценах на жилье показан в виде рассеяния точек

6. Реализуйте обычную модель ГП, используя постоянную функцию среднего и ядро Матерна 5/2 с автоматическим определением релевантности (ARD) и масштабом вывода. Дополнительную информацию о ядрах Матерна и ARD можно найти в разделах 3.4.2 и 3.4.3 соответственно.
7. Создайте вероятность, шум которой как минимум ограничен значением 0,1, используя следующий код:

```
likelihood = gpytorch.likelihoods.GaussianLikelihood(
    #Ограничение делает значение шума не меньше 0,1
    noise_constraint=gpytorch.constraints.GreaterThan(1e-1)
)
```

   Это поможет сгладить обучающие метки за счет повышения устойчивости к шуму.
8. Инициализируйте ранее реализованную модель ГП и обучите ее с помощью вероятности, используя градиентный спуск на 10 итераций. Учтите общее время обучения.
9. Реализуйте VGP с теми же средними значениями и ковариационными функциями, что и обычный ГП. Данная модель похожа на класс `ApproximateGPModel`, который был реализован ранее в этой главе, за исключением того, что теперь нам требуется ядро Матерна 5/2 с ARD.
10. Обучите VGP с помощью аналогично инициализированной вероятности и 100 стимулирующих точек, используя естественный градиентный спуск (NGD) для 10 итераций. Для мини-пакетного градиентного спуска вы можете разделить обучающий набор размером 100 точек на пакеты.

11. Убедитесь, что обучение VGP занимает меньше времени, чем обучение обычного ГП. Что касается функций синхронизации, вы можете использовать `time.time()` для записи времени начала и окончания обучения каждой модели. Или вы можете использовать библиотеку `tqdm` для отслеживания продолжительности обучения.

Решение включено в `CH11/02 - Exercise.ipynb`.

Заключение

- Вычислительная стоимость ГП зависит от размера набора обучающих данных. Обучение модели становится непомерно трудным по мере роста размера набора с данными.
- Вычисление потерь ML-модели по большому количеству точек численно нестабильно. Потери, вычисленные этим способом, могут ввести в заблуждение при оптимизации во время градиентного спуска, что приведет к плохой производительности прогнозирования.
- VGP масштабируется до большого набора данных, обучаясь только на небольшом подмножестве стимулирующих точек. Они должны быть репрезентативными для всего набора, чтобы обученная модель могла быть максимально похожа на ГП, обученный на полных данных.
- Чтобы создать приближенную модель, которая максимально близка к обычному ГП, обученному на всем наборе, при формулировке VGP используется расхождение Кульбака — Лейблера, которое измеряет разницу между двумя распределениями вероятностей.
- Нижняя граница доказательств (ELBO) действует как показатель истинной потери при обучении VGP. Точнее, ELBO устанавливает нижнюю границу предельной логарифмической вероятности модели, которую мы и стремимся оптимизировать. Оптимизируя ELBO, мы косвенно оптимизируем предельную логарифмическую вероятность.
- Обучение VGP может выполняться небольшими партиями, что позволяет более стабильно рассчитывать потери. Алгоритм градиентного спуска, используемый в этой процедуре, представляет собой мини-пакетный градиентный спуск.
- Несмотря на то что каждый шаг мини-пакетного градиентного спуска не гарантирует полной минимизации потерь, данный алгоритм при выполнении большого количества итераций может эффективно уменьшить потери. Это связано с тем, что многие шаги мини-пакетного спуска в совокупности могут указывать в правильном направлении, чтобы минимизировать потери.
- Естественный градиентный спуск учитывает геометрию функции потерь относительно вариационных параметров VGP. Это позволяет алгоритму обновлять вариационные параметры обученной модели и более эффективно минимизировать потери, что приводит к более быстрой сходимости.
- Естественный градиентный спуск оптимизирует вариационные параметры VGP. Обычные параметры, такие как масштаб длины и масштаб вывода, оптимизируются с помощью мини-пакетного градиентного спуска.

13 Комбинирование гауссовских процессов с нейросетями

Темы главы:

- Сложность обработки комплексных структурированных данных с помощью общих ковариационных функций.
- Использование нейросетей для обработки комплексных структурированных данных.
- Объединение нейросетей и гауссовских процессов.

Как известно из главы 2, функции среднего значения и ковариации выступают в качестве априорной информации, которую важно включать в модель при составлении прогнозов. Поэтому их выбор сильно влияет на поведение обученного ГП. Следовательно, если функции среднего и ковариации указаны неправильно или не подходят для поставленной задачи, полученные прогнозы будут бесполезны.

В качестве примера можно вспомнить, что ковариационная функция (ядро) выражает корреляцию (сходство) между двумя точками. Чем больше две точки похожи, тем больше вероятность, что они будут иметь одинаковые значения для меток, которые мы стремимся предсказать. В примере прогнозирования цен на жилье похожие дома, скорее всего, будут продаваться по схожим ценам.

Как ядро точно вычисляет сходство между любыми двумя заданными домами? Рассмотрим два случая. В первом ядро будет учитывать только цвет входной двери, и соответственно выдавать значение 1 для любых двух домов с дверью одинакового цвета и 0 в противном случае. Другими словами, ядро будет считать, что два дома похожи тогда и только тогда, когда входные двери в них одного цвета.

Данное ядро показано на рис. 13.1. Оно является плохим выбором для модели прогнозирования цен, поскольку ошибочно определяет, что у домов слева и по середине должна быть одинаковая стоимость, а у домов справа и по середине — разная. Такое в реальности невозможно, поскольку левый дом намного больше двух других, схожих по размеру. Неправильный прогноз происходит потому, что ядро неверно определяет, какая характеристика дома является правильным предсказательным признаком его стоимости.



Рис. 13.1. Ковариация между домами, вычисленная с помощью неподходящего ядра: поскольку ядро учитывает только цвет входной двери, оно не создает необходимых соответствий между домами

В качестве другого примера рассмотрим более сложное ядро, которое учитывает соответствующие факторы, такие как местоположение и жилая площадь. Оно будет лучше показывать ковариацию, поскольку может более обоснованно описать сходство цен на два дома. Наличие правильного ядра (т. е. правильной меры сходства) имеет первостепенное значение для ГП. Если оно может правильно описать, насколько похожи или различаются две определенные точки данных, ГП сможет давать хорошо калиброванные прогнозы. В противном случае они будут низкого качества.

Можно подумать, что первое ядро для примера с домами использовать неуместно, и ни одна более-менее разумная функция ковариации в ML не будет вести себя подобным образом. Однако в этой главе мы продемонстрируем, что некоторые распространенные алгоритмы (которые мы уже использовали, например RBF или ядро Матерна), одинаково проваливаются при обработке таких структурированных входных данных, как изображения. Если более конкретно, то они не могут адекватно описать сходство между двумя картинками, что создает проблему для обучения ГП работе с подобными структурированными типами данных. Более подходящий способ решить задачу заключается в использовании нейросетей. Это гибкие модели, которые могут хорошо аппроксимировать любую функцию при наличии достаточного количества информации. В этой главе мы научимся использовать нейросеть для преобразования входных данных, которые ядро ГП не может правильно обрабатывать. При этом мы получим лучшее от обоих вариантов: гибкое моделирование с помощью нейросети и прогнозы ГП с калибровкой по неопределенности.

Мы продемонстрируем, что обычное ядро RBF плохо отражает структуру общего набора данных. Это, в свою очередь, приводит к неверным прогнозам. Затем мы попробуем объединить модель нейросети и получившийся ГП и увидим, что новое ядро может успешно делать предположения о сходстве. К концу главы мы получим структуру, которая будет помогать ГП обрабатывать структурированные типы данных и повысит производительность прогнозирования.

13.1. Данные, имеющие структуру

В этом разделе мы объясним, что именно подразумевается под структурированными данными. В отличие тех, которые использовались для обучения ГП в предыдущих главах, где каждый признак (столбец) в наборе может принимать значения

в непрерывном диапазоне, во многих приложениях информация имеет более сложную структуру:

- Количество этажей в доме может быть только положительным целым числом.
- В задачах компьютерного зрения значение пикселя на изображении представляет собой целое число от 0 до 255.
- Молекулы в ML часто представлены в виде графа.

В этих примерах есть *структуры*, встроенные в точки данных, или *требования*, которым точки данных должны соответствовать. Ни один дом не может иметь отрицательное число этажей, пиксель не может принимать в качестве значения дробь, а граф, обозначающий молекулу, будет иметь узлы и ребра, представляющие химические вещества и связи. Такие данные называются *структурированными*. На протяжении всей главы мы будем использовать набор рукописных цифр MNIST в качестве наглядного примера.

Определение

Набор данных от *Модифицированного национального института стандартов и технологий* (**The Modified National Institute of Standards and Technology**, **MNIST**) содержит изображения рукописных цифр, каждое из которых представляет собой матрицу целых чисел от 0 до 255 размером 28×28.

Пример точки данных из набора показан на рис. 13.2. Пиксель показан с оттенком, соответствующим его значению: 0 соответствует белому пикселю, а 255 соответст-

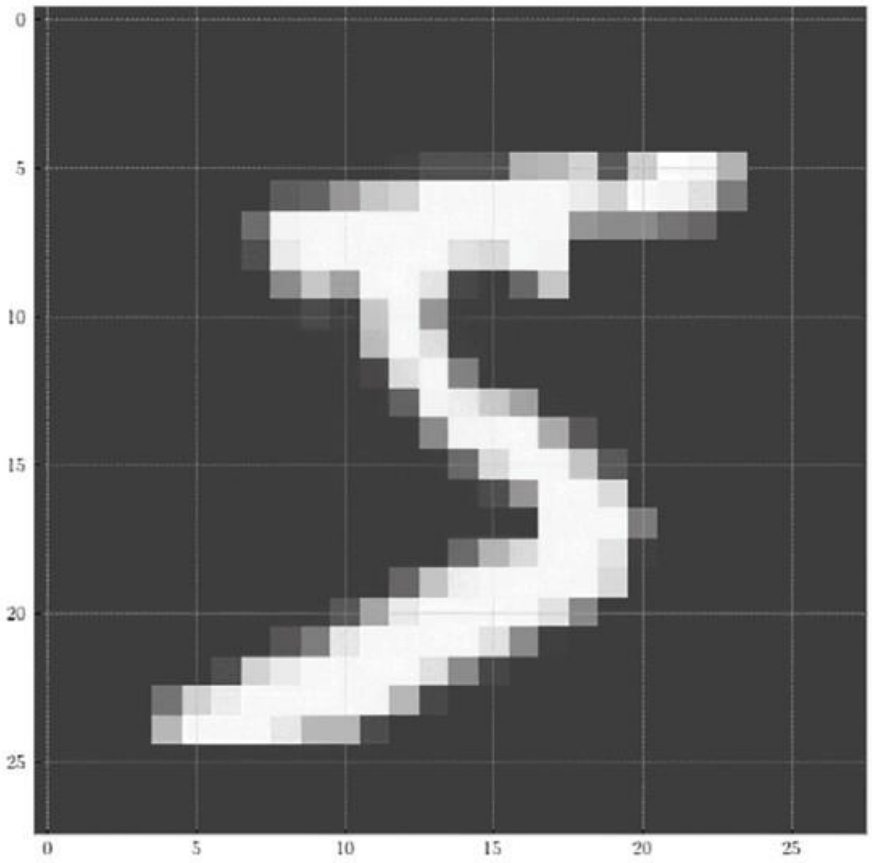


Рис. 13.2. Точка из набора данных MNIST: она выглядит как изображение размером 28×28 и представлена в виде тензора PyTorch

вует темному пикселю. Отчетливо видно, что эта точка данных представляет собой изображение цифры пять.

> **Примечание**
>
> Несмотря на то что распознавание рукописных цифр технически является задачей классификации, она используется для моделирования регрессии (тип задачи, которую мы стремимся решить в БО). Поскольку каждая метка представляет собой число (цифру), мы представляем, что они существуют в непрерывном диапазоне, и напрямую используем их в качестве цели прогнозирования.

Наша цель — обучить ГП на наборе данных меток изображений (каждая метка здесь представляет собой значение цифры, записанной в соответствующем изображении), а затем использовать его для прогнозирования на тестовом наборе. Различие показано на рис. 13.3. В отличие от классификации, где мы выбираем один из классов в качестве прогноза для каждой точки данных (выбираем подходящую «коробочку»), прогнозы здесь представляют собой число внутри непрерывного диапазона в задаче регрессии.

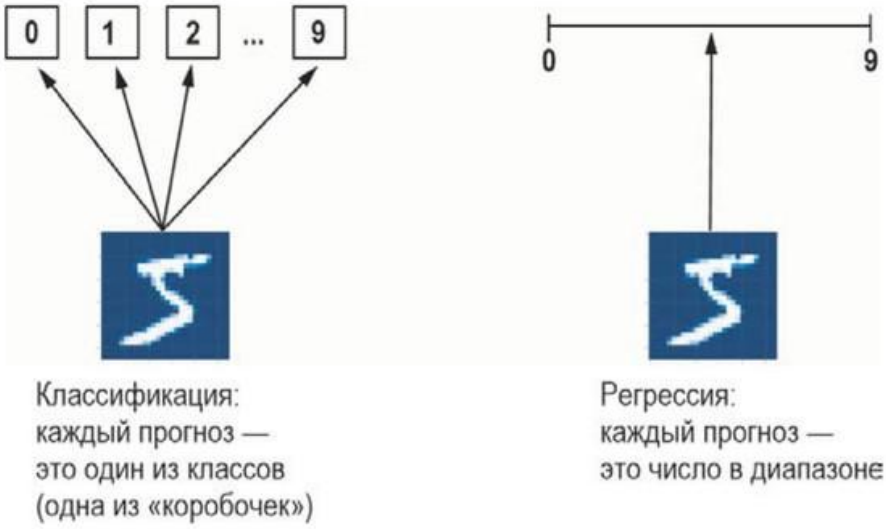


Рис. 13.3. Классификация и регрессия в контексте данных MNIST: каждый прогноз в классификации соответствует одному из классов, каждый прогноз в регрессии представляет собой число внутри непрерывного диапазона

Существует множество реальных приложений, которые используют данную форму данных (диапазон):

- В рекомендациях по продуктам: предсказать вероятность, что кто-то нажмет на настроенную рекламу. Объявления имеют вид изображений, которые можно настроить, и представляют собой структурированные данные. Вероятность клика — цель прогнозирования. Она может быть любым числом от 0 до 1.
- В материаловедении: предсказать уровень энергии молекулярной композиции, когда она синтезируется в лаборатории. Каждую молекулярную композицию можно представить в виде графа конкретной структуры с узлами и ребрами. Уровень энергии может быть любым числом между возможными минимальным и максимальным уровнями энергии, которые может демонстрировать композиция.

- При открытии лекарств: предсказать эффективность потенциального препарата. Как показано на рис. 13.4, каждому веществу соответствует химическое соединение, которое также можно представить в виде графа. Его эффективность может быть реальным числом на шкале (например, от 0 до 10).

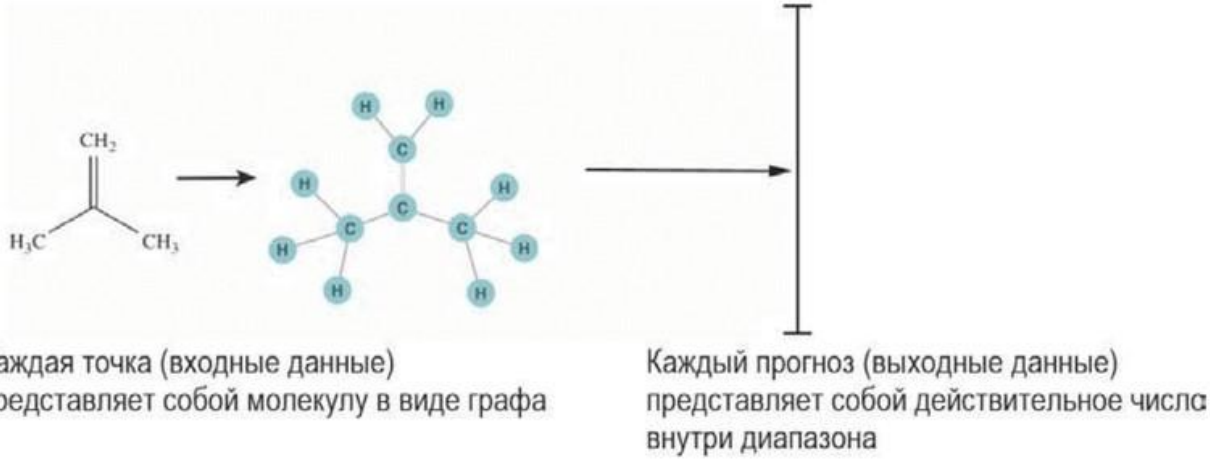


Рис. 13.4. Открытие лекарств как пример задачи структурированной регрессии: каждое соединение представлено в виде графа, и мы стремимся предсказать эффективность соединения при лечении какого-либо заболевания по шкале от 0 до 10

Во всех подобных приложениях входные данные, на основе которых нужно выполнить прогноз, являются структурированными, а цель прогнозирования — действительное число. Иначе говоря, это регрессионные задачи со структурированными данными. Используя набор данных MNIST, смоделируем одну такую задачу.

13.2. Нахождение сходства в структурированных данных

В этом разделе мы исследуем, как распространенные ядра, вроде RBF (раздел 2.4), не способны описать сходства в структурированных данных. Вывод для любых двух входных данных x_1 и x_2, который количественно определяет ковариацию между ними, определяются следующим образом:

$$K(x_1, x_2) = \exp\left(-\frac{\|x_1 - x_2\|^2}{2l^2}\right).$$

Данный вывод представляет собой экспоненту, которая взята со знаком «минус» от разницы между двумя входными данными, разделенной на шкалу. Выходные данные всегда находятся в диапазоне от 0 до 1, и чем больше разница в значениях, тем меньше выходные данные.

Во многих случаях это имеет смысл, поскольку, если два входа имеют одинаковые значения и, следовательно, небольшую разницу, их ковариация будет высокой. И наоборот, если они имеют разные значения, ковариация будет низкой. Два дома с примерно одинаковой жилой площадью, скорее всего, будут иметь одинаковые цены (они будут иметь высокую ковариацию). С другой стороны, цены очень большого и очень маленького дома, скорее всего, будут иметь низкую ковариацию.

13.2.1. Использование ядра с GPyTorch

Проверим это с помощью кода. Обычно мы инициализируем объект `RBFKernel` при создании модели. Здесь мы будем работать напрямую с объектом ядра. Для этого сначала создадим объект RBF с помощью GPyTorch:

```
import gpytorch
rbf_kernel = gpytorch.kernels.RBFKernel()
```

> **Примечание**
>
> Чтобы вспомнить, как объект ядра используется ГП в GPyTorch, можно обратиться к разделу 2.4.

Чтобы вычислить ковариацию между двумя входными данными, можно просто передать их объекту ядра. Например, попробуем вычислить ковариацию между 0 и 0,1:

```
>>> rbf_kernel(torch.tensor([0.]), torch.tensor([0.1])).evaluate().item() 0.9896470904350281
```

Эти два числа находятся близко на числовой прямой (иначе говоря, они похожи), поэтому их ковариация очень высока — почти 1. Далее, вычислим ковариацию между двумя разными числами, например, 0 и 10:

```
>>> rbf_kernel(torch.tensor([0.]), torch.tensor([10.])).evaluate().item()
0.0
```

В этом случае разница между двумя числами намного больше, следовательно, их ковариация падает до 0. Данный контраст показан на рис. 13.5.



Рис. 13.5. Ковариация между различными числами: когда разница между двумя числами мала, ковариация увеличивается; когда разница велика, ковариация уменьшается

Проблема возникает, когда разница в значениях между двумя входными данными не отражает структурную разницу в значении данных. Это часто относится к структурированным данным, таким как изображения.

13.2.2. Работа с изображениями в PyTorch

В этом разделе мы увидим, как импортировать и сохранять изображения в виде тензоров PyTorch, а также как метрики сходства на основе значений вроде RBF ломаются при обработке такого рода данных. Сначала переопределим RBF с масштабом большой длины, чтобы более вероятными были высокие ковариации:

```
#Масштаб большой длины приводит к более высоким ковариациям
rbf_kernel = gpytorch.kernels.RBFKernel()
rbf_kernel.lengthscale = 100
```

Далее нужно импортировать изображения из набора данных MNIST в код Python. Это можно сделать, используя PyTorch и его библиотеку torchvision:

```
import torch
from torchvision import datasets, transforms

#Определяем преобразование для нормализации значений пикселей
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

#Загружаем и импортируем набор данных
dataset = datasets.MNIST(
    "../data", train=True, download=True, transform=transform
)

#Извлекает значения пикселей в виде «сплюснутого» тензора
train_x = dataset.data.view(-1, 28 * 28)
```

Мы не будем углубляться в данный код, поскольку он не является предметом нашего обсуждения. Нам достаточно знать, что `train_x` содержит изображения из набора MNIST. Каждое из них хранится в виде тензора PyTorch, который содержит значения пикселей, сформированных в изображение рукописной цифры.

Поскольку точки данных представляют собой изображения, можно визуализировать их в виде тепловых карт, используя знакомую функцию `imshow()` из Matplotlib. Например, следующий код визуализирует первую точку данных в `train_x`:

```
plt.figure(figsize=(8, 8))

#Каждое изображение имеет 28 строк и 28 столбцов пикселей,
#поэтому нет нужды преобразовать его в квадратный тензор 28×28
plt.imshow(train_x[0, :].view(28, 28));
```

Этот код создаст изображение (рис. 13.2), которое представляет собой начертание цифры 5. При выводе фактических значений первой точки данных видно, что она является тензором PyTorch размером 28×28 = 784 элемента:

```
>>> train_x[0, :]
tensor([ 0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
         0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
```

```
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,  18,
         18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170, 253,
        253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253, 253,
        253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,  18, 219, 253, 253, 253, 253, 253,
        198, 182, 247, 241,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,  80, 156, 107, 253, 253, 205,
         11,   0,  43, 154,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,  14,   1, 154, 253,  90,
[output truncated]
```

Каждый элемент тензора в диапазоне от 0 до 255 представляет собой пиксель, который показан на рис. 13.2. Значение 0 соответствует самому низкому сигналу (фону на рисунке), а более высокие значения соответствуют ярким пятнам.

13.2.3. Вычисление ковариации двух изображений

Вся базовая информация, нужная для изучения задачи, с которой сталкиваются обычные ядра ГП при обработке структурированных данных, у нас уже есть. Чтобы подчеркнуть задачу, выделим три конкретные точки данных, которые назовем A, B и C, и будем индексировать их следующими числами:

```
ind1 = 304 #Точка A
ind2 = 786 #Точка B
ind3 = 4   #Точка C
```

Прежде чем проверять реальные цифры, которые эти изображения показывают, сначала вычислим их ковариационную матрицу, используя RBF:

```
>>> rbf_kernel(train_x[[ind1, ind2, ind3], :]).evaluate()
tensor([[1.0000e+00, 4.9937e-25, 0.0000e+00],
        [4.9937e-25, 1.0000e+00, 0.0000e+00],
        [0.0000e+00, 0.0000e+00, 1.0000e+00]], ...)
```

Перед нами ковариационная матрица 3×3 со знакомой структурой: диагональные элементы принимают значение 1, представляя дисперсии отдельных переменных, в то время как недиагональные элементы представляют различные ковариации. Также видно, что точки A и C совершенно не коррелируют и имеют нулевую ковариацию, а точки A и B коррелируют слабо. Согласно ядру RBF точки A и B похожи друг на друга и полностью отличаются от точки C.

Ожидается, что точки A и B будут иметь одну и ту же метку. Однако это не так! Визуализируем еще раз эти точки данных в виде тепловых карт и получим рис. 13.6.

Здесь точки A и C имеют одну и ту же метку (цифра 9). Так почему же ядро RBF считает, что коррелируют точки A и B? Глядя на рис. 13.6, можно предположить, что хотя точки A и B имеют разные метки, сами изображения похожи, поскольку

многие пиксели совпадают. Фактически штрихи, образующие хвосты цифр, почти точно совпадают на этих изображениях. В каком-то смысле RBF выполняет свою работу, вычисляя разницу между изображениями и выдавая число, представляющее их ковариацию на основе этой разницы. Однако вычисляется она путем сравнения пикселей самих по себе, что не является показателем, указывающим на то, что мы пытаемся изучить: значения цифр.

