
QMLCourse

ODS Quantum Community

Dec 13, 2021

CONTENTS

1 О квантовых компьютерах, биткоине и превосходстве	5
1.1 Описание лекции	5
1.2 Что это вообще за компьютеры такие?	5
1.3 А зачем это вообще нужно?	8
1.4 Ну и когда взламают биткоин?	12
1.5 О квантовом превосходстве	15
1.6 А как это вообще выглядит? И сколько стоит?	17
1.7 Заключение	17
2 Пререквизиты для прохождения курса QML	19
2.1 Python, Poetry и среда разработки	19
2.2 Линейная алгебра и Numpy	20
2.3 Машинное обучение	20
3 О блоке “Python”	21
4 Вводная лекция про Python	23
4.1 Что такое Python?	23
4.2 Но не путайтесь!	25
5 Установка необходимых инструментов и настройка среды	27
5.1 Описание лекции	27
5.2 Git и получение файлов курса	27
5.3 Что делать дальше	28
5.4 Проверка корректности установки	31
5.5 Что мы узнали в лекции	33
6 Знакомство с инструментарием: Jupyter	35
6.1 Описание лекции	35
6.2 Введение	35
6.3 Типы ячеек	35
6.4 Режимы работы	39
6.5 Оставшиеся кнопки на панели	41
6.6 Что это за In [*]?	42
6.7 Самая полезная клавиша	43
6.8 Что мы узнали из лекции	44
6.9 Бонус-материал	44
7 Переменные и вывод информации в Python	45
7.1 Описание лекции	45
7.2 Суть переменных в Python	45

7.3	Типы переменных	46
7.4	Арифметические операции с числами	47
7.5	Строковые переменные	49
7.6	Что мы узнали из лекции	51
8	Условные конструкции, булева логика и сравнения	53
8.1	Описание лекции	53
8.2	Ветвление логики	53
8.3	А что вообще такое эти ваши условия?	54
8.4	Способы получения <code>bool</code>	55
8.5	Блоки кода и отступы	57
8.6	Что мы узнали из лекции	58
9	Списки и циклы в Python	59
9.1	Описание лекции	59
9.2	Введение в списки объектов	59
9.3	Индексация списков	61
9.4	Срезы	62
9.5	Циклы	63
9.6	List comprehensions	65
9.7	Что мы узнали из лекции	66
10	Словари	67
10.1	Описание лекции	67
10.2	Что такое <code>dict</code>	67
10.3	Основные методы словаря	69
10.4	Доступ к ключам и значениям	72
10.5	Сортировка	73
10.6	Что можно хранить	74
10.7	Что мы узнали из лекции	76
11	Функции	79
11.1	Описание лекции	79
11.2	Что такое “функция” в Python	79
11.3	Именованные функции	80
11.4	Пространства имен и области видимости	82
11.5	Параметры	84
11.6	Анонимные функции	87
11.7	Декораторы	87
11.8	Что мы узнали из лекции	89
12	Объекты, классы и методы	91
12.1	Описание лекции	91
12.2	ООП: Объектно-Ориентированное Программирование	91
12.3	Определение классов	93
12.4	Пример использования для хранения состояния	95
12.5	Что мы узнали из лекции	97
13	Финальная лекция обо всем и ни о чем	99
13.1	Описание лекции	99
13.2	Заметки по прохождению курса	99
13.3	Примеры кода, не разобранные в лекциях	103
13.4	Блок дополнительных ссылок	104
13.5	Вместо заключения	105

14 О блоке “Линейная алгебра и NumPy”	107
15 Векторы	109
15.1 Определение	109
15.2 Операции с векторами	110
15.3 Норма (длина) вектора	110
15.4 Скалярное произведение	110
15.5 Линейная независимость	111
15.6 Базис	112
15.7 Что мы узнали?	113
16 Скобки	115
16.1 Гильбертово пространство	115
16.2 Полное пространство	115
16.3 Нотация Дирака, или Bra-Ket нотация	115
16.4 Внешнее произведение (outer-product)	116
16.5 Эрмитов оператор	116
16.6 Унитарный оператор	116
16.7 Произведение Кронекера	119
16.8 Рекомендованная литература	120
16.9 Что мы узнали?	121
17 Матрицы	123
17.1 Определение	123
17.2 Операции с матрицами	123
17.3 Норма матрицы	125
17.4 Ранг матрицы	125
17.5 Линейные преобразования	125
17.6 Обратная матрица	127
17.7 Определитель	128
17.8 Собственные числа и векторы	128
17.9 Что мы узнали?	129
18 Numpy	131
18.1 Массивы	131
18.2 Арифметические операции	132
18.3 Indexing, slicing and sugar	132
18.4 Broadcasting	134
18.5 Операции с плавающей точкой	135
18.6 NumPy и линейная алгебра	136
18.7 Мы вкратце рассмотрели	138
19 О блоке “Машинное обучение”	139
20 Машинное обучение как пазл	141
20.1 Введение	141
20.2 Составляющие части задачи машинного обучения	142
21 О блоке “Квантовые вычисления”	153
22 Квантовый бит	155
22.1 Описание лекции	155
22.2 Введение	155
22.3 Что такое кубит	155
22.4 Состояние vs значение	156