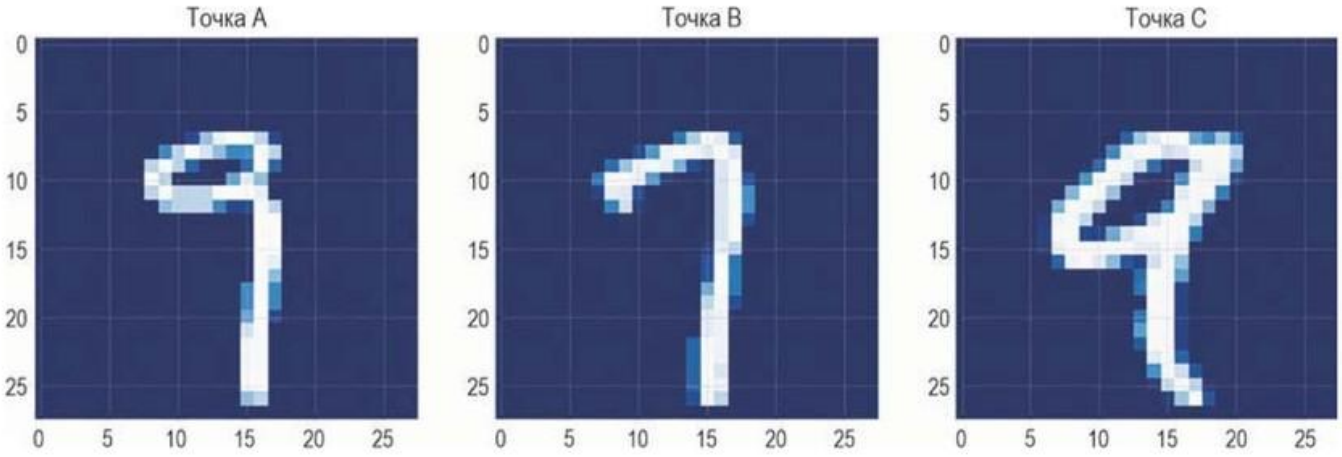


Рис. 13.6. Три конкретных значения из набора данных MNIST: первая и вторая точки имеют ненулевую ковариацию, несмотря на разные метки; первая и третья точки имеют нулевую ковариацию, несмотря на то, что имеют одинаковую метку

Рассматривая только значения пикселей, ядро RBF переоценивает ковариацию между точками A и B, имеющими разные метки, и недооценивает ковариацию между точками A и C, имеющими одну и ту же метку, как показано на рис. 13.7. Здесь можно использовать аналогию, чтобы продемонстрировать неподходящее ядро, о котором упоминалось в начале главы. Оно учитывает только цвет входной двери, чтобы решить, коррелируют ли два дома. Это приводит к неточным предсказаниям их цен. Аналогично (но не настолько слепо) в текущем примере ядро RBF учитывает только значения пикселей, а не шаблоны более высокого уровня при сравнении двух изображений. Это приводит к снижению производительности прогнозирования.



Рис. 13.7. Ковариация между рукописными цифрами, вычисленная ядром RBF: поскольку ядро рассматривает только значения пикселей, оно не делает соответствующих ковариаций

13.2.4. Обучение ГП на изображениях

Используя неправильную метрику сходства, RBF путает, какая из точек B и C коррелирует с точкой A, что приводит к плохим результатам при обучении ГП. Мы еще раз используем набор MNIST, на этот раз извлекая 1000 точек данных для

формирования обучающего набора и еще 500 точек в качестве тестового набора. Рабочий процесс подготовки и обучения данных обобщен на рис. 13.8, отдельные этапы которого мы более подробно рассмотрим позже.

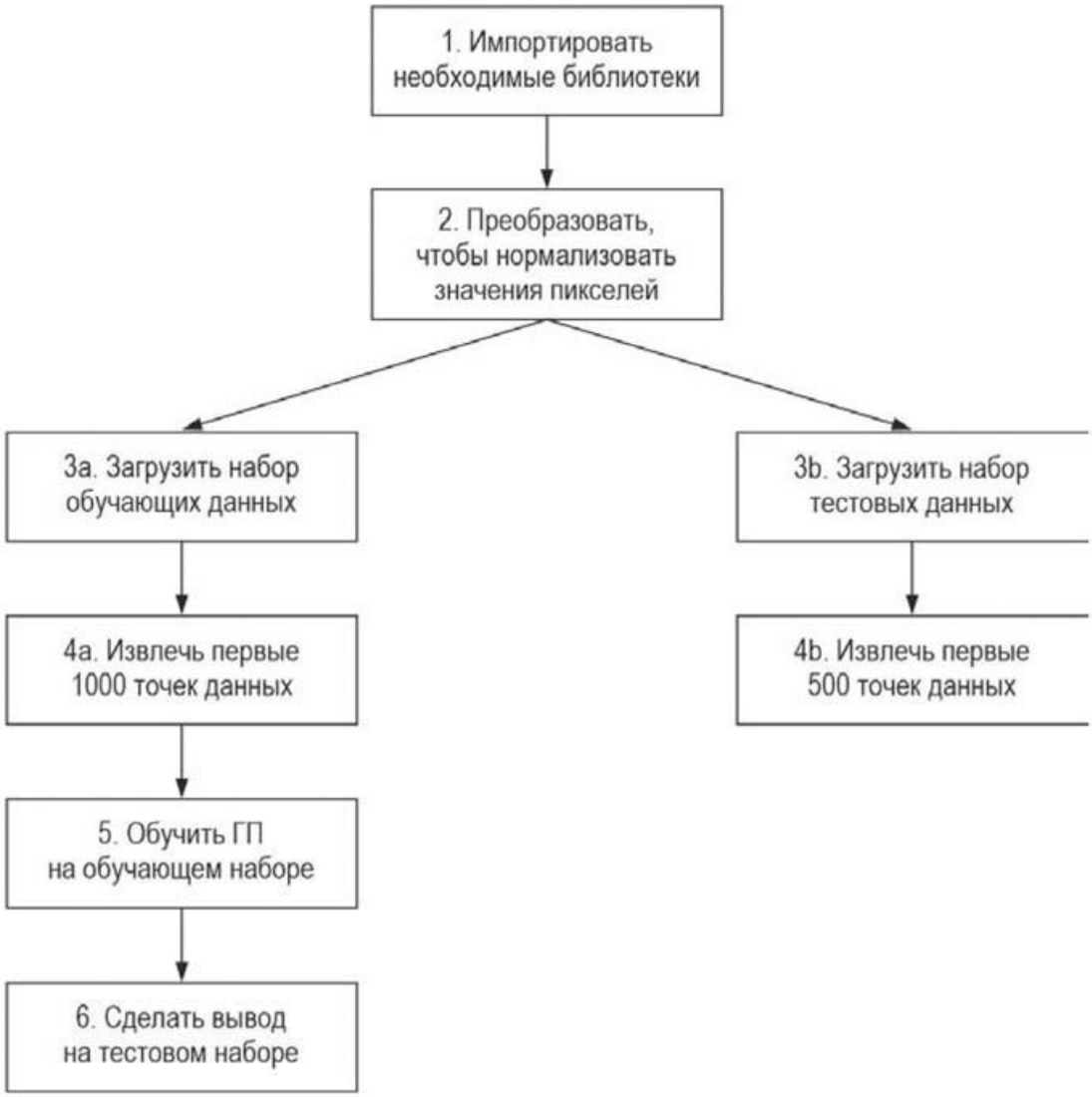


Рис. 13.8. Блок-схема обучения ГП на данных MNIST: сначала мы извлекаем 1000 точек данных, чтобы составить обучающий набор, затем еще 500 точек в качестве тестового набора

Сначала необходимо импортировать PyTorch и его расширение torchvision, которое управляет функциями и наборами данных (например, MNIST), связанными с компьютерным зрением. Из torchvision нужно импортировать наборы данных и преобразования модулей, которые помогают загружать и манипулировать данными MNIST:

```
import torch
from torchvision import datasets, transforms
```

На втором этапе подготовки данных мы снова используем объект, который преобразует изображения в тензоры PyTorch (это структура данных, которую могут обрабатывать ГП, реализованные в GPyTorch). Также он нормализует значения пикселей с помощью вычитания из значений пикселей числа 0,1307 (среднее значе-

ние данных) и последующего деления на 0,3081 (стандартное отклонение данных). Такой процесс считается обычной практикой для набора MNIST (более подробную информацию об этом шаге можно найти на официальном форуме PyTorch):

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

Объект преобразования, хранящийся в `transform`, теперь можно передать при вызове любой инициализации набора данных torchvision. Конвертация в тензоры PyTorch и нормализация будут применены к данным. Инициализация набора MNIST с помощью объекта преобразования показана ниже. Обратите внимание, что набор данных мы создаем дважды, один раз устанавливая `train = True` для создания обучающего набора и один раз устанавливая `train = False` для тестового набора:

```
#Загружаем и импортируем обучающий набор
dataset1 = datasets.MNIST(
    "../data", train=True, download=True, transform=transform
)

#Загружаем и импортируем тестовый набор
dataset2 = datasets.MNIST(
    "../data", train=False, download=True, transform=transform
)
```

На последнем шаге мы извлекаем первые 1000 точек данных из обучающего набора и 500 точек из тестового набора. Это делается через обращения к объектам `dataset1` и `dataset2`:

- Атрибут `data` нужен для получения характеристик (значений пикселей), составляющих изображение для каждой точки данных.
- Атрибут `targets` нужен для получения меток (значений рукописных цифр):

```
#Получаем первые 1000 точек в обучающем наборе
train_x = dataset1.data[:1000, ...].view(1000, -1).to(torch.float)
train_y = dataset1.targets[:1000]

#Получаем первые 500 точек в тестовом наборе
test_x = dataset2.data[:500, ...].view(500, -1).to(torch.float)
test_y = dataset2.targets[:500]
```

Мы также реализуем простую модель, имеющую постоянное среднее значение и ядро RBF с масштабом вывода:

```
class GPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super().__init__(train_x, train_y, likelihood)
        #Постоянная функция среднего
        self.mean_module = gpytorch.means.ConstantMean()
```

```
        #Ковариационная функция RBF с масштабом вывода
        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.RBFKernel()
        )

    #Создаем MVN-распределение в качестве прогноза для входных данных x
    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

Примечание

Метод `forward()` модели GPyTorch впервые обсуждался в разделе 2.4.

Затем мы инициализируем ГП и обучаем его на наборе из 1000 точек, используя градиентный спуск с оптимизатором Adam. Код ниже оптимизирует значения гиперпараметров ГП (например, среднюю константу, масштаб длины и масштаб вывода), чтобы достичь высокой предельной вероятности наблюдаемых данных:

```
#Объявляем функцию вероятности и модель ГП
likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = GPModel(train_x, train_y, likelihood)

#Объявляем алгоритм градиентного спуска и функцию потерь
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

#Включаем режим обучения
model.train()
likelihood.train()

#Выполняем 500 итераций градиентного спуска
for i in tqdm(range(500)):
    optimizer.zero_grad()

    output = model(train_x)
    loss = -mll(output, train_y)

    loss.backward()
    optimizer.step()

#Включаем режим прогнозирования
model.eval()
likelihood.eval()
```

Примечание

Как градиентный спуск оптимизирует вероятность наблюдаемых данных (обучает ГП), можно повторить в разделе 2.3.2.

Наконец, чтобы увидеть, насколько хорошо модель работает на тестовом наборе, вычислим среднюю абсолютную разницу между предсказанием ГП и основной истиной (значение метки каждой точки данных). Такую метрику часто называют *средней абсолютной ошибкой* (**Mean Absolute Error**, **MAE**).

> **Примечание**
>
> Типичным показателем для набора MNIST является процент тестового набора, который модель предсказывает правильно (точность), что является нормой для задачи классификации. Но поскольку мы используем этот набор данных для моделирования задачи регрессии, нам подходит метрика средней абсолютной ошибки.

Это делается путем сравнения средних прогнозов с истинными метками, хранящимися в `test_y`:

```
with torch.no_grad():
    mean_preds = model(test_x).mean

print(torch.mean(torch.abs(mean_preds - test_y)))

Output: 2.7021167278289795
```

Этот результат означает, что в среднем прогноз для значения цифры на изображении отличается почти на 3. Такая производительность является довольно низкой, учитывая, что в данной задаче нужно исследовать только 10 значений. Результат подчеркивает неспособность обычных моделей работать со структурированными данными.

13.3. Использование нейросетей для обработки сложных структурированных данных

Основная причина низкой производительности ГП заключается в том, что ядра не способны обрабатывать сложную структуру входных данных. Это приводит к ошибочному расчету ковариации. Ядро RBF имеет простую форму, которая учитывает только разницу числовых значений между двумя входными данными. В этом разделе мы научимся решать данную проблему, используя нейросеть для обработки структурированных данных, прежде чем передавать их в функцию среднего и ядро ГП.

13.3.1. Зачем использовать нейросети для моделирования?

В начале книги отмечалось, что нейросети не очень хороши в создании прогнозов с калибровкой по неопределенности, особенно когда получение данных обходится дорого. Поэтому в байесовской оптимизации используются гауссовские процессы. Однако нейросети хороши в изучении сложных структур. Такая гибкость обусловлена наличием нескольких уровней вычислений (в частности, умножения матриц), как показано на рис. 13.9.

В нейросети каждый слой соответствует матричному умножению, выходные данные которых затем проходят через нелинейную функцию *активации*. Благодаря

объединению множества таких слоев, за один прямой проход можно обрабатывать входные данные и манипулировать ими гибким способом. В итоге нейросеть может хорошо моделировать сложные функции. Подробные объяснения можно найти в книге Франсуа Шолле *«Deep Learning with Python, Second Edition»* (*«Глубокое обучение с помощью Python», второе издание*, издательство Manning, 2021).

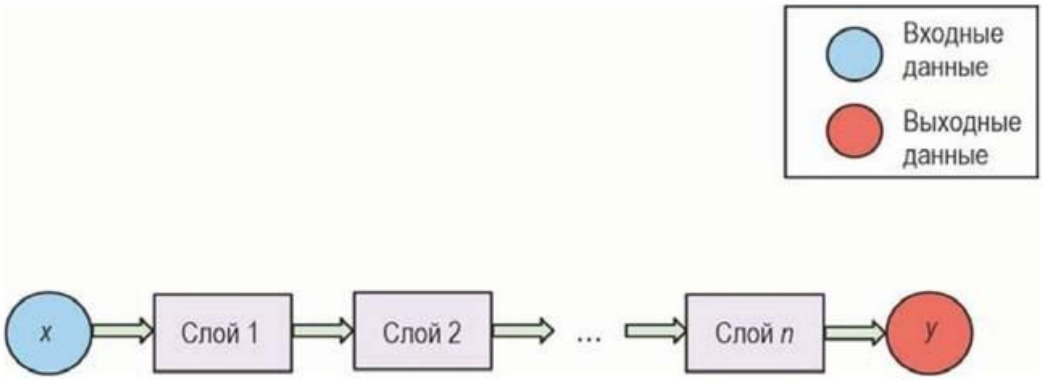


Рис. 13.9. Нейронная сеть — это совокупность многоуровневых вычислений, которая может хорошо моделировать сложные функции

Замечание

Благодаря своей гибкости нейросети хорошо справляются со многими задачами обучения, включающими структурированные данные вроде изображений, графиков и белковых структур.

Гибкость, которой обладают нейросети, может помочь решить задачу, описанную в предыдущем разделе. Если ядро ГП не может хорошо обрабатывать сложные структуры данных, лучше оставить эту работу для нейросети и передавать ядру только обработанные данные. Такая процедура показана на рис. 13.10, где входные данные x сначала проходят через слои нейросети, а затем передаются в функцию среднего и ядро ГП.

Несмотря на то что конечным результатом по-прежнему является MVN-распределение, входные данные теперь представляют *обработанный ввод*, полученный нейросетью. Такой подход является многообещающим, поскольку благодаря

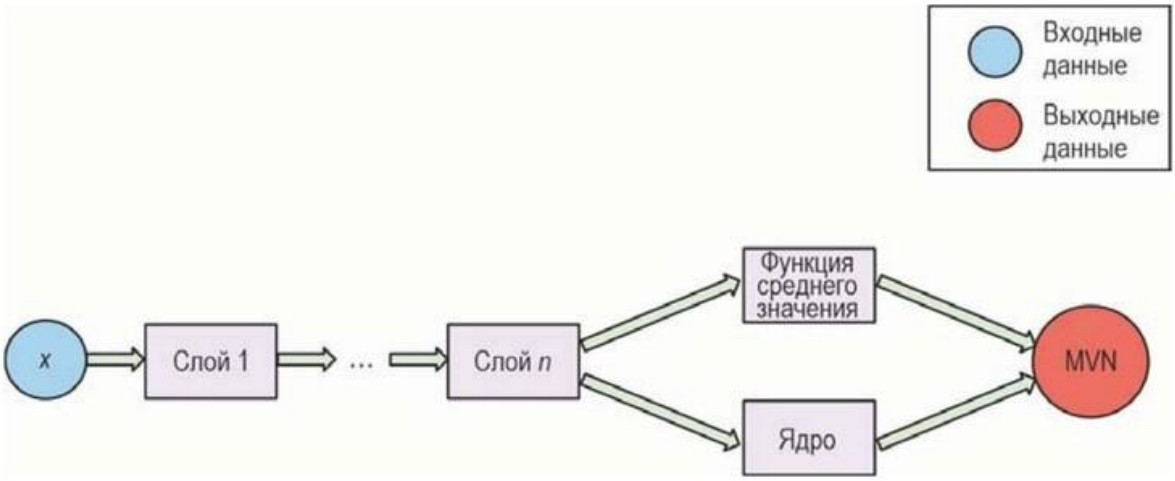


Рис. 13.10. Объединение нейросети и ГП: нейросеть сначала обрабатывает входные структурированные данные x, а затем передает их в функцию среднего и ядро

своим гибким возможностям моделирования нейросеть может извлекать важные характеристики из структурированных входных данных (что важно с точки зрения предоставления информации для расчета сходства) и сводить их к числовым значениям, которые ядро ГП может обработать.

Определение

Нейросеть часто называют *экстрактором характеристик* (**Feature extractor**) комбинированной модели, поскольку она *извлекает* (extract) из структурированных данных особенности (признаки, характеристики), способствующие ГП-моделированию.

Таким образом, можно воспользоваться преимуществами гибкого обучения нейросети, сохраняя при этом возможность делать прогнозы с калибровкой по неопределенности, используя ГП. Это лучшее от обеих сторон! Кроме того, обучение комбинированной модели следует той же процедуре, что и обучение обычного ГП. Мы определяем функцию потерь, которая представляет собой отрицательную логарифмическую вероятность, и используем градиентный спуск, чтобы найти значения для гиперпараметров, которые лучше всего объясняют данные (путем минимизации потерь). Вместо оптимизации только средней константы, а также масштабов длины и вывода теперь дополнительно оптимизируются еще и веса нейросетей. В следующем разделе мы увидим, что реализация данной процедуры обучения с помощью GPyTorch требует минимальных изменений по сравнению с тем, что мы делали в предыдущих главах.

Примечание

Такая комбинированная структура представляет собой метод *динамического обучения* (**Dynamically learning**) при обработке структурированных данных исключительно на основе нашего набора обучающих данных. Раньше мы использовали только фиксированное ядро, которое обрабатывало информацию одинаково в нескольких приложениях. Здесь мы на лету узнаем, как лучше всего обрабатывать входные данные, уникальные для поставленной задачи. Это связано с тем, что веса нейросети оптимизированы по отношению к обучающим данным.

13.3.2. Реализация комбинированной модели в GPyTorch

Теперь мы можем реализовать эту структуру и применить ее к набору данных MNIST. Здесь определение класса модели является более сложным, поскольку нужно реализовать нейросеть и подключить ее к нему. Сначала сделаем первую часть — нейросеть. Она будет простой. Ее архитектура показана на рис. 13.11. Она имеет четыре уровня с 1000, 5000, 50 и 2 узлами соответственно. Это общая архитектура набора данных MNIST.

Примечание

Обратите внимание на размер последнего слоя (2). Он обозначает размерность обработанного вывода, который будет подан в функцию среднего и ядро ГП. Установив размер этого слоя равным 2, мы нацелены изучать представления изображений, существующих в двумерном пространстве. Можно использовать значения, отличные от 2. Здесь оно выбрано лишь в целях демонстрации.

Реализуем эту архитектуру, используя классы `Linear()` и `ReLU()` из PyTorch. Каждый уровень сети реализован в качестве модуля `torch.nn.Linear` соответствующего размера, как показано на рис. 13.11. Каждый модуль также связан с модулем функции

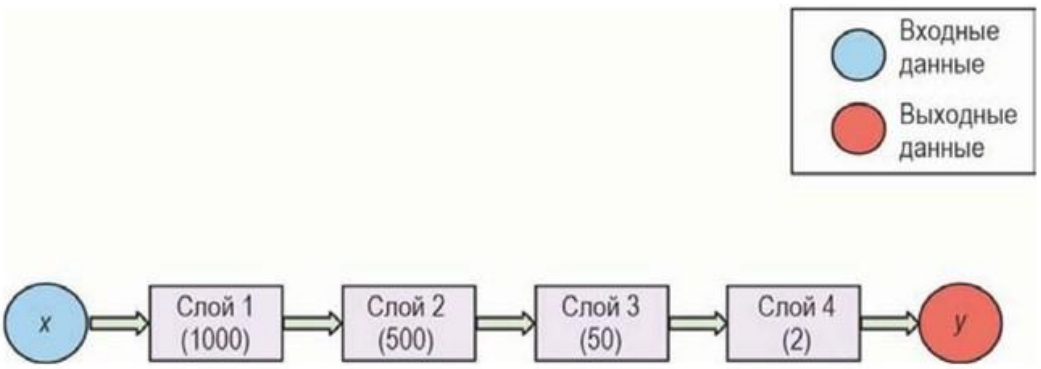


Рис. 13.11. Архитектура реализуемой нейросети: она имеет четыре слоя и создает массив размером 2 для каждой точки входных данных

активации `torch.nn.ReLU`, который реализует упомянутое ранее нелинейное преобразование. Это показано на рис. 13.12, где каждый компонент сетевой архитектуры помечен соответствующим кодом, который его реализует.

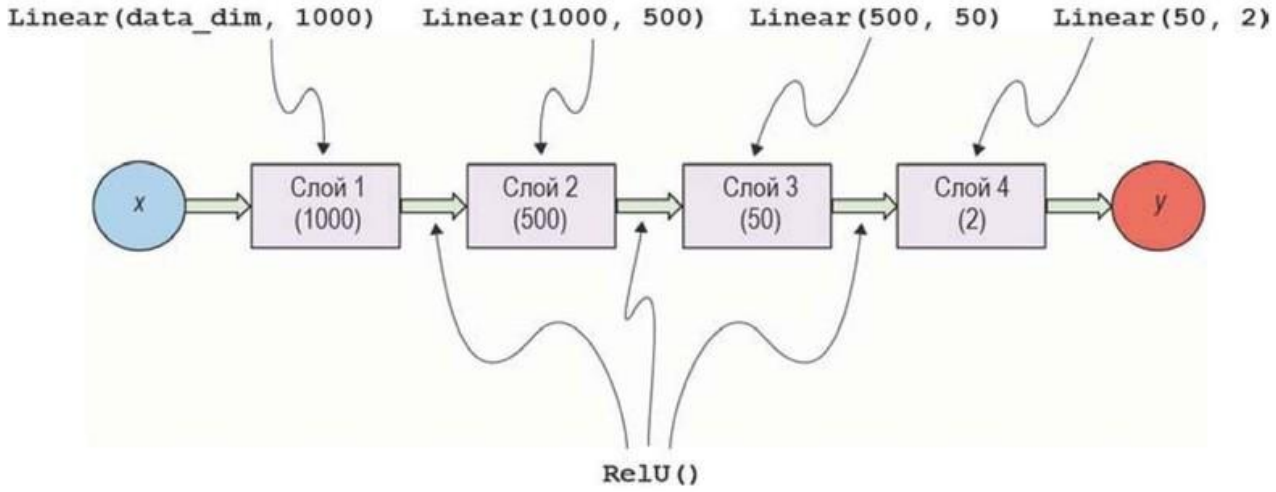


Рис. 13.12. Архитектура реализуемой нейросети и соответствующий код PyTorch: каждый уровень реализуется с помощью `torch.nn.Linear`, а каждая функция активации — с помощью `torch.nn.ReLU`

Используя удобный метод `add_module()`, мы неявно определяем логику метода `forward()`. Далее реализуем модель с помощью класса `LargeFeatureExtractor`. Он последовательно передает входные данные x через слои, реализованные в методе `__init__()`, который принимает `data_dim` — размерность входных данных. В нашем случае это число составляет $28 \times 28 = 784$, и мы вычисляем его с помощью `train_x.size(-1)`:

```
#Размерность данных
data_dim = train_x.size(-1)

class LargeFeatureExtractor(torch.nn.Sequential):
    def __init__(self, data_dim):
        super(LargeFeatureExtractor, self).__init__()

        #Первый слой нейросети
        self.add_module('linear1', torch.nn.Linear(data_dim, 1000))
        self.add_module('relu1', torch.nn.ReLU())
```

```
        #Второй слой нейросети
        self.add_module('linear2', torch.nn.Linear(1000, 500))
        self.add_module('relu2', torch.nn.ReLU())

        #Третий слой нейросети
        self.add_module('linear3', torch.nn.Linear(500, 50))
        self.add_module('relu3', torch.nn.ReLU())

        #Четвертый слой нейросети
        self.add_module('linear4', torch.nn.Linear(50, 2))

#Инициализируем сеть
feature_extractor = LargeFeatureExtractor(data_dim)
```

Далее обсудим комбинированную модель, которая представляет собой класс модели, использующий `feature_extractor` — экстрактор функций нейросети, который мы только что инициализировали. Сначала реализуем его метод `__init__()`, который состоит из нескольких компонентов:

1. Модуль ковариации обернут в объект `gpytorch.kernels.GridInterpolationKernel`, который обеспечивает ускорение вычислений для среднеразмерного обучающего набора из 1000 точек. Количество измерений равно 2, поскольку оно равно размерности выходных данных, создаваемых экстрактором характеристик.
2. Сам экстрактор является переменной `feature_extractor`, которая была объявлена ранее.
3. Выходные данные экстрактора могут принимать экстремальные значения (отрицательную или положительную бесконечность), если веса нейросети плохо инициализированы. Чтобы решить эту проблему, нужно масштабировать выходные значения до диапазона от –1 до 1, используя модуль `gpytorch.utils.grid.ScaleToBounds`.

Метод `__init__()` реализован следующим образом:

```
class GPRegressionModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super(GPRegressionModel, self).__init__(train_x, train_y, likelihood)

        self.mean_module = gpytorch.means.ConstantMean()

        #Двумерное ядро RBF с ускорением вычислений
        self.covar_module = gpytorch.kernels.GridInterpolationKernel(
            gpytorch.kernels.ScaleKernel(
                gpytorch.kernels.RBFKernel(ard_num_dims=2)
            ),
            num_dims=2,
            grid_size=100
        )

        #Экстрактор характеристик
        self.feature_extractor = feature_extractor
```

```
        #Модуль для масштабирования результатов нейросети до разумных значений
        self.scale_to_bounds = gpytorch.utils.grid.ScaleToBounds(-1., 1.)
```

В методе `forward()` все компоненты объединяются. Сначала мы используем экстрактор для обработки входных данных. Затем передаем их (уже обработанные) в модули среднего и ковариации нашей модели. В конце концов, мы все равно получаем MVN-распределение, которое возвращает метод `forward()`:

```
class GPRegressionModel(gpytorch.models.ExactGP):
    def forward(self, x):
        #Масштабированный вывод экстрактора характеристик
        projected_x = self.feature_extractor(x)
        projected_x = self.scale_to_bounds(projected_x)

        #Создаем объект MVN-распределения из обработанных входных данных
        mean_x = self.mean_module(projected_x)
        covar_x = self.covar_module(projected_x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

Наконец, чтобы обучить комбинированную модель с использованием градиентного спуска, нужно объявить еще некоторые объекты. В дополнение к обычным гиперпараметрам, таким как средняя константа, масштабы длины и вывода, мы также оптимизируем веса экстрактора характеристик, которые хранятся в `model.feature_extractor.parameters()`:

```
#Функция вероятности, модель ГП и функция потерь такие же, как и раньше
likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = GPRegressionModel(train_x, train_y, likelihood)
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

#Оптимизатору градиентного спуска Adam теперь необходимо оптимизировать
#веса экстрактора характеристик и гиперпараметров ГП
optimizer = torch.optim.Adam([
    {'params': model.feature_extractor.parameters()},
    {'params': model.covar_module.parameters()},
    {'params': model.mean_module.parameters()},
    {'params': model.likelihood.parameters()},
], lr=0.01)
```

Теперь мы можем запустить градиентный спуск так же, как и раньше:

```
#Включаем режим обучения
model.train()
likelihood.train()

#Запускаем 500 итераций градиентного спуска
for i in tqdm(range(500)):
    optimizer.zero_grad()

    output = model(train_x)
    loss = -mll(output, train_y)
```

```
    loss.backward()
    optimizer.step()

#Включаем режим прогнозирования
model.eval()
likelihood.eval()
```

> **Примечание**
>
> Стоит напомнить, что при обучении ГП необходимо включить режим обучения, как для модели, так и для вероятности (используя `model.train()` и `likelihood.train()`). После обучения и перед тем, как делать прогнозы, нужно включить режим прогнозирования (с помощью `model.eval()` и `likelihood.eval()`).

Мы обучили модель в сочетании с экстрактором характеристик. Прежде чем делать прогнозы на тестовом наборе с использованием этой модели, мы можем заглянуть внутрь нее и посмотреть, научился ли экстрактор хорошо обрабатывать данные. Стоит помнить, что каждое изображение преобразуется в массив из двух элементов. Мы уже можем передать обучающие данные через экстрактор и визуализировать выходные данные с помощью диаграммы.

> **Примечание**
>
> Обучение комбинированной модели занимает больше времени, чем обучение обычного ГП. Это потому, что теперь нам нужно оптимизировать гораздо больше параметров. Однако эти затраты того стоят, поскольку достигается гораздо более высокий прирост производительности.

Если на диаграмме появятся кластеры точек с одинаковой меткой (изображения с одной и той же цифрой), это будет признаком, что экстрактор характеристик смог эффективно обучиться на данных. Это делается путем передачи обучающих данных через экстрактор так же, как данные обрабатываются в методе `forward()` класса модели:

```
with torch.no_grad():
    extracted_features = model.feature_extractor(train_x)
    extracted_features = model.scale_to_bounds(extracted_features)
```

Здесь `extracted_features` — это тензор PyTorch размером 1000×2, в котором хранятся двумерные извлеченные из обучающего набора характеристики для 1000 точек данных. Чтобы визуализировать тензор на диаграмме, воспользуемся `plt.scatter()` из библиотеки Matplotlib, проверяя, что каждая метка соответствует цвету:

```
for label in range(10):
    #Фильтруем точки данных с определенной меткой
    mask = train_y == label

    #Создаем точечную диаграмму для текущих точек данных,
    #которые имеют один и тот же цвет
    plt.scatter(
        extracted_features[mask, 0],
        extracted_features[mask, 1],
        c=train_y[mask],
        vmin=0,
```

```
        vmax=9,
        label=label,
    )
```

Результат можно увидеть на рис. 13.13. Он может отличаться в зависимости от версии библиотеки и системы, в которой работает код. Как и ожидалось, точки данных с одной и той же меткой группируются друг вокруг друга. Это означает, что экстрактор характеристик успешно группирует точки с одной и той же меткой. После обработки сетью два изображения с одной и той же меткой становятся двумя точками, расположенными близко друг к другу в двумерной плоскости, которые затем будут иметь высокую ковариацию при вычислении ядром RBF. Это именно то, что мы хотели, чтобы наш экстрактор помог нам сделать!

Еще один интересный аспект рис. 13.13 заключается в том, что существует явный градиент относительно значения метки: при движении от нижнего кластера к верх-

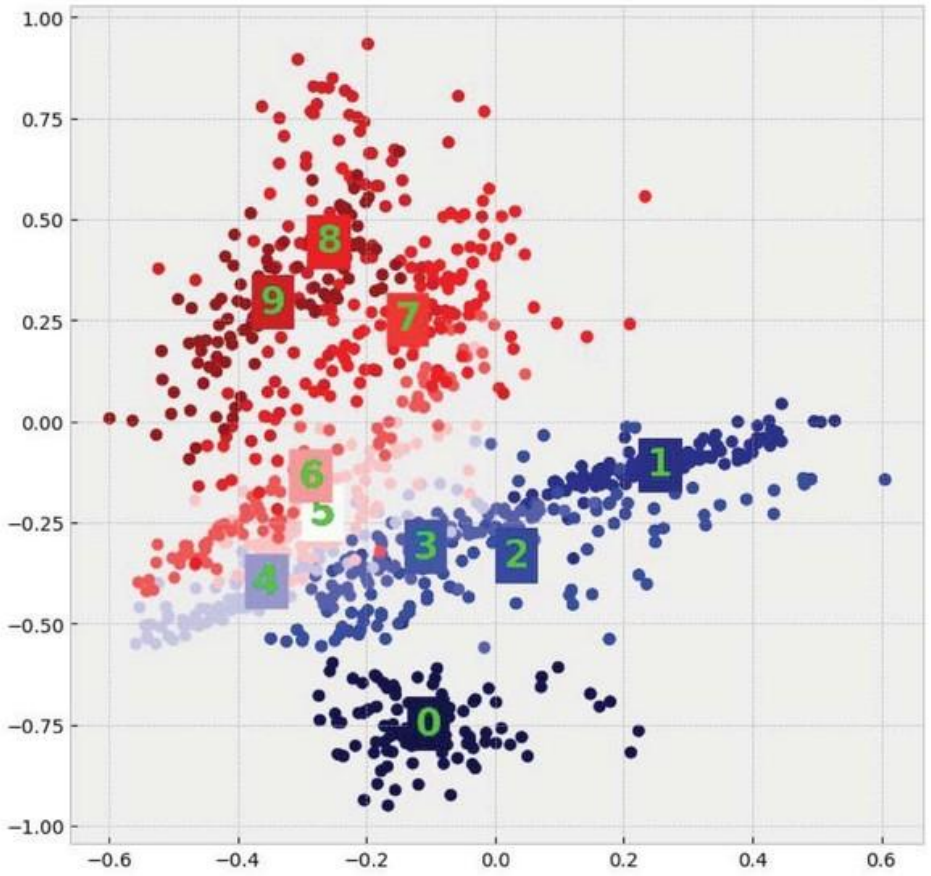


Рис. 13.13. Характеристики, извлеченные из набора данных MNIST с помощью нейросети: точки данных с одной и той же меткой не просто группируются вместе, на графике существует целый градиент меток; при движении снизу вверх значение метки увеличивается

нему значение соответствующей метки постепенно увеличивается от 0 до 9. Это отличная возможность экстрактора, поскольку она указывает, что модель нашла гладкое представление изображений MNIST, которое соответствует значениям меток.

Замечание

Чем более гладкими являются извлеченные характеристики, тем легче для ГП изучить их.

Рассмотрим сравнение на рис. 13.14, где левая часть демонстрирует рис. 13.13, а правая — случайную замену меток одной и той же диаграммы, что делает объекты более «грубыми». Под «грубым» подразумевается, что значения меток хаотично колеблются: нижний кластер содержит 0, некоторые из кластеров в середине соответствуют 7 и 9, а верхний кластер содержит 5. Другими словами, тенденция появления меток с грубыми чертами не является монотонной, что затрудняет обучение ГП.

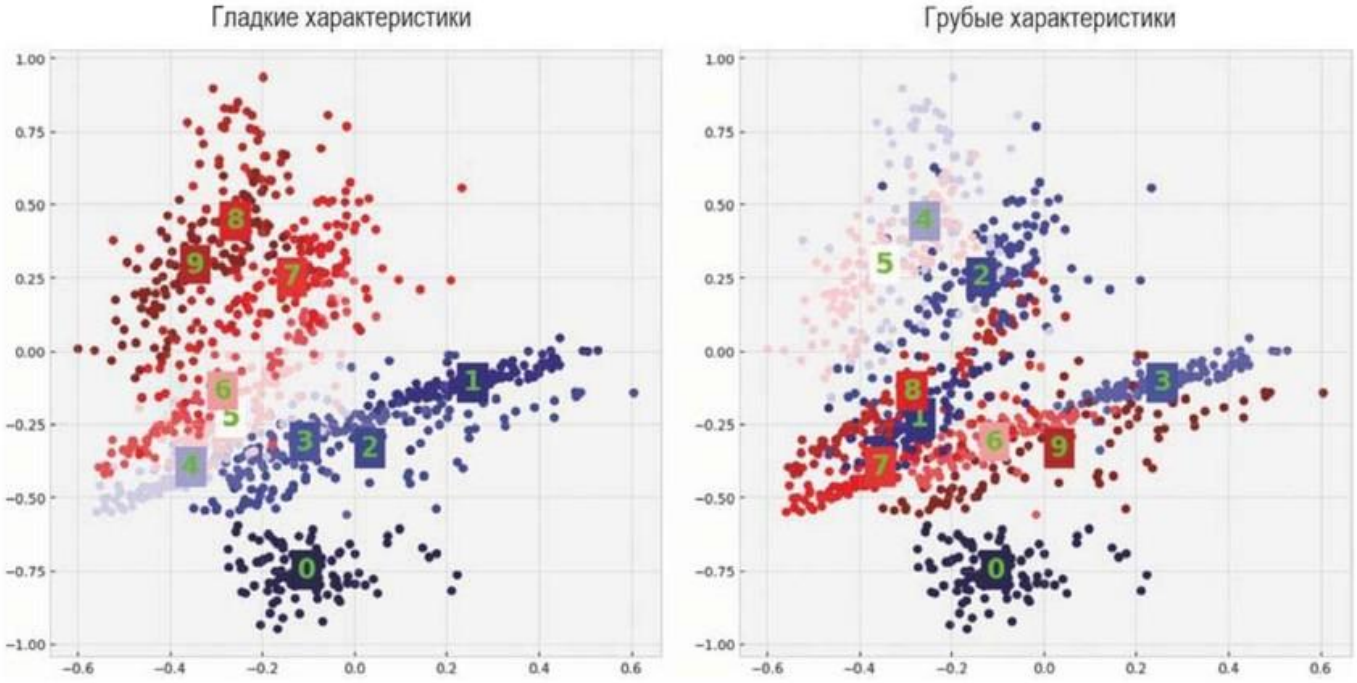


Рис. 13.14. Сравнение сглаженных извлеченных характеристик на рис. 13.13 (слева) и случайной замены меток, делающей объекты грубыми (справа): гладкие черты легче изучить с помощью ГП, чем грубые

Можно наблюдать, что нейросеть неплохо справляется с извлечением полезных признаков из изображений. Чтобы увидеть, действительно ли это приводит к повышению эффективности прогнозирования, снова вычислим среднюю абсолютную ошибку (MAE):

```
with torch.no_grad():
    mean_preds = model(test_x).mean

print(torch.mean(torch.abs(mean_preds - test_y)))

Output: 0.8524129986763
```

Результат говорит о том, что в среднем прогноз отклоняется на значение 0,85. Это значительное улучшение по сравнению с обычным ГП из предыдущего раздела, MAE которого составляла примерно 2,7. Улучшение демонстрирует превосходную производительность комбинированной модели, обусловленную гибкими возможностями нейросетей для моделирования.

Как говорилось вначале, такая структура применима не только к рукописным цифрам, но и к различным типам структурированных данных, на которых может учиться нейросеть, включая другие типы изображений и графовых структур, вроде молекул и протеинов. Все, что нужно сделать, это определить соответствующую архи-

тектуру, которая извлекает функции из структурированных данных, передаваемых затем функции среднего значения и ядру ГП.

На этом можно завершить главу 13. В ней мы узнали о сложности обучения на таких структурированных данных, как изображения, где обычные ядра не могут эффективно вычислять ковариации между точками. Подключив экстрактор характеристик (признаков) к ГП, мы научились преобразовывать структурированные данные в форму, которую затем сможет обработать ядро. Конечным результатом является комбинированная модель, которая может гибко учиться на структурированных данных, но при этом давать вероятностные прогнозы с количественной оценкой по неопределенности.

Заключение

- Структурированные данные — это данные, характеристики (признаки) которых должны удовлетворять ограничениям, например быть целыми или неотрицательными. Также они не могут рассматриваться как непрерывные данные с действительным значением. Это могут быть изображения в компьютерном зрении или белковые структуры при открытии лекарств.
- Структурированные данные представляют собой проблему для обычных ядер ГП. Это связано с тем, что ядра учитывают только числовые значения входных данных, что может быть плохим предсказательным свойством.
- Ядро, использующее неправильные характеристики для вычисления ковариаций, может привести к низкому качеству прогнозов на основе полученного ГП. Это особенно часто встречается при работе со структурированными данными.
- В частности, для изображений необработанные значения пикселей не являются информативным признаком. Ядро, вычисляющее ковариации с использованием необработанных значений пикселей, может привести к получению ГП низкого качества.
- Имея несколько уровней нелинейных вычислений, нейросети эффективны при изучении сложных функций и могут извлекать особенности из структурированных данных. Используя нейросеть для извлечения непрерывных реальных характеристик из структурированных данных, ГП по-прежнему может эффективно обучаться.
- Объединив нейросеть с ГП, мы можем динамически изучать способ обработки данных, посвященный рассматриваемой задаче. Такая гибкость позволяет комбинированной модели обобщать многие виды структурированных данных.
- Важно масштабировать выходные данные нейросети до небольшого диапазона, прежде чем передавать их в ГП. При этом мы должны избегать экстремальных значений, которые могут возникнуть в результате неправильно инициализированного экстрактора характеристик.
- Представление данных, полученное с помощью экстрактора, демонстрирует плавный градиент по отношению к меткам. Он делает извлеченные функции более пригодными для изучения с помощью ГП.

ПРИЛОЖЕНИЕ

Решения для упражнений

П.1. Глава 2: Гауссовский процесс как распределение по функциям

В этом упражнении мы обучаем ГП на реальном наборе данных, который уже фигурировал в главе 1. Решение включено в `CH02/02 - Exercise.ipynb`. Выполните следующие шаги:

1. Создайте четырехмерный набор данных. Сначала импортируйте необходимые библиотеки — PyTorch для манипуляций с массивами/тензорами, GPyTorch для ГП-моделирования и Matplotlib для визуализации:

```
import torch
import gpytorch
import matplotlib.pyplot as plt
```

   Затем сохраните числа в таблице в двух тензорах (`train_x` и `train_y`), которые содержат функции и метки набора данных соответственно:

```
train_x = torch.tensor(
    [
        [1 / 2, 1 / 2, 0, 0],
        [1 / 3, 1 / 3, 1 / 3, 0],
        [0, 1 / 2, 1 / 2, 0],
        [0, 1 / 3, 1 / 3, 1 / 3],
    ]
)

train_y = torch.tensor([192.08, 258.30, 187.24, 188.54])
```

2. Нормализуйте пятый столбец, вычтя среднее из всех значений и разделив результаты на их стандартное отклонение.

   Нормализуйте метки следующим образом:

```
#Нормализуем метки
train_y = (train_y - train_y.mean()) / train_y.std()
```

   При выводе `train_y` должен содержать следующие значения: `tensor([-0,4183, 1,4974, -0,5583, -0,5207])`.

3. Первые четыре столбца необходимо считать объектами, а пятый — метками. Обучите ГП на этих данных.

   Класс модели переопределяется следующим образом:

```
class BaseGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super().__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ZeroMean()
        self.covar_module = gpytorch.kernels.RBFKernel()

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

   Затем проинициализируйте объект этого класса с обучающими данными:

```
lengthscale = 1
noise = 1e-4

likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = BaseGPModel(train_x, train_y, likelihood)

model.covar_module.lengthscale = lengthscale
model.likelihood.noise = noise

model.eval()
likelihood.eval()
```

4. Создайте тестовый набор данных, содержащий сплавы с нулевым процентом германия и марганца.

   Чтобы собрать тестовый набор данных, сначала создайте сетку для первого и второго столбцов, охватывающую единичную площадь:

```
grid_x = torch.linspace(0, 1, 101)
grid_x1, grid_x2 = torch.meshgrid(grid_x, grid_x, indexing="ij")
```

   Эти столбцы хранятся в `grid_x1` и `grid_x2`. Затем добавьте два дополнительных столбца со всеми нулями в `grid_x1` и `grid_x2`, завершая тестовый набор четырьмя столбцами:

```
xs = torch.vstack(
    [
        grid_x1.flatten(),     #Первый столбец
        grid_x2.flatten(),     #Второй столбец
        torch.zeros(101 ** 2), #третий столбец со всеми нулями
        torch.zeros(101 ** 2), #Четвертый столбец со всеми нулями
    ]
).transpose(-1, -2)
```

5. Спрогнозируйте температуру смешивания на данном тестовом наборе.

   Чтобы это сделать, нужно просто передать `xs` через модель ГП в контексте `torch.no_grad()`:

```
with torch.no_grad():
    predictive_distribution = likelihood(model(xs))
    predictive_mean = predictive_distribution.mean
    predictive_stddev = predictive_distribution.stddev
```

6. Визуализируйте прогнозы.

   Чтобы это сделать, сначала нужно создать рисунок с двумя частями (двумя подграфиками Matplotlib):

```
fig, ax = plt.subplots(1, 2, figsize=(16, 6))
```

   Затем используйте `plt.imshow()` для визуализации векторов среднего значения и стандартного отклонения в виде тепловых карт, обязательно преобразуя их в квадратные матрицы:

```
c = ax[0].imshow(
    predictive_mean.detach().reshape(101, 101).transpose(-1, -2),
    origin="lower",
    extent=[0, 1, 0, 1],
) #Тепловая карта для прогностического среднего значения

c = ax[1].imshow(
    predictive_stddev.detach().reshape(101, 101).transpose(-1, -2),
    origin="lower",
    extent=[0, 1, 0, 1],
) #Тепловая карта для прогностического стандартного отклонения

plt.colorbar(c, ax = ax[1])
```

   Это создаст графики, как на рис. П.1.

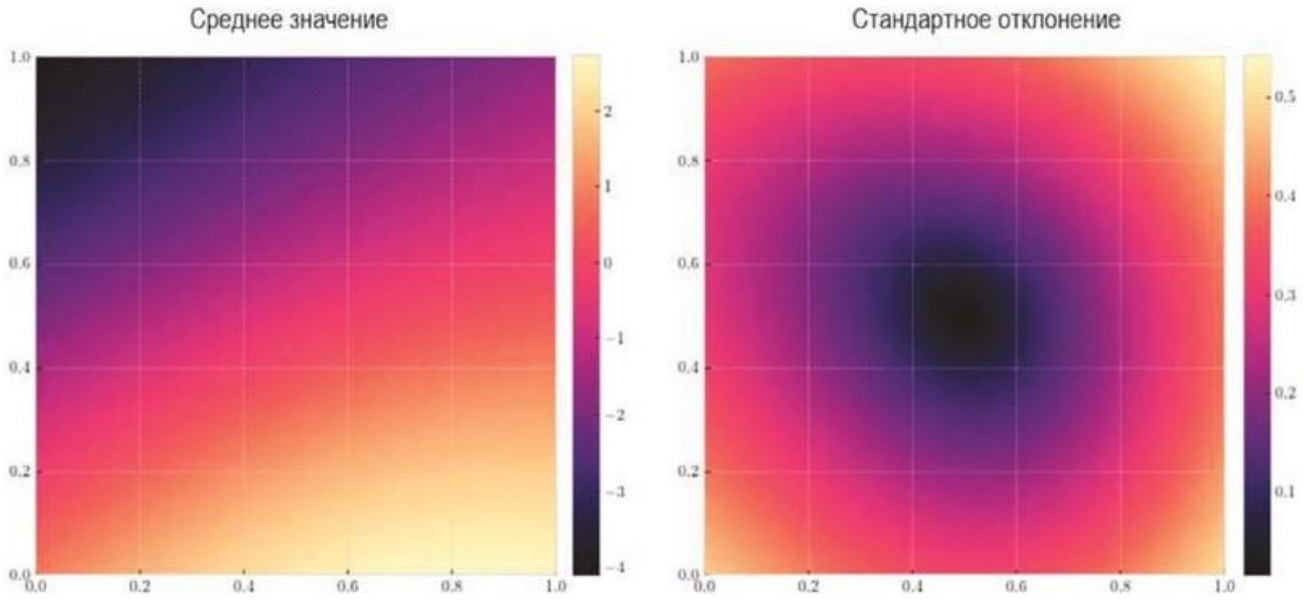


Рис. П.1. Прогнозы ГП в двумерном пространстве

Примечание

Если вы используете другую реализацию ГП, вполне есть возможность создать тепловые карты, которые будут немного отличаться от тех, что показаны на рис. П.1. Если общая тенденция тепловых карт одинакова, ваше решение правильное.

П.2. Глава 3: Объединение предварительных знаний с функциями среднего и ковариации

В этом упражнении представлена практика, как реализовать модель ГП с автоматическим определением релевантности (ARD). Решение включено в `CH03/03 - Exercise.ipynb`.

Выполните следующие шаги:

1. Реализуйте двумерную функцию на Python с помощью PyTorch.

   Сначала импортируйте необходимые библиотеки — PyTorch для манипуляций с массивами/тензорами, GPyTorch для ГП-моделирования и Matplotlib для визуализации:

   ```
   import torch
   import gpytorch
   import matplotlib.pyplot as plt
   ```

   Затем реализуйте целевую функцию, используя данную формулу:

   ```
   def f(x):
       return (
        torch.sin(5 * x[..., 0] / 2 - 2.5) * torch.cos(2.5 - 5 * x[..., 1])
          + (5 * x[..., 1] / 2 + 0.5) ** 2 / 10
       ) / 5 + 0.2
   ```

2. Визуализируйте функцию в области $[0, 2]^2$.

   Чтобы это сделать, нужно создать сетку области, которая хранится в `xs`:

   ```
   lb = 0
   ub = 2
   xs = torch.linspace(lb, ub, 101) #Одномерная сетка
   x1, x2 = torch.meshgrid(xs, xs)
   #Двумерная сетка
   xs = torch.vstack((x1.flatten(), x2.flatten())).transpose(-1, -2)
   ```

   Затем получите значения функции по этой сетке, передав `xs` в `f()`. Результаты сохраняются в `ys`:

   ```
   ys = f(xs)
   ```

   Для визуализации `ys` в виде тепловой карты используйте функцию `plt.imshow()`:

   ```
   plt.imshow(ys.reshape(101, 101).T, origin="lower", extent=[lb, ub, lb, ub])
   ```

3. Извлеките 100 случайных точек данных из области $[0, 2]^2$. Они будут использоваться в качестве обучающих данных.

Чтобы сделать это в пределах области, можно воспользоваться `torch.rand()` для выборки из единичного квадрата, а затем умножить результат на 2, чтобы масштабировать его на нашу область:

```
torch.manual_seed(0)
train_x = torch.rand(size=(100, 2)) * 2
```

Значения функции в этих точках можно получить, вызвав `f(train_x)`:

```
train_y = f(train_x)
```

4. Реализуйте модель ГП с постоянной функцией среднего и ядром Матерна 5/2 с масштабом вывода, которая выполнена как объект `gpytorch.kernels.ScaleKernel`:

```
class GPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super().__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.MaternKernel(
                nu=2.5,
                #Чтобы отключить ARD, установите значение «None
                #Чтобы включить ARD, установите значение «2»
                ard_num_dims=None
            )
        )

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

5. Не указывайте параметр `ard_num_dims` при инициализации объекта ядра и не устанавливайте для этого параметра значение `None`.

   Это сделано в предыдущем коде.

6. Обучите гиперпараметры модели ГП, используя градиентный спуск, и проверьте масштаб длины после обучения.

   Инициализируйте ГП и затем обучите его с помощью градиентного спуска в течение 500 итераций следующим образом:

```
noise = 1e-4

likelihood = gpytorch.likelihoods.GaussianLikelihood()
model = GPModel(train_x, train_y, likelihood)

model.likelihood.noise = noise

optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)
```

```
#Включаем режим обучения
model.train()
likelihood.train()

#Градиентный спуск для оптимизации параметров ГП
losses = []
for i in tqdm(range(500)):
    optimizer.zero_grad()

    output = model(train_x)
    loss = -mll(output, train_y)

    loss.backward()
    losses.append(loss.item())

    optimizer.step()

#Включаем режим прогнозирования
model.eval()
likelihood.eval()
```

После этих 500 итераций проверьте масштаб вывода с помощью вывода:

```
>>> model.covar_module.base_kernel.lengthscale
tensor([[1.1535]])
```

Оптимизированный масштаб длины равен примерно 1,15.

7. Переопределите класс модели, установив на этот раз `ard_num_dims = 2` и перезапустив все ячейки кода.

   После этого мы получим следующие значения масштабов длины:

   ```
   >>> model.covar_module.base_kernel.lengthscale
   tensor([[1.6960, 0.8739]])
   ```

   Здесь масштаб длины первого измерения велик (примерно 1,70), а масштаб длины второго измерения мал (примерно 0,87). Это соответствует тому, что целевая функция больше меняется по второму измерению.

П.3. Глава 4: Достижение наилучшего результата с помощью политик, основанных на улучшениях

Для этой главы есть два упражнения:

1. Первое описывает способ совершенствования политики PoI, позволяющий лучше исследовать пространство поиска.
2. Второе применяет две изученные нами политики к моделируемой реальной задаче настройки гиперпараметров.

П.3.1. Упражнение 1: Поощрение исследований с помощью вероятности улучшения

Это упражнение, реализованное в `CH04/02 - Exercise 1.ipynb`, показывает, как изменить политику PoI так, чтобы стимулировать исследование.

Выполните следующие шаги:

1. Воссоздайте цикл БО из `CH04/01 - BayesOpt loop.ipynb`, который использует одномерную функцию Форрестера в качестве цели оптимизации.
2. Перед циклом `for`, реализующим БО, объявите переменную с именем `epsilon`:

```
epsilon = 0.1
```

3. Внутри цикла инициализируйте политику PoI, как и раньше. Но на этот раз укажите, что действующий порог, установленный аргументом `best_f`, представляет собой сумму действующего значения и `epsilon`:

```
policy = botorch.acquisition.analytic.ProbabilityOfImprovement(
    model, best_f=train_y.max() + epsilon
)
```

4. Снова запустите код и посмотрите, приведет ли поощрение исследований к лучшей производительности оптимизации по сравнению с исходной политикой PoI, как показано на рис. П.2.

   Здесь модифицированный PoI нашел оптимум.

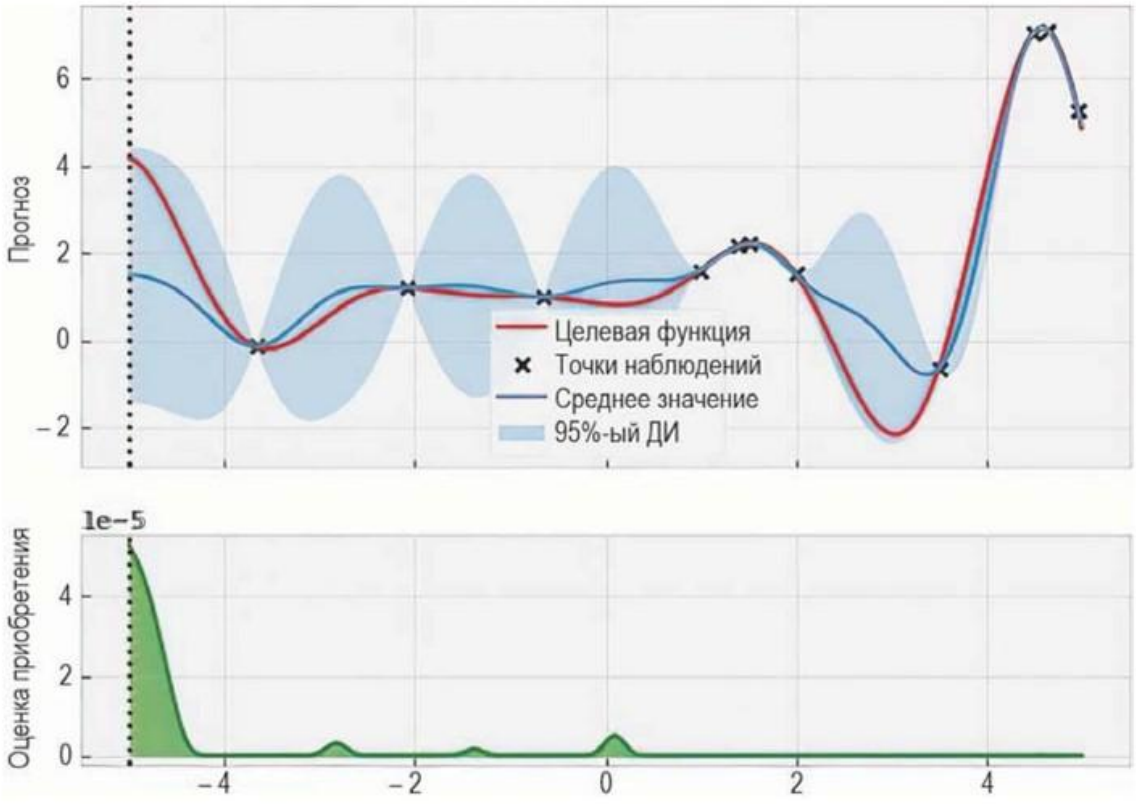


Рис. П.2. Ход оптимизации модифицированной политики PoI на последней итерации: оптимум найден

5. Насколько более «исследовательской» станет PoI, во многом зависит от минимального порога улучшения, хранящегося в `epsilon`. Присвоение этой переменной значения 0,001 недостаточно стимулирует исследования, и политика снова застревает. А при значении 0,5 она работает хорошо.
6. Внедрите относительный минимальный порог с требованием улучшения на 110%:

```
epsilon_pct = 0.1

for i in range(num_queries):
 ...

 #Относительное улучшение
 policy = botorch.acquisition.analytic.ProbabilityOfImprovement(
     model, best_f=train_y.max() * (1 + epsilon_pct)
 )
```

П.3.2. Упражнение 2: БО для настройки гиперпараметров

В этом упражнении (`CH04/03 - Exercise 2.ipynb`) БО применяется к целевой функции, которая моделирует поверхность точности для модели машины опорных векторов в задаче по настройке гиперпараметров. Выполните следующие шаги:

1. Воссоздайте цикл БО из `CH04/01 – BayesOptloop.ipynb`. Целевая функция реализуется как:

```
def f(x):
    return (
        torch.sin(5 * x[..., 0] / 2 - 2.5)
        * torch.cos(2.5 - 5 * x[..., 1])
        + (5 * x[..., 1] / 2 + 0.5) ** 2 / 10
    ) / 5 + 0.2
```

2. Объявите соответствующие тестовые данные с помощью `xs` для двумерной сетки, представляющей домен, и `ys` для значений `xs`:

```
lb = 0
ub = 2
num_queries = 20

bounds = torch.tensor([[lb, lb], [ub, ub]], dtype=torch.float)

xs = torch.linspace(lb, ub, 101)
x1, x2 = torch.meshgrid(xs, xs)
xs = torch.vstack((x1.flatten(), x2.flatten())).transpose(-1, -2)
ys = f(xs)
```

3. Измените вспомогательную функцию `visualize_progress_and_policy()`, которая визуализирует ход оптимизации. Она принимает объект политики, а также `next_x`

в качестве следующей точки для запроса. В первую очередь функция вычисляет оценки приобретения для тестовых данных `xs`:

```
def visualize_progress_and_policy(policy, next_x=None):
    with torch.no_grad():
        acquisition_score = policy(xs.unsqueeze(1))
    ... #Продолжение можно опустить
```

Далее нужно объявить два подграфика Matplotlib. Для первого из них отобразите основную истину, хранящуюся в `ys`:

```
#Тепловая карта, показывающая основную истину
c = ax[0].imshow(
    ys.reshape(101, 101).T, origin="lower", extent=[lb, ub, lb, ub]
)
ax[0].set_xlabel(r"$C$", fontsize=20)
ax[0].set_ylabel(r"$\gamma$", fontsize=20)
plt.colorbar(c, ax=ax[0])

#Разбросанные точки, показывающие помеченные данные
ax[0].scatter(train_x[..., 0], train_x[..., 1], marker="x", c="k")
```

Постройте еще одну тепловую карту на втором подграфике, показывающую оценки приобретения:

```
#Тепловая карта, показывающая оценку приобретения
c = ax[1].imshow(
    acquisition_score.reshape(101, 101).T,
    origin="lower",
    extent=[lb, ub, lb, ub]
)
ax[1].set_xlabel(r"$C$", fontsize=20)
plt.colorbar(c, ax=ax[1])
```

Дополнительно можно показать `next_x`:

```
if next_x is not None:
    ax[1].scatter(
        next_x[..., 0],
        next_x[..., 1],
        c="r",
        marker="*",
        s=500,
        label="next query"
    )
```

4. Скопируйте класс ГП из упражнения к главе 3, который реализует ядро Матерна 2.5 с ARD. Далее измените его, чтобы сделать интегрируемым с BoTorch:

```
#Изменения для интеграции с BoTorch
class GPModel(gpytorch.models.ExactGP,
```

```
botorch.models.gpytorch.GPyTorchModel):
    _num_outputs = 1
    def __init__(self, train_x, train_y, likelihood):
        super().__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ConstantMean()
        #Ядро Матерна 2.5 с ARD
        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.MaternKernel(
                nu=2.5,
                ard_num_dims=2
            )
        )
    def forward(self, x):
        ... #Пропустим часть кода
```

5. Повторно используйте вспомогательную функцию `fit_gp_model()` и цикл `for`, реализующий БО. Скопируйте `fit_gp_model()` и объявите исходный набор данных:

```
train_x = torch.tensor([
    [1., 1.],
])
train_y = f(train_x)
```

Затем объявите цикл БО:

```
num_queries = 20

for i in range(num_queries):
    print("iteration", i)
    print("incumbent", train_x[train_y.argmax()], train_y.max())

    model, likelihood = fit_gp_model(train_x, train_y)

    policy = ... #Заглушка для инициализации политики

    next_x, acq_val = botorch.optim.optimize_acqf(
        policy,
        bounds=bounds,
        q=1,
        #Делаем поиск более исчерпывающим
        num_restarts=40,
        raw_samples=100,
    )

    #Вызов новой вспомогательной функции для визуализации
    visualize_progress_and_policy(policy,next_x=next_x)

    next_y = f(next_x)
    train_x = torch.cat([train_x, next_x])
    train_y = torch.cat([train_y, next_y])
```

6. Запустите политику PoI для целевой функции. Обратите внимание, что она снова застревает на локальном оптимуме. Замените строку, инициализирующую политику, на другую:

```
policy = botorch.acquisition.analytic.ProbabilityOfImprovement(
    model, best_f=train_y.max()
)
```

Анализ всего кода показывает, что политика снова застревает на локальном оптимуме, как показано на рис. П.3.

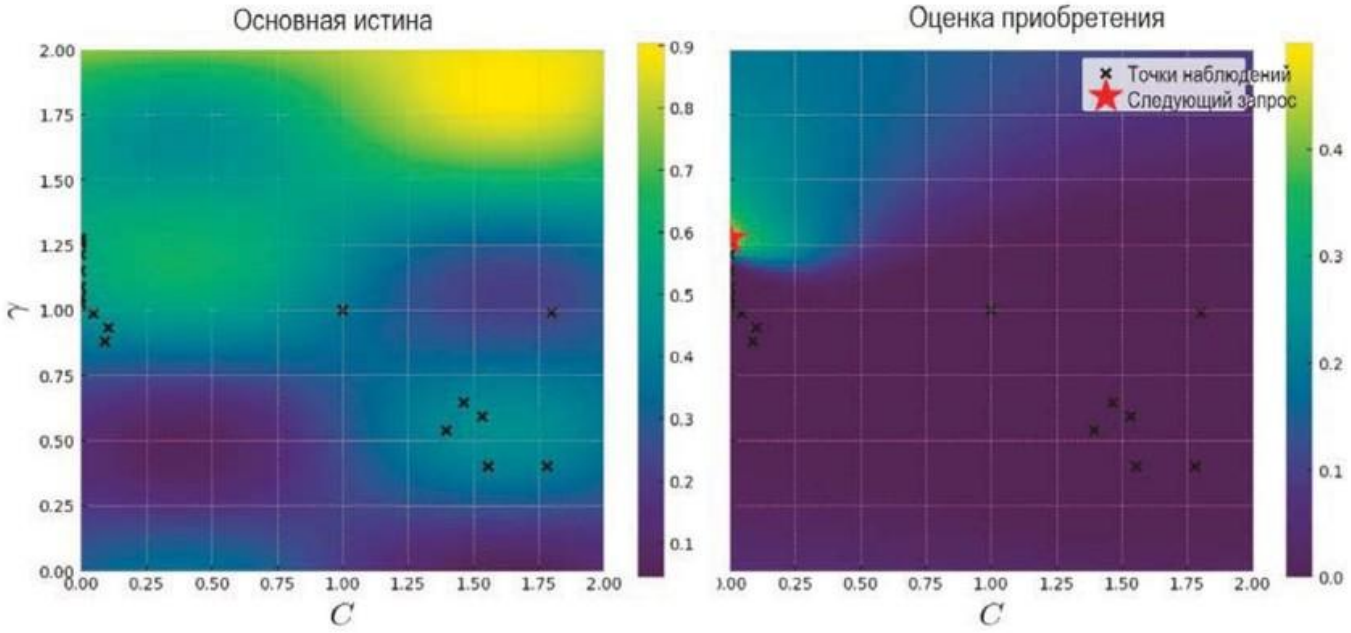


Рис. П.3. Ход оптимизации политики PoI на последней итерации: политика застряла на локальном оптимуме

7. Запустите модифицированную версию PoI, где минимальный порог улучшения установлен на 0,1. Замените строку, инициализирующую политику, на другую:

```
policy = botorch.acquisition.analytic.ProbabilityOfImprovement(
    model, best_f=train_y.max() + 0.1
)
```

Эта политика является более исследовательской и превосходит обычную PoI. На рис. П.4 показан ее прогресс на 17-й итерации, где она впервые достигает точности не менее 90%.

Параметры, дающие такую точность, следующие: $C = 1{,}6770$ и $\gamma = 1{,}9039$.

8. Запустите политику ожидаемого улучшения (EI) для целевой функции. Для этого замените строку, инициализирующую политику, на другую:

```
policy = botorch.acquisition.analytic.ExpectedImprovement(
    model, best_f=train_y.max()
)
```

Эта политика хорошо работает с целевой функцией, обеспечивая точность не менее 90% на 15-й итерации, как показано на рис. П.5.

Параметры, дающие такую точность, следующие: $C = 1{,}6331$ и $\gamma = 1{,}8749$.

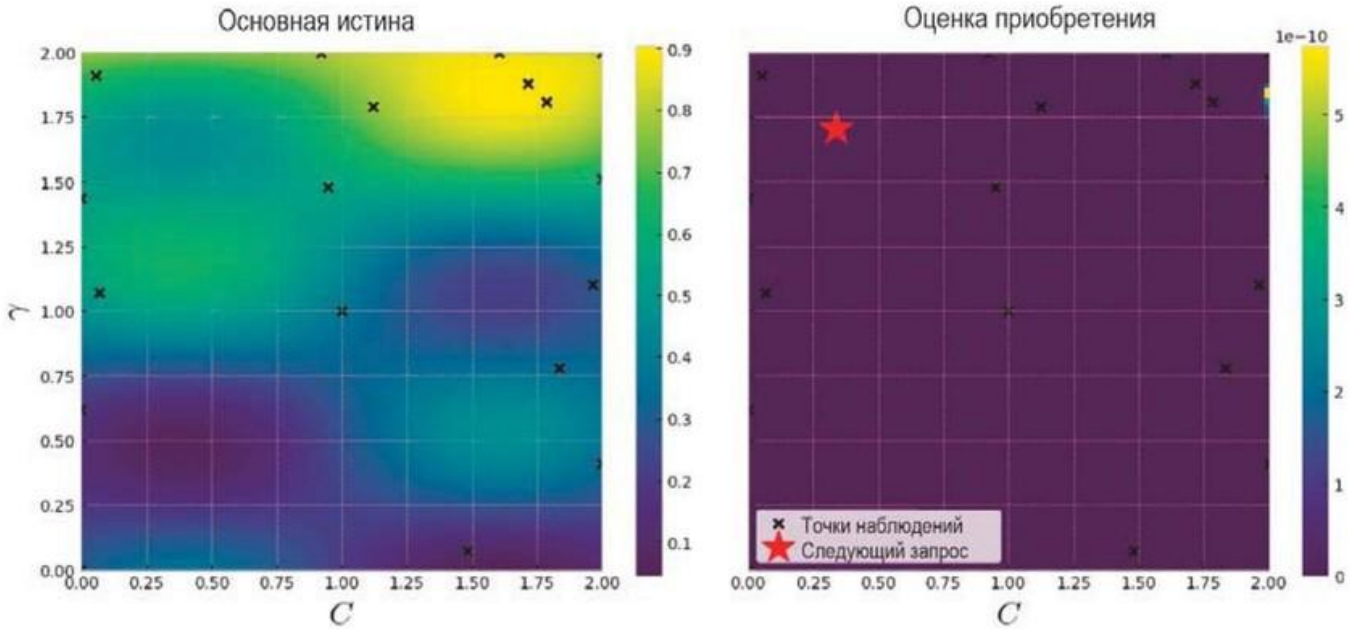


Рис. П.4. Ход оптимизации модифицированной политики PoI на 17-й итерации, где политика впервые достигает точности не менее 90%

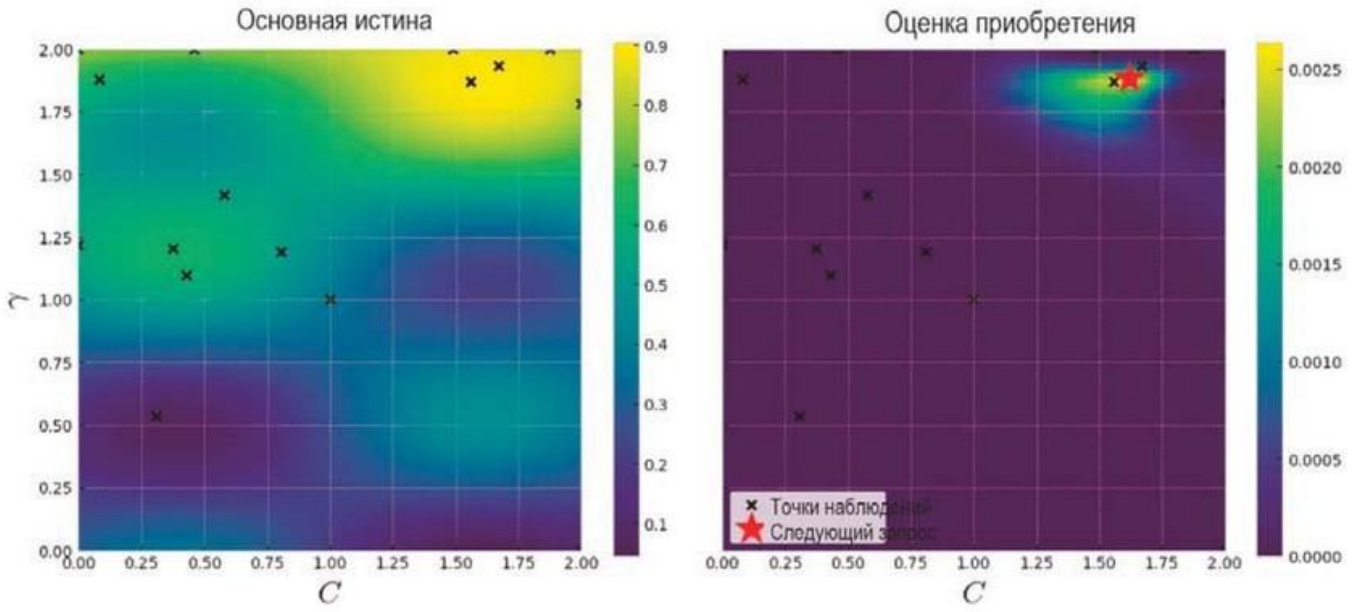


Рис. П.5. Ход оптимизации политики EI на 15-й итерации, где она впервые достигает точности не менее 90%

9. Реализуйте повторяющиеся эксперименты и визуализируйте средние действующие значения и полосы ошибок по 10 экспериментам. Сначала поместите код для цикла БО во внешний цикл, который повторяет несколько экспериментов. Сохраните наиболее заметное значение на каждом этапе экспериментов в `incumbents`:

```
num_repeats = 10

incumbents = torch.zeros((num_repeats, num_queries))

for trial in range(num_repeats):
    print("trial", trial)
```

```
    #Равномерно выбираем точку данных в пространстве поиска
    #в качестве отправной точки
    torch.manual_seed(trial)
    train_x = bounds[0] + (bounds[1] - bounds[0]) * torch.rand(1, 2)
    train_y = f(train_x)

    #Отслеживаем наиболее заметное значение
    for i in tqdm(range(num_queries)):
        incumbents[trial, i] = train_y.max()

        ... #Пропущенный код такой же, как и раньше

#Сохраняем результаты в файл, чтобы можно было визуализировать их позже
torch.save(incumbents, [path to file])
```

Затем реализуйте вспомогательную функцию, которая отображает средние действующие значения и полосы ошибок. Она считывает тензор PyTorch из переменной `path`, которая является сохраненной версией `incumbents` на предыдущем шаге:

```
def show_agg_progress(path, name):
    def ci(y):
        #Вспомогательная подфункция для вычисления планок ошибок
        return 2 * y.std(axis=0) / np.sqrt(num_repeats)

    #Загружаем сохраненные результаты оптимизации
    incumbents = torch.load(path)

    #Вычисляем результаты среднего и полосы ошибок
    avg_incumbent = incumbents.mean(axis=0)
    ci_incumbent = ci(incumbents)

    #Визуализируем результаты среднего и полосы ошибок
    plt.plot(avg_incumbent, label=name)
    plt.fill_between(
        np.arange(num_queries),
        avg_incumbent + ci_incumbent,
        avg_incumbent - ci_incumbent,
        alpha=0.3,
    )
```

Затем можно запустить предыдущие политики и сравнить их производительность:

```
plt.figure(figsize=(8, 6))

show_agg_progress([path to EI data], "EI")
show_agg_progress([path to PoI data], "PoI")
show_agg_progress([path to modified PoI data], "PoI"
  + r"$(\epsilon = 0.1)$")
```

```
plt.xlabel("# queries")
plt.ylabel("accuracy")

plt.legend()

plt.show()
```

В результате получается рис. П.6, на котором показана эффективность оптимизации PoI, модифицированной версии PoI и EI.

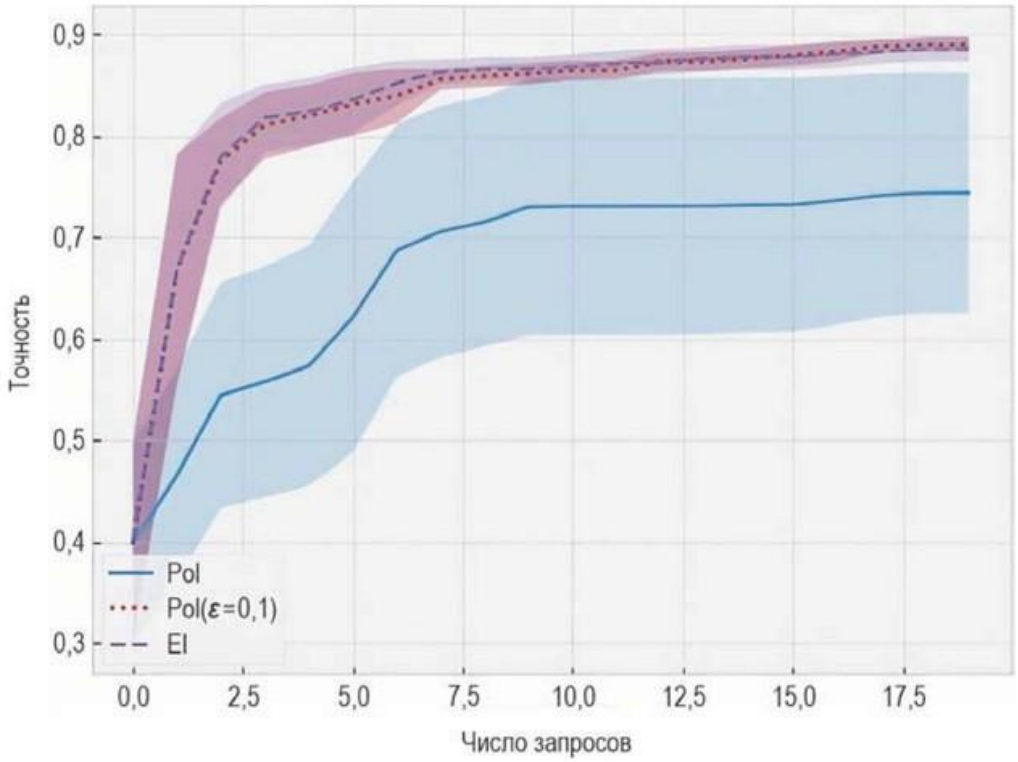


Рис. П.6. Прогресс оптимизации различных политик, обобщенный по 10 повторным экспериментам

Рис. П.6 дает больше информации за один проход, чем проверка каждой политики отдельно. Здесь PoI не только работает хуже, чем две другие политики, но ее производительность также менее надежна, как видно на больших полосах ошибок. Модифицированные PoI и EI работают примерно одинаково. Трудно сказать, какая из них лучше, поскольку их полосы ошибок перекрываются.

П.4. Глава 5: Исследование пространства поиска с помощью политик «многорукого бандита»

Данная глава содержит два упражнения:

1. В первом упражнении исследуется потенциальный метод, как установить компромиссный параметр для политики UCB, который учитывает прогресс продвижения в оптимизации.
2. Во втором упражнении две политики (изученные в главе 5) применяются к задаче настройки гиперпараметров, рассмотренной в предыдущих главах.

П.4.1. Упражнение 1: Настройка плана исследования для верхней доверительной границы

В этом упражнении (`CH05/02 - Exercise 1.ipynb`) мы обсуждаем стратегию адаптивной установки значения компромиссного параметра β для политики UCB. Выполните следующие шаги:

1. Воссоздайте цикл БО из `CH04/02 - Exercise 1.ipynb`, который использует одномерную функцию Форрестера в качестве цели оптимизации.

   Поскольку в цикле имеется 10 итераций, β умножается на m 10 раз, чтобы постепенно перейти от значения 1 к значению 10. То есть $1 \times m^{10} = 10$. Решение этого уравнения даст код множителя (`multiplier`):

```
num_queries = 10

start_beta = 1
end_beta = 10

multiplier = (end_beta / start_beta) ** (1 / num_queries)
```

2. Реализуйте эту логику и понаблюдайте за полученной производительностью оптимизации.

   Цикл БО оптимизируется следующим образом:

```
num_queries = 10

start_beta = 1
end_beta = 10

multiplier = (end_beta / start_beta) ** (1 / num_queries)

beta = start_beta

for i in range(num_queries):
    ... #Получаем обученный ГП

    policy = botorch.acquisition.analytic.UpperConfidenceBound(
        model, beta=beta
    )

    ... #Находим точку, которая максимизирует оценку приобретения,
        #запрашивает целевую функцию и обновляет данные обучения

    beta *= multiplier
```

   Этот код создает рис. П.7.

На рисунке видно, что политика проверяет локальный оптимум на пятой итерации. Несмотря на то что она может отдалиться от глобального оптимума, политика приближается к нему в конце.

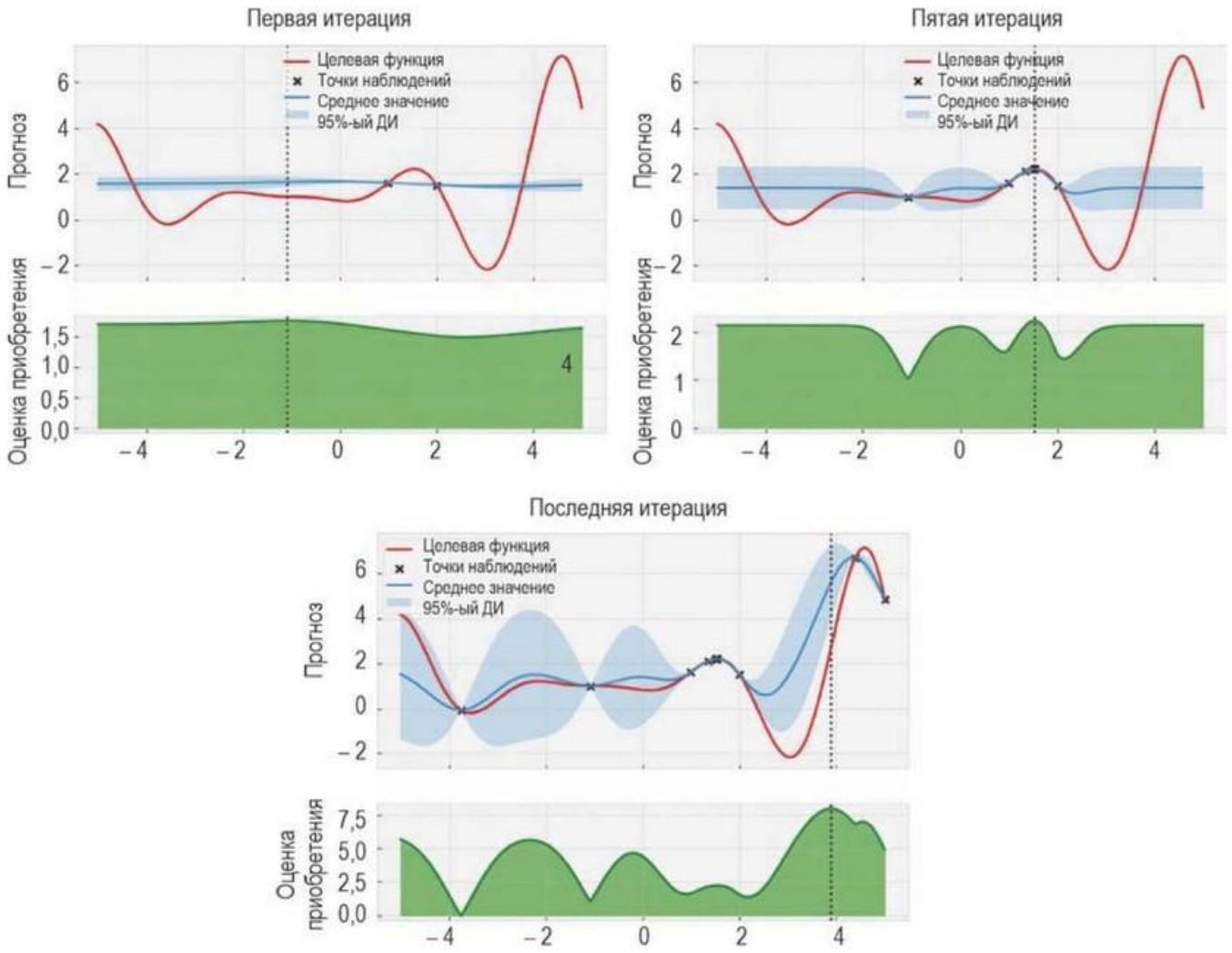


Рис. П.7. Прогресс адаптивной версии UCB: политика способна выйти за рамки локального оптимума и приблизиться к глобальному оптимуму

П.4.2. Упражнение 2: БО для настройки гиперпараметров

В этом упражнении (`CH05/03 - Exercise 2.ipynb`) мы применяем БО к целевой функции, которая имитирует поверхность точности для модели машины опорных векторов в задаче по настройке гиперпараметров. Выполните следующие шаги:

1. Воссоздайте цикл БО из `CH04/03 - Exercise 2.ipynb`, включая внешний цикл, реализующий повторяющиеся эксперименты.
2. Запустите политику UCB, установив значение компромиссного параметра равным $\beta \in \{ 1, 3, 10, 30 \}$, и понаблюдайте за совокупной производительностью значений.

   Компромиссный параметр можно установить при инициализации объекта политики:

```
policy = botorch.acquisition.analytic.UpperConfidenceBound(
    model, beta=[some value]
)
```

На рис. П.8 показаны результаты оптимизации четырех версий политики UCB. При $\beta = 1$ политика становится слишком «эксплуататорской» и дает наихудшие результаты.

По мере того как увеличивается значение компромиссного параметра, увеличивается и производительность. Но при $\beta = 30$ чрезмерное исследование приводит к тому, что UCB медленнее определяет местоположение с 90%-ной точностью. В целом при $\beta = 10$ обеспечивается наилучшая производительность.

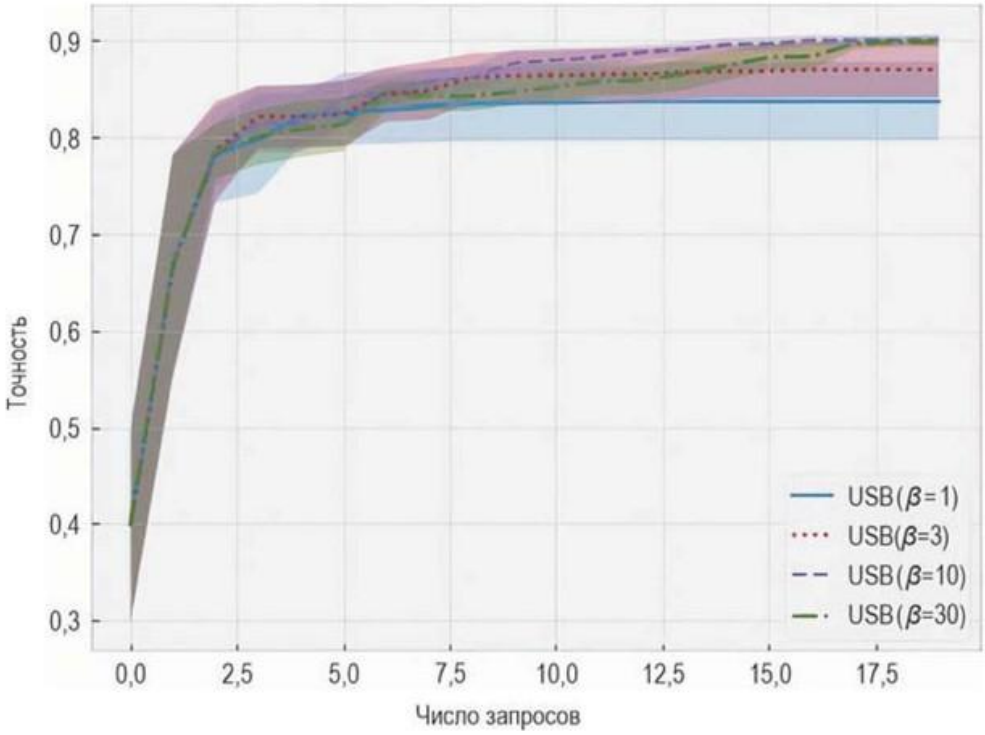


Рис. П.8. Прогресс, достигнутый различными политиками UCB

3. Запустите адаптивную версию UCB (упражнение 1).

   Модифицируйте цикл БО следующим образом:

```
num_repeats = 10

start_beta = 3
end_beta = 10

multiplier = (end_beta / start_beta) ** (1 / num_queries)

incumbents = torch.zeros((num_repeats, num_queries))

for trial in range(num_repeats):
    ... #Случайным образом генерируем начальные данные обучения

    beta = start_beta
```

```
for i in tqdm(range(num_queries)):
    ... #Записывает действующее значение и переобучает модель

    policy = botorch.acquisition.analytic.UpperConfidenceBound(
        model, beta=beta
    )

    ... #Находим точку, которая максимизирует оценку приобретения,
        #запрашивает целевую функцию и обновляет данные обучения

    beta *= multiplier
```

На рис. П.9 показаны результаты оптимизации двух адаптивных версий по сравнению с наиболее эффективным фиксированным значением $\beta = 10$.

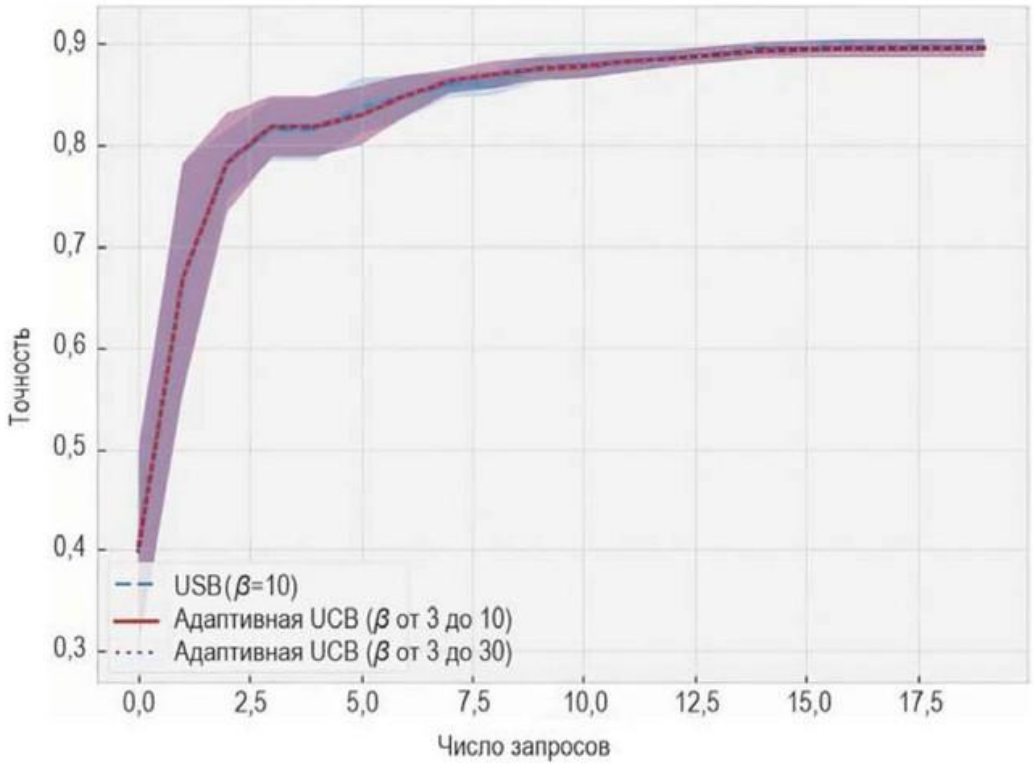


Рис. П.9. Прогресс двух адаптивных версий UCB: данная политика устойчива к конечному значению компромиссного параметра

Эти версии сопоставимы, и изменение конечного значения β с 10 на 30 не сильно влияет на производительность оптимизации.

4. Запустите политику выборки Томпсона (TS) и понаблюдайте за ее совокупной производительностью.

   Реализуйте TS следующим образом:

```
num_candidates = 2000
num_repeats = 10

incumbents = torch.zeros((num_repeats, num_queries))
```

```
for trial in range(num_repeats):
    ... #Случайным образом генерируем начальные обучающие данные

    for i in tqdm(range(num_queries)):
        ... #Записываем действующее значение и переобучаем модель

        sobol = torch.quasirandom.SobolEngine(1, scramble=True)
        candidate_x = sobol.draw(num_candidates)
        candidate_x = bounds[0] + (bounds[1] - bounds[0]) * candidate_x
        ts = botorch.generation.MaxPosteriorSampling(model,
          replacement=False)
        next_x = ts(candidate_x, num_samples=1)
        ... #Находим точку, которая максимизирует оценку приобретения,
            #запрашивает целевую функцию и обновляет данные обучения.
```

На рис. П.10 показаны результаты оптимизации. Политика вначале демонстрирует значительный прогресс. Она сравнима с EI из главы 6 и немного уступает лучшей версии политики UCB.

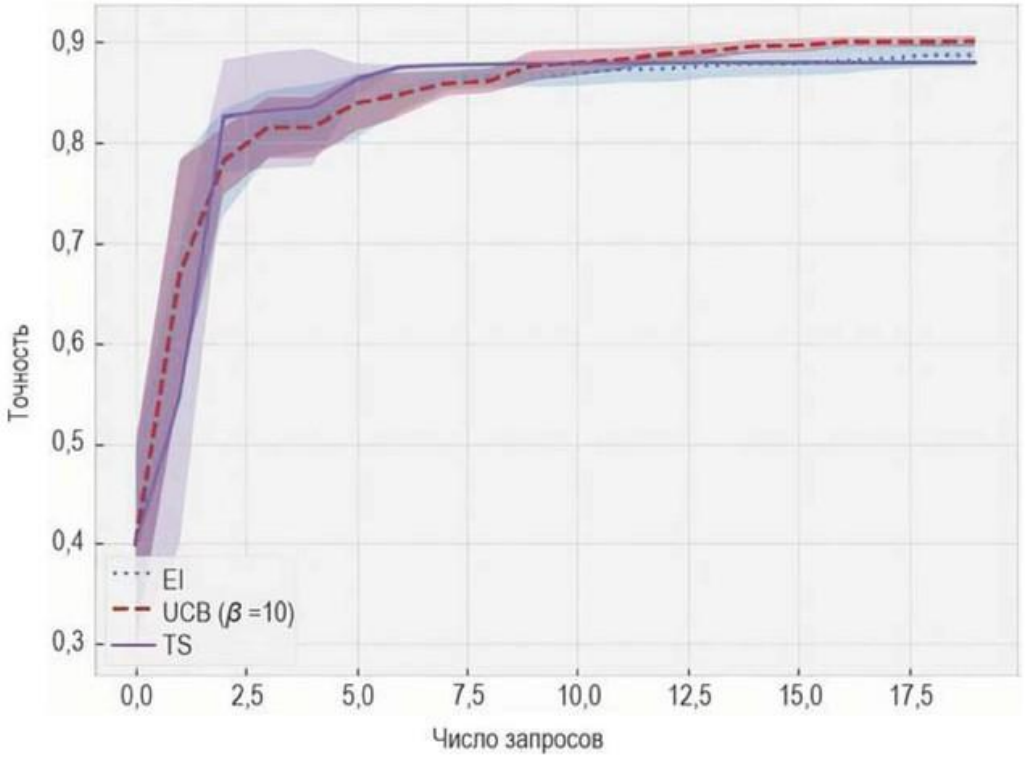


Рис. П.10. Прогресс, достигнутый TS: политика сравнима с EI и немного уступает лучшей версии UCB

П.5. Глава 6: Использование теории информации с политиками, основанными на энтропии

Для этой главы есть два упражнения:

1. Первое охватывает вариант бинарного поиска, в котором может быть учтена априорная информация при принятии решений.

2. Второе касается поиска энтропии с максимальным значением (MES) в задаче по настройке гиперпараметров, рассмотренной в предыдущих главах.

П.5.1. Упражнение 1: Включение предварительных знаний в энтропийный поиск

Это упражнение (`CH06/02 - Exercise 1.ipynb`) демонстрирует использование различных предварительных (априорных) знаний при поиске теоретически оптимального решения и помогает еще больше оценить элегантность и гибкость энтропийного поиска с точки зрения универсального процесса принятия решений в условиях неопределенности. Выполните следующие шаги:

1. Докажите, что $Pr(X = 1) + Pr(X = 2) + ... + Pr(X = 10) = 1$.

   Это можно сделать, просто сложив вероятности:

   $$1 / 2 + 1 / 4 + ... + 1 / 2^9 + 1 / 2^9 = 1 / 2 + 1 / 4 + ... + 1 / 28 + 1 / 28 = ... = \\ = 1 / 2 + 1 / 2 = 1.$$

2. Вычислите энтропию этого предварительного распределения.

   Помните, что формула энтропии следующая: $-\Sigma_i\ p_i\ log(p_i)$. Напишите функцию Python, которая вычисляет эту сумму:

```
def compute_entropy(first, last):
    entropy = 0
    for i in range(first, last + 1):
        #Получаем текущую вероятность
        p = marginal_probability(i, first, last)
        #Сумма согласно условиям
        entropy += -p * np.log2(p)

    return entropy
```

   Функция принимает в качестве параметров `first` и `last`, которые соответствуют наименьшему и наибольшему числу X (которые начинаются с 1 и 10) соответственно. Затем переберите числа между `first` и `last` и сложите члены уравнения $-p_i\ log(p_i)$. Здесь `marginal_probability()` — это вспомогательная функция, вычисляющая $Pr(X = n)$, которая реализуется как:

```
def marginal_probability(floor, first, last):
    #Крайний случай, когда этаж является последним
    if floor == last:
        return 2 ** -(last - first)
    return 2 ** -(floor - first + 1)
```

   Выполнение `compute_entropy(1, 10)` дает результат 1,99609375 — это энтропия априорного распределения X.

3. Учитывая априорное распределение (между 1 и 10), какова вероятность, что телефон сломается при падении со второго этажа? Какова эта вероятность для пятого этажа? Какова вероятность для первого этажа?

Вероятность, что телефон сломается при падении со второго этажа, составляет $Pr(X = 1)$, что равно 0,5.

Вероятность, что телефон сломается при падении с пятого этажа, равна вероятности, что $X \leq 4$. Это составляет

$$Pr(X = 1) + Pr(X = 2) + Pr(X = 3) + Pr(X = 4) = 15 /16 = 0{,}9375.$$

Эти два расчета можно реализовать как функцию:

```
#Суммируем все вероятности для X, которые меньше порогового значения
def cumulative_density(floor, first, last):
    return sum(
        [
            marginal_probability(i, first, last)
            for i in range(first, floor)
        ]
    )
```

Поскольку предварительные знания показывают, что телефон не сломается при падении с первого этажа, эта вероятность равна 0.

4. Вычислите энтропию фиктивного апостериорного распределения при падении с пятого этажа.

   Используя уже реализованную функцию `compute_entropy()`, можно вычислить энтропию для двух возможных сценариев. Если телефон сломается при падении с пятого этажа, устанавливаем `first = 1` и `last = 4`, в противном случае — `first = 5` и `last = 10`:

```
>>> compute_entropy(1, 4)
Output: 1.75
>>> compute_entropy(5, 10)
Output: 1.9375
```

5. Учитывая априорное распределение, вычислите ожидаемую апостериорную энтропию после проведения испытания на пятом этаже.

   Необходимые расчеты для данного ожидаемого вычисления апостериорной энтропии уже выполнены. Во-первых, вероятность, что телефон сломается при падении с пятого этажа, равна 0,9375. В этом случае апостериорная энтропия равна 1,75. Во-вторых, вероятность, что телефон не сломается, упав с пятого этажа, равна 1 – 0,9375 = 0,0625. В этом случае апостериорная энтропия равна 1,9375.

   Если взять среднее значение двух случаев, получится (0,9375)*1,75 + (0,0625)* *1,9375 = 1,76171875 — это ожидаемая апостериорная энтропия после проведения испытания на пятом этаже.

6. Вычислите ожидаемую апостериорную энтропию для других этажей.

   Для этого реализуйте функцию, которая выполняет только что описанные вычисления:

```
def compute_expected_posterior_entropy(floor, first, last):
    #Вероятность, что телефон сломается, упав с заданного этажа
    break_probability = cumulative_density(floor, first, last)

    #Берем среднее значение из двух случаев
    return (
        break_probability * compute_entropy(first, floor - 1)
          + (1 - break_probability) * compute_entropy(floor, last)
    )
```

Используя данную функцию, можно построить график ожидаемой апостериорной энтропии для чисел между 1 и 10.

График показан на рис. П.11. Он демонстрирует, что теоретически оптимальным местом для первого испытания является второй этаж, поскольку значение 2 дает наименьшую ожидаемую апостериорную энтропию (и, следовательно, неопределенность).

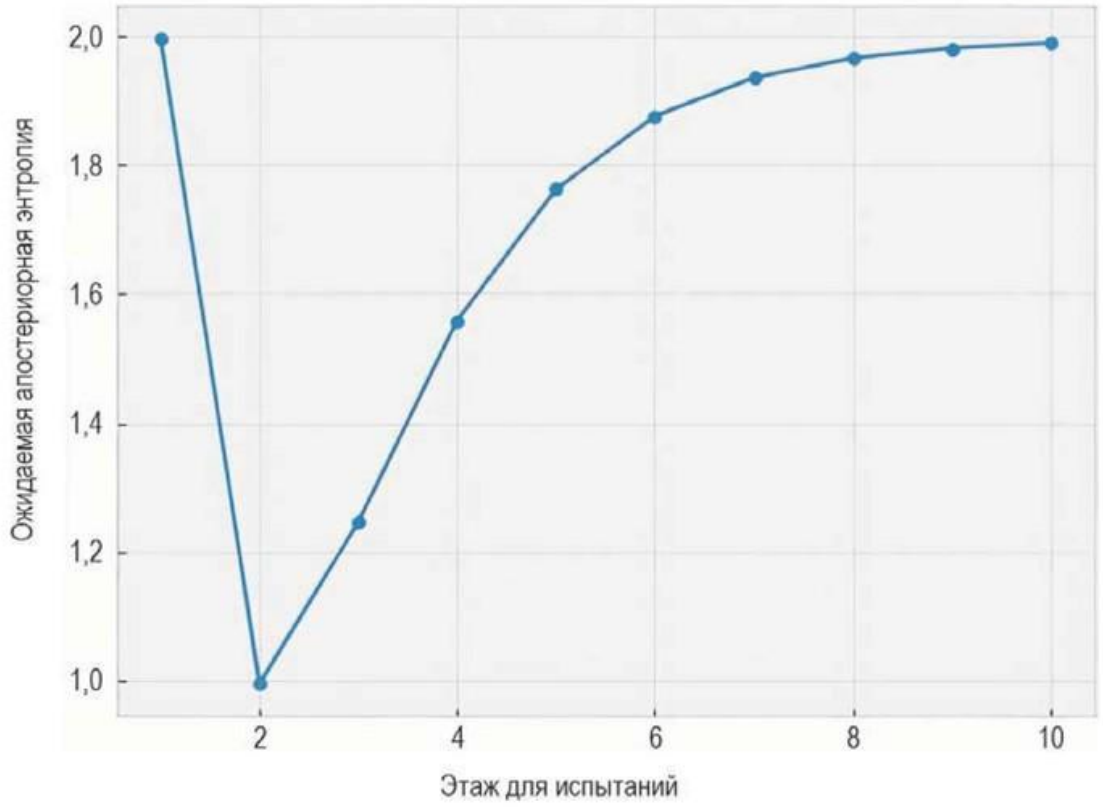


Рис. П.11. Ожидаемая апостериорная энтропия в зависимости от этажа, где проводится испытание

На рисунке видно совсем не то, что предполагает бинарный поиск решений (5). Это прямой эффект от знаний области поиска, которые мы переводим в код, используя априорное распределение для X. Поскольку существует высокая вероятность, что $X = 2$ (50%), лучше сначала попробовать именно это значение, чтобы сразу найти ответ, если телефон сломается.

Интересно, что падение телефона с первого этажа не приводит к уменьшению энтропии. Это потому, что мы точно знаем: телефон не сломается при падении с этого этажа. Поэтому знания после проведения этого испытания не изменятся.

П.5.2. Упражнение 2: БО для настройки гиперпараметров

В этом упражнении (`CH06/03 - Exercise 2.ipynb`) БО применяется к целевой функции, которая моделирует поверхность точности для модели машины опорных векторов в задаче по настройке гиперпараметров. Выполните следующие шаги:

1. Воссоздайте цикл БО из `CH04/03 - Exercise 2.ipynb`, включая внешний цикл, реализующий повторяющиеся эксперименты.
2. Запустите политику MES.

   Поскольку целевая функция является двумерной, нужно установить размер последовательности Соболя равным 2000:

```
num_candidates = 2000
num_repeats = 10

incumbents = torch.zeros((num_repeats, num_queries))

for trial in range(num_repeats):
  ... #Случайным образом генерируем начальные обучающие данные

  for i in tqdm(range(num_queries)):
    ... #Записываем действующее значение и переобучаем модель

    sobol = torch.quasirandom.SobolEngine(1, scramble=True)
    candidate_x = sobol.draw(num_candidates)
    candidate_x = bounds[0] + (bounds[1] - bounds[0]) * candidate_x
```

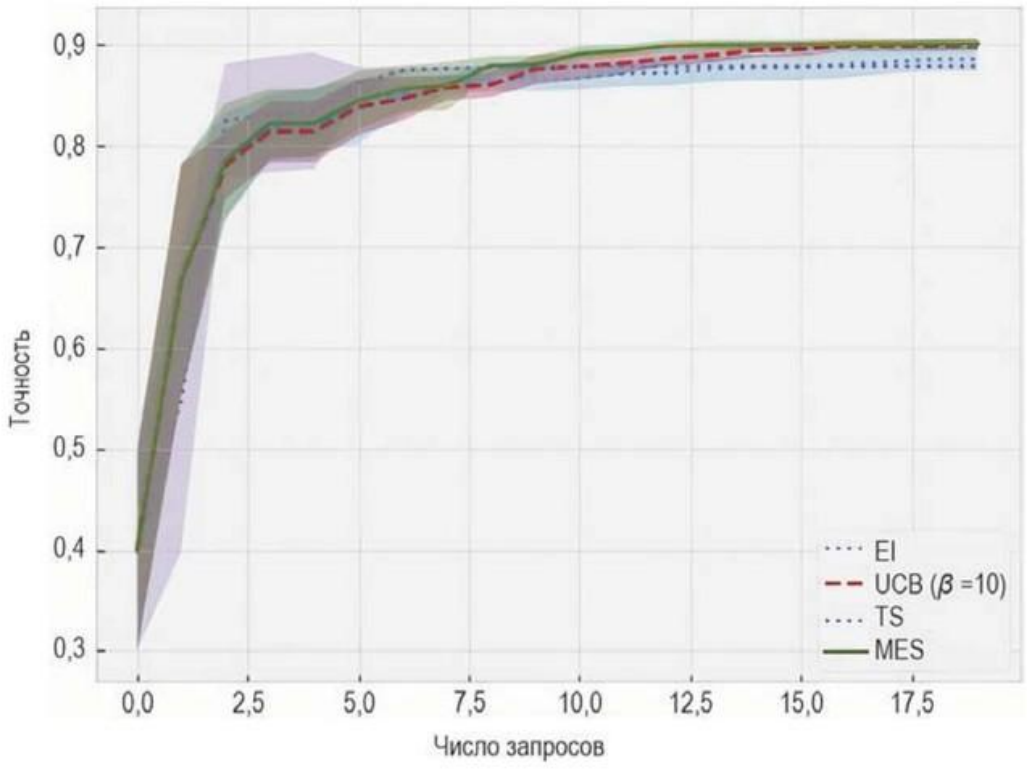


Рис. П.12. Прогресс MES: политика обеспечивает наилучшие результаты из четырех показанных политик

```
policy = botorch.acquisition.max_value_entropy_search.qMaxValueEntropy(
    model, candidate_x
)

... #Находим точку, которая максимизирует оценку приобретения,
    #запрашивает целевую функцию и обновляет данные обучения
```

На рис. П.12 показаны результаты оптимизации MES. Как видно, она является конкурентоспособной по сравнению со всеми политиками БО, которые были изучены к текущему моменту.

П.6. Глава 7: Увеличение пропускной способности с помощью пакетной оптимизации

Для этой главы есть два упражнения:

1. Первое касается реализации TS в пакетном сценарии.
2. Второе демонстрирует, как использовать политики БО для решения четырехмерной задачи оптимизации проектирования авиационных конструкций.

П.6.1. Упражнение 1: Расширение TS для пакетного режима с помощью повторной выборки

Политика TS при последовательной настройке (раздел 5.3) извлекает одну выборку из текущего предположения ГП о целевой функции и запрашивает точку данных, которая максимизирует ее. При пакетном режиме процесс выборки из ГП и ее максимизации повторяется несколько раз, пока не соберется пакет запросов желаемого размера. Код для этого упражнения можно найти в `CH07/02 - Exercise 1.ipynb`:

1. Воссоздайте пакетный цикл БО из `CH05/01 - BayesOpt loop.ipynb`.
2. Реализуйте TS с помощью генератора последовательности Соболя, как описано в разделе 5.3.

   Реализуйте политику, используя последовательность Соболя из 2000 элементов и указав количество выборок в качестве размера пакета:

```
#Определяем длину последовательности Соболя
num_candidates = 2000

... #Случайным образом выбираем начальные обучающие данные

for i in tqdm(range(num_iters)):
    ... #Переобучаем ГП

    #Инициализируем последовательность Соболя
    sobol = torch.quasirandom.SobolEngine(2, scramble=True)
    candidate_x = sobol.draw(num_candidates)
    candidate_x = (bounds[1] - bounds[0]) * candidate_x + bounds[0]
```

```
#Делаем несколько выборок из ГП
ts = botorch.generation.MaxPosteriorSampling(model, replacement=False)
next_x = ts(candidate_x, num_samples=batch_size)

... #Запрашиваем целевую функцию и обновляем данные обучения
```

3. Запустите политику для настройки гиперпараметров целевой функции и понаблюдайте за ее производительностью.

   После запуска пакетного варианта можно построить график прогресса и сравнить с другими изученными политиками. Это показано на рис. П.13. Здесь TS может добиться значительного прогресса уже после первого пакета запросов.

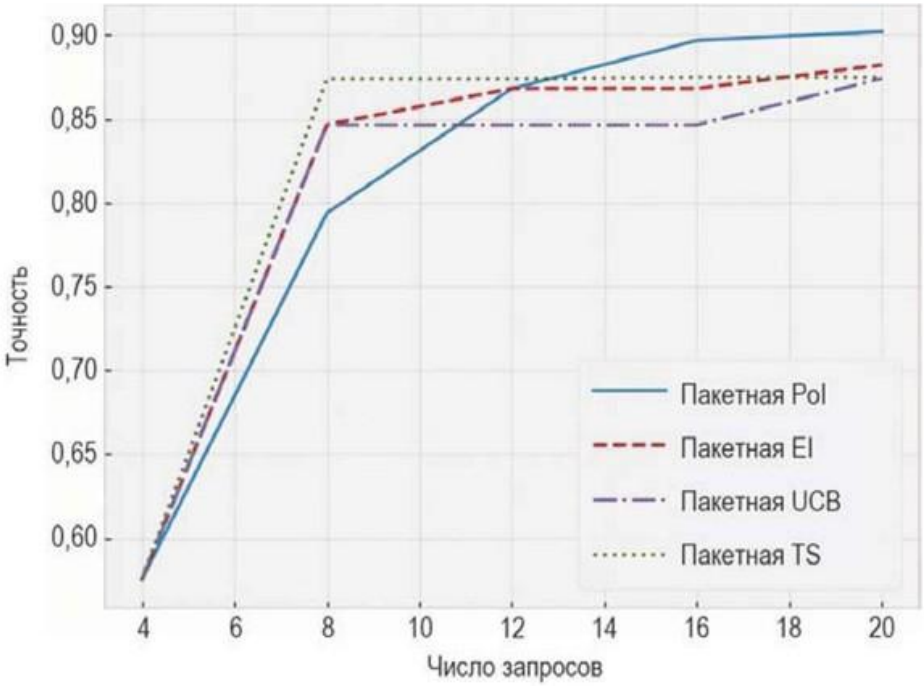


Рис. П.13. Прогресс пакетной TS в задаче по настройке гиперпараметров: политика достигает значительного прогресса уже после первого пакета запросов

П.6.2. Упражнение 2: Оптимизация конструкции самолета

В этом упражнении рассматривается целевая функция, моделирующая процесс сравнительного анализа характеристик в конструкции самолета. Код представлен в `CH07/04 - Exercise 2.ipynb`. Выполните следующие шаги:

1. Реализуйте целевую функцию, которая имитирует замер производительности.

   Код целевой функции уже есть, поэтому просто скопируйте и вставьте его в программу:

```
def flight_utility(X):
    X_copy = X.detach().clone()
    X_copy[:, [2, 3]] = 1 - X_copy[:, [2, 3]]
    X_copy = X_copy * 10 - 5

    return -0.005*(X_copy**4-16*X_copy**2+5*X_copy).sum(dim=-1)+3
```

2. Создайте модель с постоянной функцией среднего и ядром Матерна 2.5 с масштабом вывода, которая реализована как объект `gpytorch.kernels.ScaleKernel`.

   Реализация класса этой модели в основном сохраняется, как и раньше, за исключением того, что нужно указать правильное количество измерений в ARD:

```
class GPModel(
    gpytorch.models.ExactGP,
    botorch.models.gpytorch.GPyTorchModel,
    botorch.models.model.FantasizeMixin
):
    _num_outputs = 1
    def __init__(self, train_x, train_y, likelihood):
        super().__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(
            #Ядро Матерна 2.5 для четырех измерений
            gpytorch.kernels.MaternKernel(
                nu=2.5,
                ard_num_dims=4
            )
        )
    def forward(self, x):
        ...
```

3. Реализуйте вспомогательную функцию, которая обучает ГП на заданном наборе обучающих данных.

   Для этого можно просто скопировать ту же вспомогательную функцию `fit_gp_model()` из других задач к этой главе, поскольку нет необходимости в ней что-либо менять.

4. Определите параметры задачи оптимизации.

   Для этого сначала определите границы пространства поиска:

```
lb = 0
ub = 1

bounds = torch.tensor([[lb] * 4, [ub] * 4], dtype=torch.float)
```

   Затем укажите, сколько запросов можно сделать, размер пакета и количество повторных экспериментов для каждой политики:

```
num_experiments = 5

num_queries = 100
batch_size = 5
num_iters = num_queries // batch_size
```

5. Запустите каждую пакетную политику в этой главе для целевой функции, реализованной ранее.

Для этого сначала используйте код ниже, чтобы реализовать цикл оптимизации и внешний цикл, который позволяет повторно выполнять эксперименты для каждой политики. В частности, для каждого отдельного эксперимента мы случайным образом выбираем одну точку данных внутри пространства поиска, а затем для нее запускаем каждую политику, пока не закончатся запросы. Как определяется каждая политика, рассмотрим на следующем шаге:

```
incumbents = torch.zeros((num_experiments, num_iters))

pbar = tqdm(total=num_experiments * num_iters)

#Случайным образом инициализируем обучающие данные
for exp in range(num_experiments):
    torch.manual_seed(exp)
    train_x = bounds[0] + (bounds[1] - bounds[0]) * torch.rand(1, 4)
    train_y = flight_utility(train_x)

    for i in range(num_iters):
        #Отслеживаем ход оптимизации и обновляем прогнозную модель
        incumbents[exp, i] = train_y.max()
        model, likelihood = fit_gp_model(train_x, train_y)

        ... #Определяем политику и находим следующий пакет для запроса

        #Запрашиваем точки, рекомендованные политикой,
        #и обновляем данные обучения
        next_y = flight_utility(next_x)
        train_x = torch.cat([train_x, next_x])
        train_y = torch.cat([train_y, next_y])

        pbar.update()
```

Для политики PoI используйте следующий код:

```
policy = botorch.acquisition.monte_carlo.qProbabilityOfImprovement(
    model, best_f=train_y.max()
)
```

Для политики EI используйте следующий код:

```
policy = botorch.acquisition.monte_carlo.qExpectedImprovement(
    model, best_f=train_y.max()
)
```

Для политики UCB используйте следующий код:

```
policy = botorch.acquisition.monte_carlo.qUpperConfidenceBound(
    model, beta=2
)
```

Эти три политики затем можно оптимизировать с помощью вспомогательной функции `optimize_acqf()` следующим образом:

```
next_x, acq_val = botorch.optim.optimize_acqf(
    policy,
    bounds=bounds,
    q=batch_size,
    num_restarts=100,
    raw_samples=200,
)
```

В противном случае, как для TS, так и для MES, сначала нужно определить последовательность Соболя:

```
#Указываем количество измерений, равное 4
sobol = torch.quasirandom.SobolEngine(4, scramble=True)

candidate_x = sobol.draw(5000)
candidate_x = (bounds[1] - bounds[0]) * candidate_x + bounds[0]
```

Для политики TS используйте следующий код:

```
ts = botorch.generation.MaxPosteriorSampling(model, replacement=False)
next_x = ts(candidate_x, num_samples=batch_size)
```

Для MES используйте код, который использует вспомогательную функцию `optimize_acqf_cyclic()` для реализации циклической оптимизации. Обратите внимание, мы указываем, что она должна занимать не более 5 итераций:

```
policy = botorch.acquisition.max_value_entropy_search.qMaxValueEntropy(
    model, candidate_x
)

next_x, acq_val = botorch.optim.optimize_acqf_cyclic(
    policy,
    bounds=bounds,
    q=batch_size,
    num_restarts=40,
    raw_samples=100,
    #Указываем максимальное количество итераций для циклической оптимизации
    cyclic_options={"maxiter": 5}
)
```

6. Постройте график хода оптимизации задействованных политик и понаблюдайте за их эффективностью.

   На рис. П.14 показаны результаты оптимизации, полученные с помощью реализованных политик. Большинство из них сопоставимы друг с другом, за исключением TS, а пакетная PoI имеет даже небольшое преимущество.

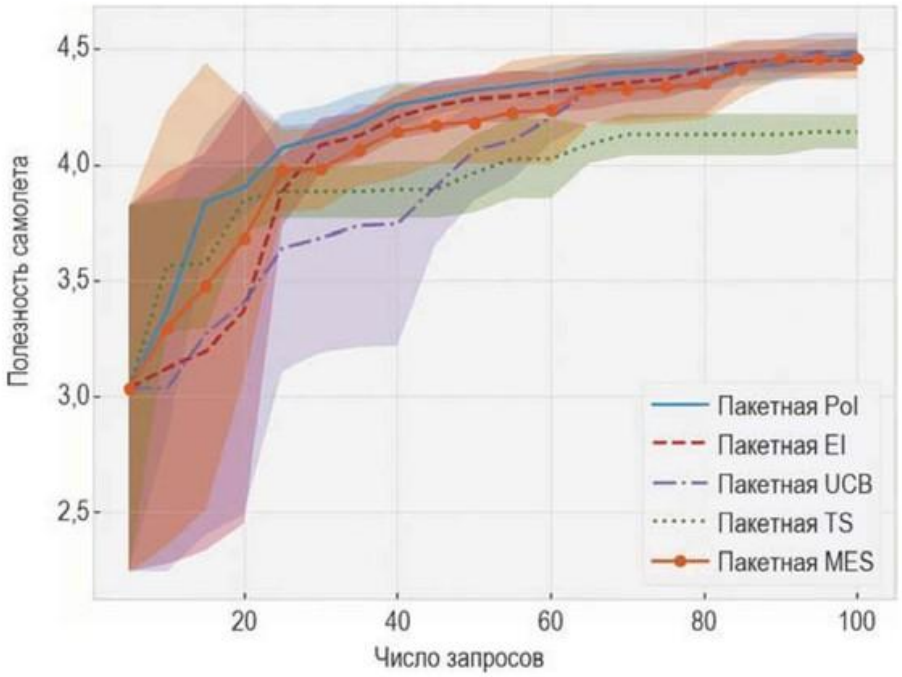


Рис. П.14. Прогресс различных политик в задаче по оптимизации конструкции самолета: большинство политик сопоставимы друг с другом, за исключением TS

П.7. Глава 8: Удовлетворение дополнительных ограничений с помощью ограниченной оптимизации

Для этой главы есть два упражнения:

1. Первое демонстрирует, что результат, полученный от реализации политики ограниченной EI, аналогичен произведению обычной оценки EI и вероятности осуществимости.
2. Второе демонстрирует, как использовать ограниченный вариант БО для решения четырехмерной задачи оптимизации авиационных конструкций.

П.7.1. Упражнение 1: Ручное вычисление ограниченной политики EI

Оценка приобретения ограниченной политики EI является произведением оценки EI и вероятности осуществимости. Несмотря на то что класс `ConstrainedExpectedImprovement` из BoTorch обеспечивает реализацию этой оценки, фактически мы не можем выполнить вычисления вручную. В данном упражнении мы исследуем ручной способ как таковой и сверим результат с результатом класса `ConstrainedExpectedImprovement`. Решение этого упражнения находится в `CH08/02 - Exercise 1.ipynb`. Выполните следующие шаги:

1. Воссоздайте задачу с ограничениями, использованную в `CH08/01 – Constrained Optimization.ipynb`, включая целевую функцию, функцию затрат, реализацию ГП и вспомогательную функцию, которая обучает ГП на некоторых обучающих данных `fit_gp_model()`.

2. Создайте тензор PyTorch, представляющий собой плотную сетку от −5 до 5. Он будет служить тестовым набором. Используйте `torch.linspace()` для создания плотной сетки:

```
lb = -5
ub = 5

xs = torch.linspace(lb, ub, 201)
```

3. Создайте пробный набор обучающих данных, случайным образом выбрав три точки из пространства поиска (от –5 до 5), а затем оценив целевые функции цели и стоимости в этих точках. Используйте `torch.rand()` для случайной выборки от 0 до 1 и масштабируйте выборку в соответствии с пространством поиска:

```
n = 3
#Исправляем начальное значение для воспроизводимости
torch.manual_seed(0)
#Делаем выборки между 0 до 1, а затем масштабируем их
#в соответствии с нашим пространством поиска
train_x = bounds[0] + (bounds[1] - bounds[0]) * torch.rand(n)

train_utility = objective(train_x)
train_cost = cost(train_x)
```

4. Обучите один ГП на данных целевой функции, а другой — на данных функции затрат, используя вспомогательную функцию `fit_gp_model()`.

```
#Обучаем ГП на данных целевой функции
utility_model, utility_likelihood = fit_gp_model(
    train_x.unsqueeze(-1), train_utility
)

#Обучаем ГП на данных функции затрат
cost_model, cost_likelihood = fit_gp_model(
    train_x.unsqueeze(-1), train_cost
)
```

5. Используйте ГП, обученный на данных функции затрат, чтобы вычислить вероятность осуществимости для каждой точки в тестовом наборе.

   Для этого сначала вычислите прогностическое распределение затрат ГП на тестовом наборе:

```
with torch.no_grad():
    cost_pred_dist = cost_likelihood(cost_model(xs))
    cost_pred_mean = cost_pred_dist.mean
    cost_pred_lower, cost_pred_upper = \ cost_pred_dist.confidence_region()
```

   Затем инициализируйте объект нормального распределения со средним значением и стандартным отклонением, соответствующим среднему значению и стандартному отклонению `cost_pred_dist`:

```
normal = torch.distributions.Normal(cost_pred_mean, cost_pred_dist.stddev)
```

Вызовите метод `cdf()` для этого объекта, чтобы вычислить вероятность осуществимости. Аргумент, который принимает метод, — это верхняя граница для ограничения стоимости, равная 0:

```
feasible_prob = normal.cdf(torch.zeros(1))
```

6. Инициализируйте обычную политику EI, при этом аргументом `model` будет ГП, обученный на данных целевой функции, а аргументом `best_f` будет текущее осуществимое действующее значение.

   Для этого вычислите текущее осуществимое действующее значение с помощью `train_utility[train_cost <= 0].max()`:

```
ei = botorch.acquisition.analytic.ExpectedImprovement(
    model=utility_model,
    best_f=train_utility[train_cost <= 0].max(),
)
```

   Затем вычислите оценку EI, вызвав объект политики на `xs[:, None, None]`. Он представляет собой тестовую плотную сетку, которая была изменена с целью убедиться, что она имеет подходящую форму:

```
with torch.no_grad():
    ei_score = ei(xs[:, None, None])
```

7. Инициализируйте ограниченную политику EI и вычислите ее оценку для каждой точки в тестовом наборе:

```
constrained_ei =
botorch.acquisition.analytic.ConstrainedExpectedImprovement(
    model=botorch.models.model_list_gp_regression.ModelListGP(
        utility_model, cost_model
    ),
    best_f=train_utility[train_cost <= 0].max(),
    objective_index=0,
    constraints={1: [None, 0]}
)
```

   Также вычислите эту оценку с помощью измененного набора тестов:

```
with torch.no_grad():
 constrained_ei_score = constrained_ei(xs[:, None, None])
```

8. Вычислите произведение оценок EI и вероятностей осуществимости и убедитесь, что ручной расчет приводит к тем же результатам, что и реализация в BoTorch. Запустите следующий код, чтобы убедиться, что все соответствующие условия совпадают:

```
assert torch.isclose(
 ei_score * feasible_prob, constrained_ei_score, atol=1e-3
).all()
```

9. Постройте график с оценками обычной и ограниченной политик EI и визуально убедитесь, что первые всегда превосходят или равны последним. Докажите, что это так.

Для этого отобразите оценки, которые вычислены на данный момент, следующим образом:

```
plt.plot(xs, ei_score, label="EI")
plt.plot(xs, constrained_ei_score, label="BoTorch constrained EI")
```

Результат этого кода показан на рис. П.15. Он демонстрирует, что оценка обычной EI на самом деле всегда является как минимум оценкой ограниченной EI.

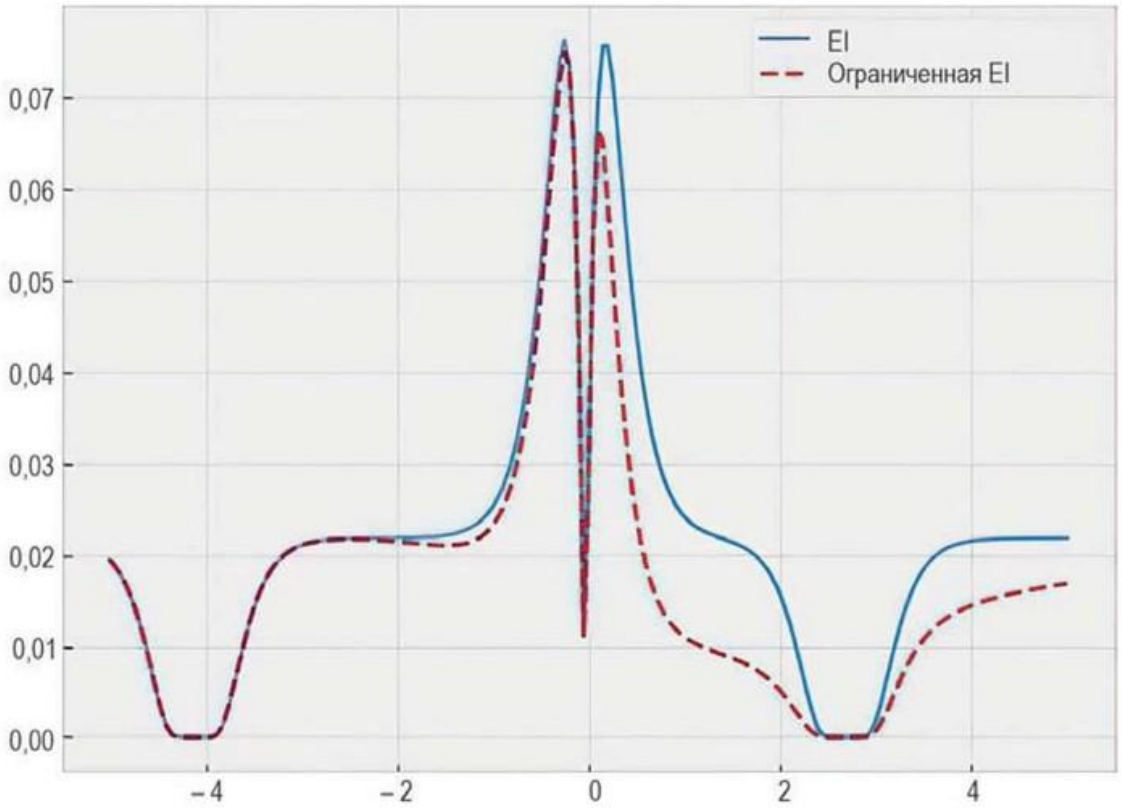


Рис. П.15. Оценка приобретения обычной EI (сплошная линия) и ограниченной EI (пунктирная линия): первая всегда больше или равна второй

Это можно доказать математически, отметив, что оценка приобретения ограниченной EI равна оценке обычной EI, умноженной на вероятность осуществимости (всегда составляет максимум от 1). Поэтому оценка обычной EI всегда больше или равна оценке ограниченной EI.

П.7.2. Упражнение 2: Ограниченная оптимизация конструкции самолета

В этом упражнении решается задача ограниченной оптимизации с использованием целевой функции полезности самолета из упражнения 2 к главе 7. Данный процесс позволяет запустить ограниченную БО для задачи более высокой размерности, в которой неочевидно, где находится оптимальное с точки зрения осуществимости решение. Данное упражнение включено в `CH08/03 - Exercise 2.ipynb`. Выполните следующие шаги:

1. Воссоздайте задачу БО, использованную в `CH07/04 - Exercise 2.ipynb`, включая целевую функцию для самолетов с именем `flight_utility()`, границы пространства

поиска (четырехмерный единичный гиперкуб), реализацию ГП и вспомогательную функцию `fit_gp_model()`, которая обучает ГП.

2. Реализуйте функцию затрат, которая имитирует стоимость создания конструкции самолета. Функция задана четырехмерными входными данными:

```
def flight_cost(X):
    X = X * 20 - 10

    part1 = (X[..., 0] - 1) ** 2

    i = X.new(range(2, 5))
    part2 = torch.sum(i * (2.0 * X[...,1:] ** 2 - X[...,:-1]) ** 2,dim=-1)

    return -(part1 + part2) / 100_000 + 2
```

3. Цель — максимизировать целевую функцию `flight_utility()`, соблюдая при этом ограничение, согласно которому стоимость (вычисленная `flight_utility()`) меньше или равна 0.

   Для этого количество запросов, которые политика может выполнить в каждом эксперименте, установите равным 50. Также обозначьте, что каждая политика должна запустить 10 повторных экспериментов:

```
num_queries = 50
num_repeats = 10
```

   Значение по умолчанию, которое определяет ход оптимизации, если осуществимое решение не найдено, установите равным –2.

```
default_value = -2
feasible_incumbents = torch.ones((num_repeats, num_queries)) * default_value
```

4. Запустите ограниченную и обычную политики EI для этой задачи. Визуализируйте и сравните средний прогресс (вместе с полосами ошибок).

   Реализуйте ограниченную EI так же, как это сделано в главе 8, где `best_f` устанавливается либо в качестве текущего осуществимого действующего значения (если осуществимое решение было найдено), либо в качестве значения по умолчанию –2 (в противном случае). Список моделей содержит целевой ГП, который имеет индекс 0, и ГП для затрат, который имеет индекс 1:

```
#Находим подходящее значение текущей действующей точки
if (train_cost <= 0).any():
    best_f = train_utility[train_cost <= 0].max()
else:
    best_f = torch.tensor(default_value)

policy = botorch.acquisition.analytic.ConstrainedExpectedImprovement(
    #Список моделей ГП
    model=botorch.models.model_list_gp_regression.ModelListGP(
        utility_model, cost_model
    ),
```

```
    best_f=best_f,
    #Индекс целевой модели
    objective_index=0,
    #Индекс ограниченной модели, а также верхняя и нижняя границы
    constraints={1: [None, 0]}
)
```

Реализуйте обычную политику EI следующим образом:

```
policy = botorch.acquisition.analytic.ExpectedImprovement(
    model=utility_model,
    best_f=train_utility.max(),
)
```

На рис. П.16 показаны результаты оптимизации, полученные с помощью двух политик. Ограниченная EI полностью доминирует над обычной политикой EI, поскольку она учитывает налагаемое ограничение на затраты.

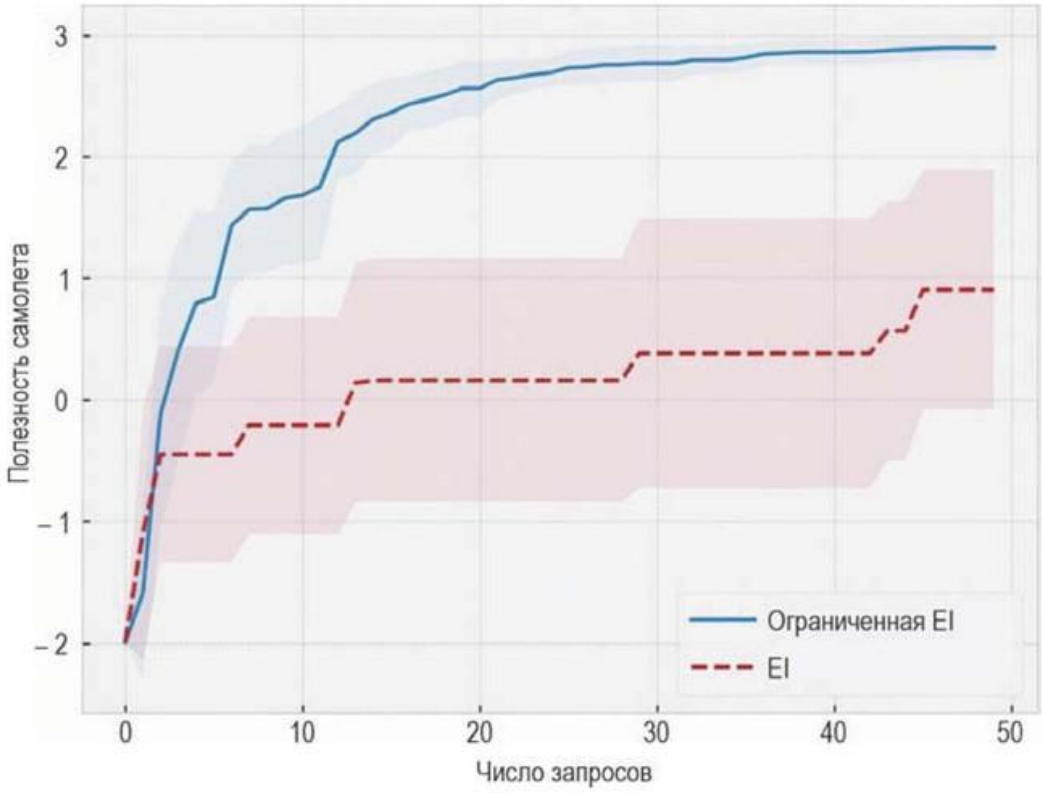


Рис. П.16. Прогресс различных политик БО в примере ограниченной оптимизации конструкции самолета: по сравнению с обычной EI ограниченный вариант в среднем находит более осуществимое решение

П.8. Глава 9: Баланс между полезностью и затратами с помощью разноточной оптимизации

Для этой главы есть два упражнения:

1. В первом рассматривается процесс измерения и визуализации средней эффективности для политики оптимизации в нескольких экспериментах.

2. Во втором известные стратегии оптимизации применяются к двумерной задаче с тремя функциями.

П.8.1. Упражнение 1: Визуализация средней производительности при разноточной оптимизации

В этом упражнении мы запустим цикл оптимизации несколько раз и узнаем, как взять среднюю производительность, чтобы получить наиболее целостное сравнение. Выполните следующие шаги:

1. Скопируйте настройки, а также цикл разноточной оптимизации из `CH09/03 – Measuring Performance.ipynb`. Добавьте еще одну переменную, обозначающую количество экспериментов, которые нужно провести (по умолчанию 10).
2. Для облегчения проведения повторных экспериментов добавьте внешний цикл в код оптимизации. Это должен быть цикл `for` с 10 итерациями, в котором каждый раз генерируется разное случайное наблюдение:

```
#Повторяем эксперимент 10 раз
num_repeats = 10

for trial in range(num_repeats):
    #Генерируем случайный начальный обучающий набор,
    #соответствующий текущей итерации
    torch.manual_seed(trial)
    train_x = bounds[0] + (bounds[1] - bounds[0]) * torch.rand(1, 1)
    train_x = torch.cat(
        [train_x, torch.ones_like(train_x) * fidelities[0]], dim=1
    )
    train_y = evaluate_all_functions(train_x)

    #Внутренний цикл, который выполняет оптимизацию,
    #пока мы не исчерпаем бюджет
    current_budget = 0
    while current_budget < budget_limit:
        ...
```

3. Сделайте каждую из переменных `recommendations` и `spent_budget` списком из списков, где каждый внутренний список отслеживает эффективность оптимизации отдельного эксперимента. Добавьте в код вложенного цикла следующее:

```
num_repeats = 10
#Каждая переменная – это список списков (пока что пустой)
recommendations = []
spent_budget = []

for trial in range(num_repeats):
    torch.manual_seed(trial)
    train_x = bounds[0] + (bounds[1] - bounds[0]) * torch.rand(1, 1)
```

```
train_x = torch.cat(
    [train_x, torch.ones_like(train_x) * fidelities[0]], dim=1
)
train_y = evaluate_all_functions(train_x)

#Добавляем пустой список к каждому списку списков
#для следующего эксперимента
current_budget = 0
recommendations.append([])
spent_budget.append([])

while current_budget < budget_limit:
    ...

    #Добавляем статистику хода оптимизации в самый новый
    #список для каждой переменной
    rec_x = get_final_recommendation(model)
    recommendations[-1].append(evaluate_all_functions(rec_x).item())
    spent_budget[-1].append(current_budget)

    ...
```

4. Запустите разноточную политику MES и ее одноточную версию для задачи оптимизации.
5. Сначала создайте обычную сетку и пустые на данный момент интерполированные рекомендуемые значения, которые будут заполнены позже:

```
xs = np.arange(budget_limit)
interp_incumbents = np.empty((num_repeats, budget_limit))
```

   После переберите каждый список в `recommendations` (в нашем коде переименованы в `incumbents`) и `spent_budget`, вычислите линейную интерполяцию, а затем заполните значения в `interp_incumbents`:

```
for i, (tmp_incumbents, tmp_budget) in enumerate(
     zip(incumbents, spent_budget)
):
     interp_incumbents[i, :] = np.interp(xs, tmp_budget, tmp_incumbents)
```

6. Используйте линейно интерполированные значения, чтобы построить среднюю производительность и полосы ошибок обеих используемых политик, а затем сравнить их. Визуализация показана на рис. П.17, где видно, что разноточная политика MES значительно превосходит своего одноточного конкурента.
7. Постройте линейно интерполированные кривые, представляющие прогресс оптимизации отдельных прогонов, а также среднюю производительность и полосы ошибок. Сравнение показано на рис. П.18. Прогресс оптимизации в каждом прогоне, измеряемый как максимальная апостериорная средняя рекомендация, не увеличивается монотонно.

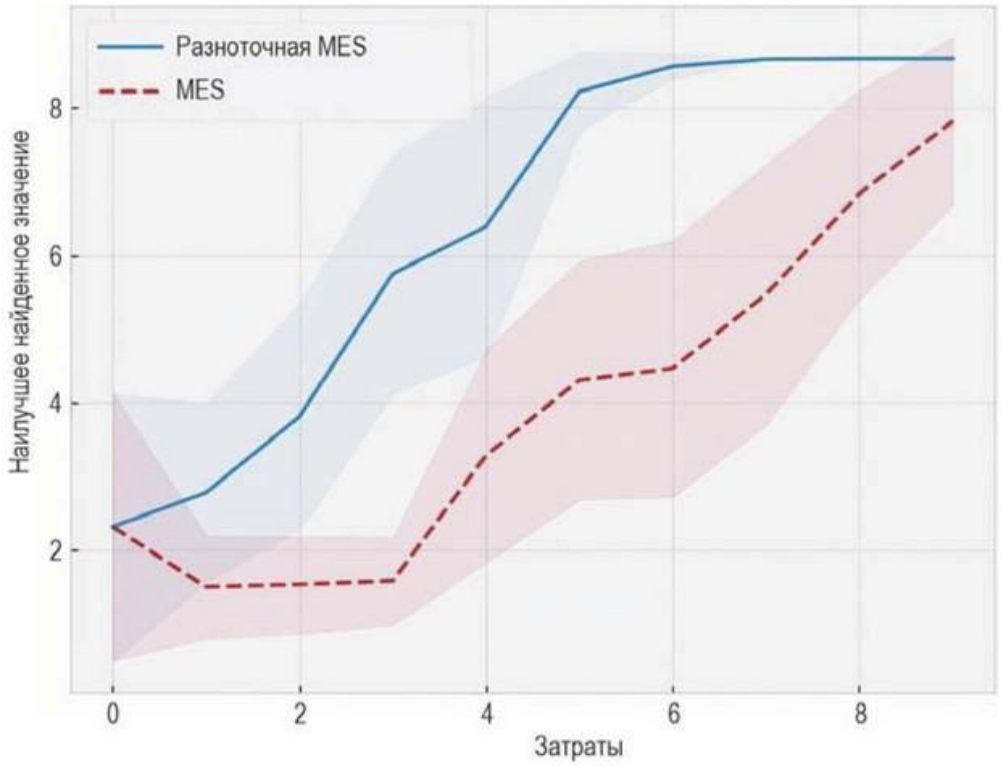


Рис. П.17. Средний прогресс оптимизации одноточной и разноточной политик MES для функции Форрестера по 10 экспериментам: разноточная политика значительно превосходит одноточную

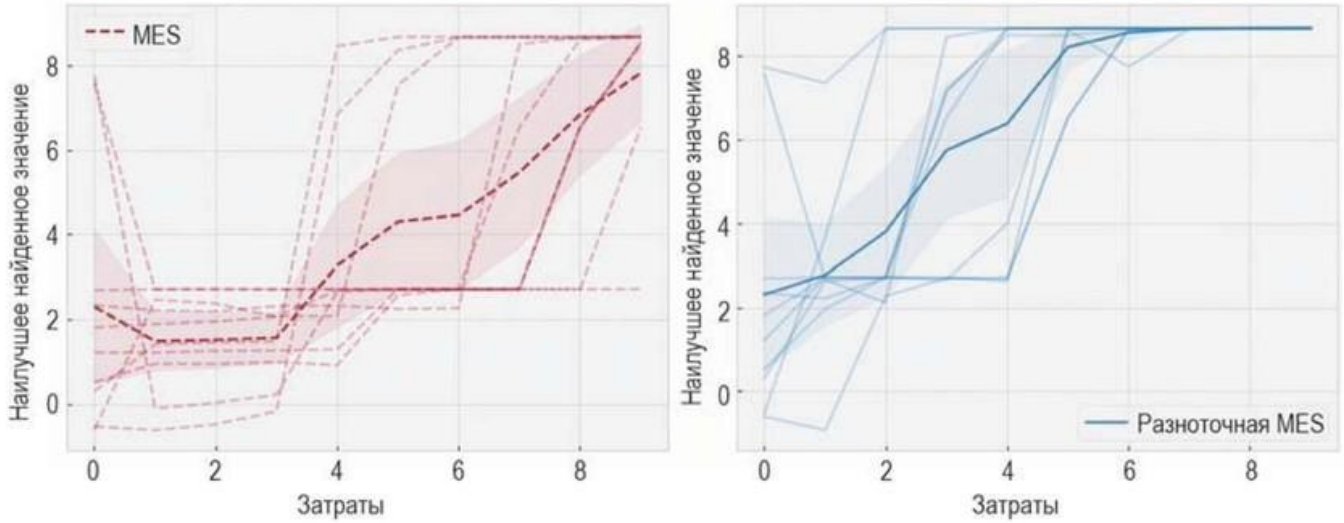


Рис. П.18. Линейно интерполированные кривые, показывающие прогресс оптимизации отдельных прогонов по 10 экспериментам: прогресс оптимизации в каждом прогоне, как максимальная апостериорная средняя рекомендация, не увеличивается монотонно

П.8.2. Упражнение 2: Разноточная оптимизация с использованием нескольких низкоточных приближений

Это упражнение демонстрирует, что политика разноточного поиска энтропии с максимальным значением может балансировать между несколькими функциями с низкой точностью. Решение задачи находится в `CH09/05 - Exercise 2.ipynb`. Выполните следующие шаги:

1. Реализуйте целевую функцию. Код для этого шага уже указан в инструкции.
2. Определите границы пространства поиска как единичный квадрат:

```
bounds = torch.tensor([[0.0] * 2, [1.0] * 2])
```

3. Объявите переменную `fidelities`, в которой хранятся значения корреляции различных функций, которые можно запросить:

```
fidelities = torch.tensor([0.1, 0.3, 1.0])
bounds_full = torch.cat(
    [
        bounds,
        torch.tensor([fidelities.min(), fidelities.max()]).unsqueeze(-1)
    ],
    dim=1
)
```

4. Установите фиксированные затраты для модели линейной стоимости равные 0,2, а вес — 1:

```
from botorch.models.cost import AffineFidelityCostModel
cost_model = AffineFidelityCostModel(fixed_cost=0.2)
```

Установите лимит бюджета в каждом эксперименте равным 10. Количество повторных экспериментов также равно 10:

```
budget_limit = 10
num_repeats = 10
```

5. Установите количество кандидатов, выбранных из последовательности Соболя, равным 5000. Используйте 100 перезапусков и 500 необработанных выборок при выполнении вспомогательных функций для оптимизации оценки приобретения данной политики:

```
num_samples = 5000

num_restarts = 100
raw_samples = 500
```

6. Переопределите вспомогательную функцию `get_final_recommendation`, которая находит максимизатор апостериорного среднего:

```
from botorch.acquisition.fixed_feature import
  FixedFeatureAcquisitionFunction
from botorch.acquisition import PosteriorMean
from botorch.optim.optimize import optimize_acqf, optimize_acqf_mixed

def get_final_recommendation(model):
    post_mean_policy = FixedFeatureAcquisitionFunction(
        acq_function=PosteriorMean(model),

        #Необходимые изменения
        d=3,
        columns=[2],
```

```
        values=[1],
    )

    final_x, _ = optimize_acqf(
        post_mean_policy,
        bounds=bounds,
        q=1,
        num_restarts=num_restarts,
        raw_samples=raw_samples,
    )

    return torch.cat([final_x, torch.ones(1, 1)], dim=1)
```

7. Запустите разноточный и одноточный варианты политики MES для задачи оптимизации и постройте график среднего прогресса для хода оптимизации и полосы ошибок в каждой политике, используя метод, описанный в упражнении 1. На рис. П.19 показано сравнение между двумя политиками.

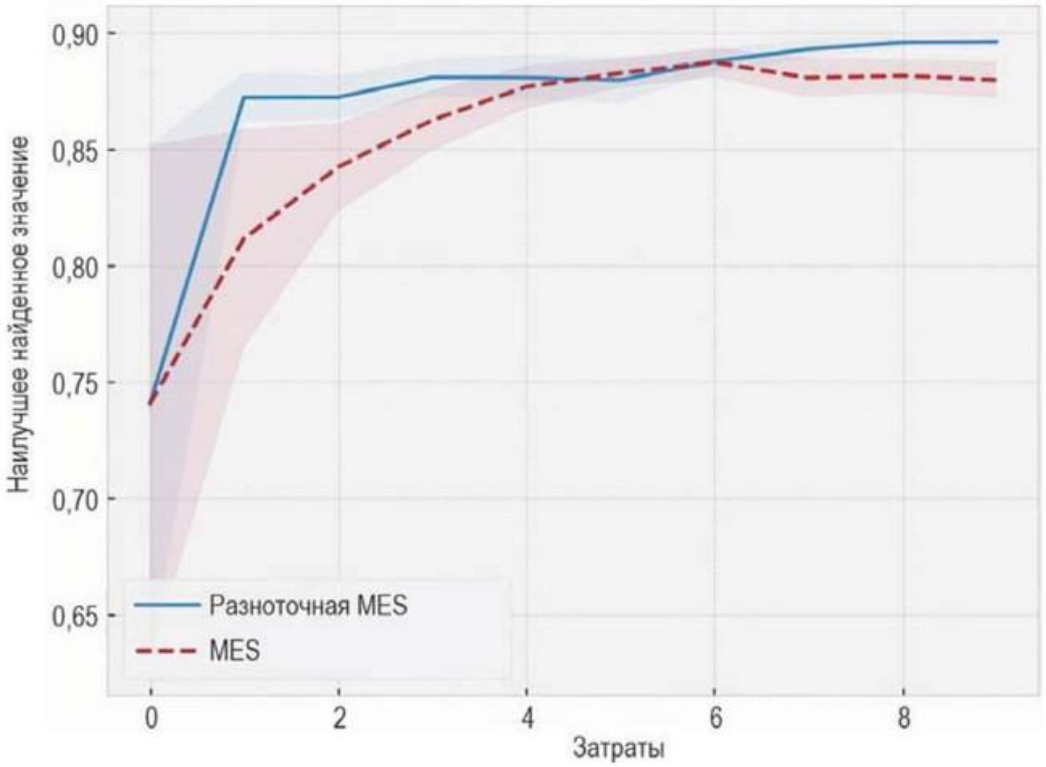


Рис. П.19. Средний прогресс оптимизации одноточной и многоточной политик MES для функции Бранина по 10 экспериментам: разноточный вариант снова превосходит одноточный

П.9. Глава 11: Оптимизация нескольких целевых функций одновременно

В этом упражнении применяются изученные методы многоцелевой оптимизации к задаче аэроструктурной конструкции самолета. Упражнение позволяет наблюдать за эффективностью политики ожидаемого улучшения гиперобъема (EHVI) для многомерной задачи. Выполните следующие шаги:

1. Скопируйте код целевых функций:

```
#Первая целевая функция
def objective1(X):
    X_copy = X.detach().clone()
    X_copy[:, [2, 3]] = 1 - X_copy[:, [2, 3]]
    X_copy = X_copy * 10 - 5
    return (
        -0.005
        *(X_copy ** 4 - 16 * X_copy ** 2 + 5 * X_copy).sum(dim=-1)
        + 3
    )

#Вторая целевая функция, отрицаемая из кода упражнения 2 к главе 8
def objective2(X):
    X = X * 20 - 10
    part1 = (X[..., 0] - 1) ** 2
    i = X.new(range(2, 5))
    part2 = torch.sum(i * (2.0 * X[..., 1:] ** 2 - X[..., :-1]) ** 2,dim=-1)
    return (part1 + part2) / 100_000 - 2
```

2. Реализуйте вспомогательную функцию:

```
def joint_objective(X):
    return torch.vstack(
        [
            objective1(X).flatten(),
            objective2(X).flatten(),
        ]
    ).transpose(-1, -2)
```

3. Объявите границы пространства поиска:

```
bounds = torch.tensor([[0.0] * 4, [1.0] * 4])
```

4. Объявите точку отсчета:

```
ref_point = torch.tensor([-1.5, -2.0])
```

5. Класс и вспомогательная функция могут быть реализованы с использованием того же кода, что и, например, в `CH08/03 -Exercise 2.ipynb`.

6. Задайте экспериментальные настройки:

```
num_queries = 50
num_repeats = 10
```

7. Реализуйте две политики БО так же, как в `CH11/02 - Multiobjective BayesOpt loop.ipynb`. На рис. П.20 показана эффективность двух политик в совокупности по 10 экспериментам. EHVI в очередной раз превосходит альтернативную политику EI.

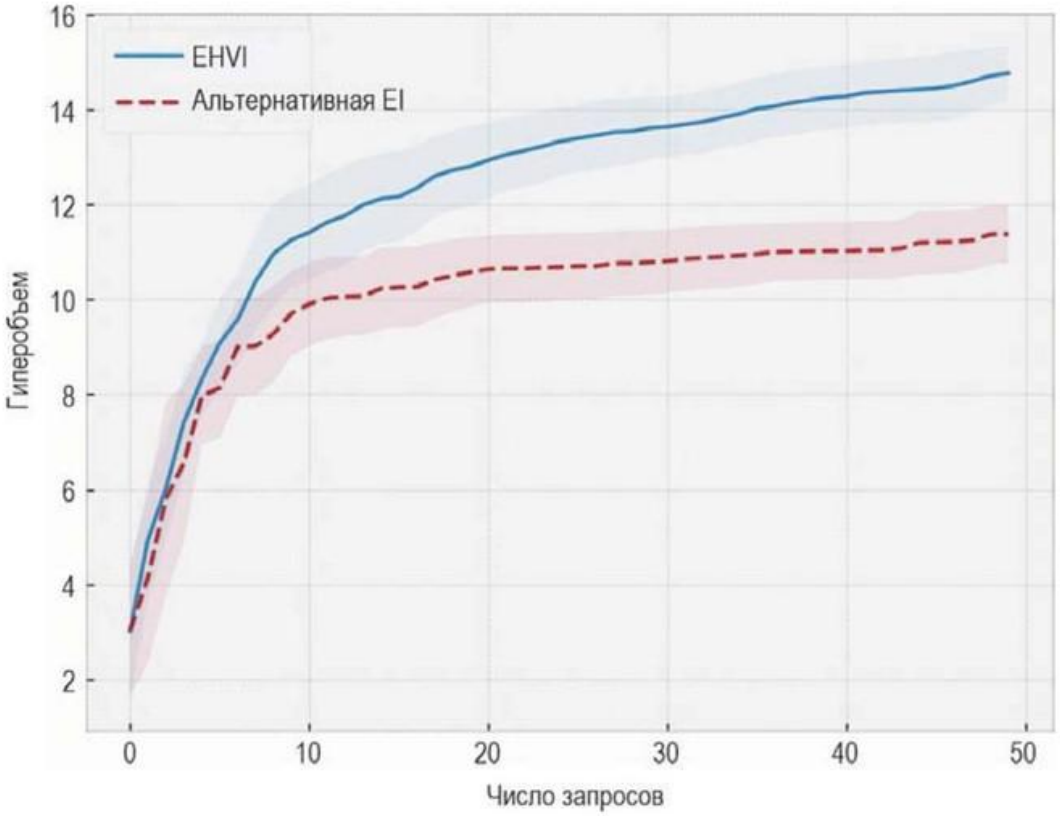


Рис. П.20. Средний гиперобъем и полосы ошибок как функция числа запросов, выполненных двумя политиками БО: EHVI неизменно превосходит альтернативную политику EI

П.10. Глава 12: Масштабирование гауссовских процессов на большие наборы данных

Это упражнение демонстрирует повышение эффективности при переходе от обычной модели ГП к модели VGP на реальном наборе данных. Пример затрагивает цены на жилье в Калифорнии. Цель — оценить вычислительные преимущества VGP в реальных условиях.

Выполните следующие шаги:

1. Используйте библиотеку Pandas для чтения набора данных:

```
import pandas as pd
df = pd.read_csv("../data/housing.csv")
```

   После считывания фрейм данных Pandas должен выглядеть примерно так, как показано на рис. П.21.

2. Создайте точечную диаграмму:

```
plt.figure(figsize=(8, 6))
plt.scatter(df.longitude, df.latitude, c=np.log(df.median_house_value))
plt.colorbar();
```

   Визуализация должна выглядеть аналогично рис. П.22.

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
0	-117.08	32.70	37	2176	418.0	1301	375	2.8750	98900
1	-117.91	34.11	20	3158	684.0	2396	713	3.5250	153000
2	-117.10	32.75	11	2393	726.0	1905	711	1.3448	91300
3	-117.22	32.74	52	1260	202.0	555	209	7.2758	345200
4	-121.99	37.29	32	2930	481.0	1336	481	6.4631	344100
...	...	...	...	...	...	...	...	...	...
4995	-118.41	34.19	42	779	145.0	450	148	3.9792	193800
4996	-122.96	38.42	50	2530	524.0	940	361	2.9375	122900
4997	-118.00	33.77	24	1324	267.0	687	264	3.4327	192800
4998	-122.81	38.54	12	2289	611.0	919	540	1.1553	139300
4999	-118.08	33.84	25	3696	953.0	2827	860	3.3438	153300

Рис. П.21. Набор данных о ценах на жилье в виде фрейма данных Pandas: это тренировочный набор данного упражнения

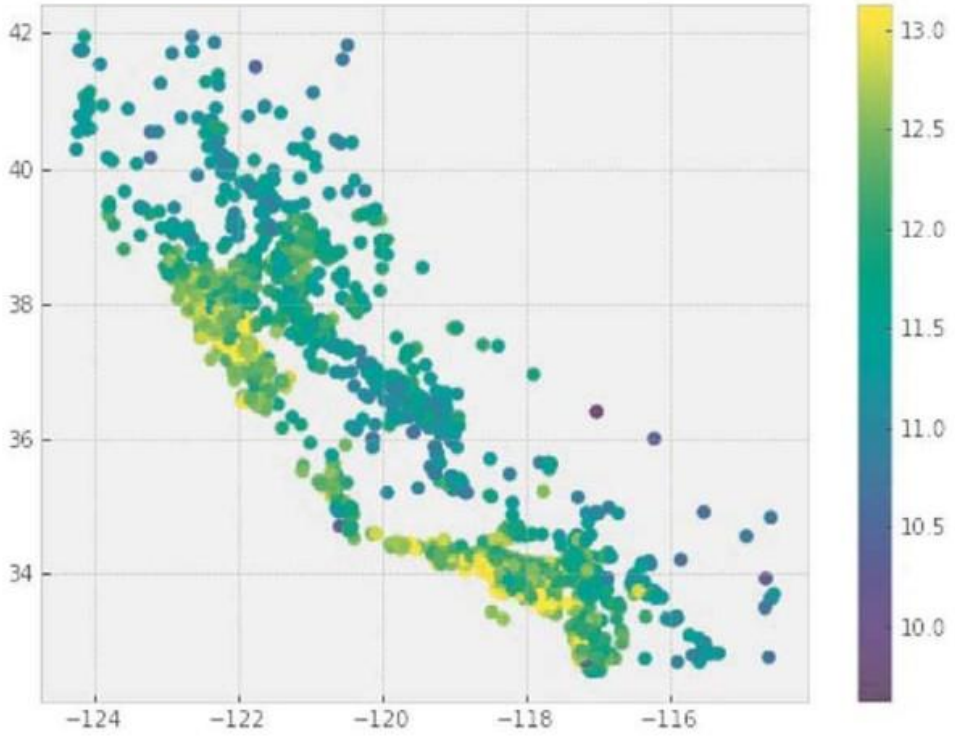


Рис. П.22. Набор данных о ценах на жилье в виде диаграммы рассеяния

3. Чтобы извлечь функции обучения, используйте метод `torch.from_numpy()` для преобразования массива NumPy в тензор PyTorch:

```
train_x = torch.from_numpy(df.drop(["median_house_value"], axis=1).values)
```

4. Аналогичным образом сделайте то же самое для списка цен на жилье, который является обучающими метками:

```
train_y = torch.from_numpy(
    df.median_house_value.values
).log().to(train_x.dtype)
```

5. Нормализуйте обучающие метки `train_y` следующим образом:

```
train_y = (train_y - train_y.mean()) / train_y.std()
```

6. Реализуйте модель ГП:

```
class GPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super().__init__(train_x, train_y, likelihood)
        #Постоянная функция среднего
        self.mean_module = gpytorch.means.ConstantMean()
        #Ядро Матерна 5/2 с ARD и масштабом вывода
        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.MaternKernel(
                nu=2.5,
                ard_num_dims=train_x.shape[1]
            )
        )

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)
```

7. Создайте вероятность, шум которой ограничен величиной не менее 0,1:

```
likelihood = gpytorch.likelihoods.GaussianLikelihood(
    noise_constraint=gpytorch.constraints.GreaterThan(1e-1)
    #Ограничение определяет значение шума не менее 0.1
)
```

8. Обучите ранее реализованную модель ГП с использованием градиентного спуска:

```
model = GPModel(train_x, train_y, likelihood)

#Оптимизатор градиентного спуска Adam
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
#Отрицательный предельный логарифм вероятности для функции потерь
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

#Включаем обучающий режим
model.train()
likelihood.train()

for i in tqdm(range(10)):
    optimizer.zero_grad()

    output = model(train_x)
    loss = -mll(output, train_y)

    loss.backward()
    optimizer.step()
```

Общее время обучения на MacBook составило 24 секунды.

9. Реализуйте модель VGP:

```
class ApproximateGPModel(gpytorch.models.ApproximateGP):
    #Вариационные параметры (вариационное распределение)
    def __init__(self, inducing_points):
        variational_distribution = gpytorch.variational
          .NaturalVariationalDistribution(
              inducing_points.size(0)
        )

        #Вариационные параметры (вариационная стратегия)
        variational_strategy = gpytorch.variational.VariationalStrategy(
            self,
            inducing_points,
            variational_distribution,
            learn_inducing_locations=True,
        )

        super().__init__(variational_strategy)
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(
            gpytorch.kernels.MaternKernel(
                nu=2.5,
                ard_num_dims=inducing_points.shape[1]
            )
        )
        def forward(self, x):
          ... #То же самое как и в ГП
```

10. Обучите VGP:

```
#Случайным образом выбираем 100 точек в качестве
#начальных стимулирующих точек
num_datapoints = 100
torch.manual_seed(0)
model = ApproximateGPModel(
    train_x[torch.randint(train_x.shape[0],(num_datapoints,)), :]
)

likelihood = gpytorch.likelihoods.GaussianLikelihood(
    noise_constraint=gpytorch.constraints.GreaterThan(1e-1)
)

#Готовим мини-пакеты
train_dataset = torch.utils.data.Tensordataset(train_x, train_y)
train_loader = torch.utils.data.DataLoader(
    train_data set,
    batch_size=100,
    shuffle=True
)
```

```
#Естественный градиентный спуск для вариационных параметров
ngd_optimizer = gpytorch.optim.NGD(
    model.variational_parameters(),num_data=train_y.size(0), lr=0.1
)

#Adam для остальных параметров
hyperparam_optimizer = torch.optim.Adam(
    [{"params": model.parameters()}, {"params":likelihood.parameters()}], lr=0.01
)

mll = gpytorch.mlls.VariationalELBO(
    likelihood, model, num_data=train_y.size(0)
)

model.train()
likelihood.train()

for i in tqdm(range(10)):
    for x_batch, y_batch in train_loader:
        ngd_optimizer.zero_grad()

        output = model(x_batch)
        loss = -mll(output, y_batch)

        loss.backward()

        ngd_optimizer.step()
        hyperparam_optimizer.step()
```

11. На том же MacBook обучение заняло 6 секунд. Увеличение скорости составило 400%.

    Решение включено в `CH12/02 - Exercise.ipynb`.

Предметный указатель

А

Автоматическое определение релевантности (Automatic Relevance Determination, ARD) 102

Б

Байесовская оптимизация (Bayesian optimization) 11, 30
Байесовский вывод 11

В

Вариационная стратегия (Variational strategy) 332
Вариационное распределение (Variational distribution) 332
Вариационный гауссовский процесс (Variational Gaussian Process, VGP) 318
Вероятность (Likelihood) 286
Вероятность улучшения (Probability of Improvement, PoI) 18, 107, 127
Верхний доверительный предел (Upper Confidence Bound, UCB) *См.* Верхняя доверительная граница
Верхняя доверительная граница (Upper Confidence Bound, UCB) 18, 107
Взвешенная сумма (Weighted sum) 313
Выборка Томпсона (Thompson Sampling, TS) 18, 107, 157

Г

Гауссиана *См.* Многомерное нормальное распределение
Гауссовский процесс (Gaussian Process, GP) 11, 31, 60
Гладкость (Smoothness) 99
Глубокое обучение (Deep Learning, DL) 25
Градиентный спуск (Gradient descent) 70
Графический процессор (Graphics Processing Units, GPU) 199
Гребневая регрессия (Grid Regression) 31

Д

Действующая точка (Incumbent point) 124
Действующее значение (Incumbent value) 124
Дерево решений (Decision Tree) 31
Дивергенция Кульбака — Лейблера (Kullback — Leibler divergence, KL-расхождение) 327
Динамическое обучение (Dynamically learning) 361
Доверительный интервал (Confidence interval) 33
Доминирующая точка (Dominated point) 301
Доминирующий гиперобъем (Dominated Hypervolume) 304
Допустимая точка данных (Infeasible data point) 229
Достоверный интервал (Credible interval) 33

Е

Естественный градиентный спуск (Natural Gradient Descent, NGD) 338

И

Исследование (Exploration) 107

К

Калиброванная количественная оценка неопределенности (calibrated quantification of uncertainty) 64
Колоколообразная кривая (Bell curve) 51
Кумулятивная функция распределения (Cumulative Density Function, CDF) 127

М

Магнитуда (Magnitude) 135
Масштаб вывода 95
Масштаб длины 95
Машина опорных векторов (Support Vector Machine, SVM) 26, 104
Машинное обучение (Machine Learning, ML) 11
Мини-пакетный градиентный спуск (Mini-batch Gradient descent) 330
Многомерное гауссово распределение *См.* Многомерное нормальное распределение
Многомерное нормальное распределение (Multivariate Normal Distribution, MVN) 51
Многорукий бандит (Multi-Armed Bandit, MAB) 18, 107, 143, 145
Многоцелевая оптимизация (Multiobjective optimization) 296

Модель машинного обучения (Machine learning model, ML-model) 61

Н

Настройка гиперпараметров (Hyperparameter tuning) 25
Недоминирующая точка (Nondominated point) 301
Недопустимая точка данных (Infeasible data point) 229
Неуступающая точка (Noninferior point) 301
Нижняя граница доказательств (Evidence Lower Bound, ELBO) 329

О

Обусловливание (Conditioning) 55
Обучение с подкреплением (Reinforcement Learning, RL) 112
Ограничение неравенств (Inequality constraint) 228
Ограниченная оптимизация (Constrained optimization) 228
Ожидаемое значение (Expected value) 136, 309
Ожидаемое улучшение (Expected Improvement, EI) 18, 107, 135, 136
Ожидаемое улучшение гиперобъема (Expected Hypervolume Improvement, EHVI) 309
Оптическое распознавание символов (Optical Character Recognition, ORC) 22
Оценка приобретения (Acquisition score) 34, 107, 112

П

Пакетная байесовская оптимизация (Batch Bayesian optimization) 197
Параллелизм 198
Парное сравнение 278
Плавность ГП (Smoothness) 64
Планирование экспериментов (Design of Experiments, DoE) 35
Поиск энтропии с максимальным значением (Max-value Entropy Search, MES) 185
Последовательность Соболя (Sobol sequence) 161
Проклятие размерности (curse of dimensionality) 184
Простое вознаграждение (simple reward) 148
Пул-реквест (Pull request) 20

Р

Радиальная базисная функция (Radial Basis Function, RBF) 68
Разноточная оптимизация (Multifidelity optimization) 247
Разноточность (multifidelity) 196
Регрессия гауссовского процесса 12
Рентабельность инвестиций (Return on Investment, ROI) 263

С

Секуляризация (Scalarization) 313
Средняя абсолютная ошибка (Mean Absolute Error) 359
Стимулирующая точка (Inducing point) 319
Суррогатная модель 31

Т

Теория информации (Information theory) 171
Тепловая карта (Heat map) 59
Точка оптимальная по Парето (Pareto optimal point) 301
Точка эффективная по Парето (Pareto efficient point) 301

У

Условное распределение (Conditional distribution) 55

Ф

Фронт Парето (Pareto frontier) 302
Функция вероятности (Likehood function) 67

Ц

Цикл БО 112
Циклическая оптимизация 219

Ч

Черный ящик (Black box) 27
Численная оптимизация функции сбора оценки 12

Э

Эксплуатация (Exploitation) 107
Экстрактор характеристик (Feature extractor) 361

Куан Нгуен

Байесовская оптимизация с примерами из библиотек Python

Перевод с английского А. Морозова

ТОО "АЛИСТ"
010000, Республика Казахстан,
г. Астана, пр. Сарыарка, д. 17, ВП 30