22.5 Сфера Блоха	159
22.6 Что можно делать с таким кубитом?	161
22.7 Измерение	163
22.8 Что мы узнали?	167
23 Квантовые гейты	169
23.1 Описание лекции	169
23.2 Введение	169
23.3 Основные однокубитные гейты	170
23.4 Многокубитные состояния и гейты	174
23.5 Основные двухкубитные гейты	176
23.6 Первое знакомство с PennyLane	177
23.7 Заключение	179
23.8 Задачи	179
24 Смешанные состояния и матрицы плотности	181
24.1 Описание лекции	181
24.2 Смешанные состояния	181
24.3 Почему нельзя обойтись волновой функцией	182
24.4 Матрица плотности	182
24.5 Чистые и смешанные состояния	184
24.6 Смешанные состояния и запутанность	185
24.7 Описания эволюции смешанного состояния	188
24.8 Что мы узнали?	189
25 О блоке “Фреймворки”	191
26 Обзор фреймворков для квантовых вычислений	193
26.1 Квантовые ассемблеры	193
26.2 Высокоуровневое программирование	195
26.3 Заключение	198
27 PennyLane	199
27.1 QNode	199
27.2 Операторы	200
27.3 QubitUnitary	201
28 Cirq & TFQ	203
28.1 Введение	203
28.2 Работа с кубитами	203
28.3 Квантовое машинное обучение	205
28.4 Заключение	211
29 Qiskit	213
29.1 Кубиты	213
29.2 Квантовые гейты	214
29.3 Визуализация	215
29.4 Экосистема Qiskit	216
30 О блоке “Базовые квантовые алгоритмы”	217
31 Обзор квантовых алгоритмов	219
31.1 Классификация задач по временной сложности	219
31.2 Наиболее известные квантовые алгоритмы	220
31.3 Алгоритмы в нашем курсе	221

32 Алгоритм Дойча	223
32.1 Алгоритм Дойча в коде	227
33 Квантовая телепортация	229
34 Квантовое сверхплотное кодирование	233
35 Алгоритм Гровера	237
35.1 Реализация алгоритма Гровера	243
35.2 Задание	245
36 Алгоритм Саймона	247
36.1 Задача Саймона	247
36.2 Описание алгоритма	247
36.3 Пример	249
36.4 Приложение	252
36.5 Ссылки	253
37 О блоке “Железо квантовых компьютеров”	255
38 Обзор и классификация физических платформ для квантовых вычислений	257
38.1 Радиочастотные и оптические кубиты	257
38.2 Природа степени свободы для кодирования информации	258
38.3 Способ кодирования информации	260
38.4 Способ выполнения операций	261
38.5 Наборы инструкций и универсальность	262
38.6 Успехи в практической реализации	264
39 Сверхпроводниковые кубиты	265
39.1 Квантование электрических цепей	265
39.2 Джозефсоновский контакт	268
39.3 Трансмон	269
39.4 Считывание	271
39.5 Микроволновые однокубитные операции	272
39.6 Декогеренция в сверхпроводниковых кубитах	275
39.7 Двухкубитные вентили	276
39.8 Что почитать для дальнейшего изучения вопроса?	278
40 О блоке “Квантовое машинное обучение”	281
41 О квантовом машинном обучении	283
41.1 Виды квантового машинного обучения	283
41.2 Квантовые данные и классический алгоритм	283
41.3 Классические данные и квантовый алгоритм	288
41.4 Гибридное обучение	289
41.5 Заключение	291
42 Алгоритм HHL	293
42.1 Задача	293
42.2 Реализация HHL	295
42.3 Пример	298
43 Quantum K-nearest neighbor	301
43.1 Введение	301
43.2 Алгоритм	302

44 О блоке “Вариационные схемы”	307
45 Вариационные квантовые схемы	309
45.1 Описание лекции	309
45.2 Введение	309
45.3 Квантово-классическое обучение	309
45.4 Вариационные квантовые схемы	311
45.5 Кодирование данных в VQC	312
45.6 Что мы узнали из лекций	315
46 О блоке “Квантовые градиенты”	317
47 Градиенты квантовых схем	319
47.1 Описание лекции	319
47.2 Введение	319
47.3 Задача лекции	319
47.4 Вариационная схема	320
47.5 Функция потерь	323
47.6 Решение средствами PennyLane	323
47.7 А как оно работает?	326
47.8 Что мы узнали?	328
48 Градиенты высших порядков	329
48.1 План лекции	329
48.2 Важность гейтов вращений	329
48.3 Производная от измерения	329
48.4 Parameter-shift для гейтов Паули	331
48.5 Обобщенный parameter-shift	331
48.6 Вторая производная и гессиан	332
48.7 Заключение	333
49 О блоке “Квантовый SVM”	335
50 Квантово-классический SVM	337
50.1 Описание лекции	337
50.2 VQC как ядерная функция	337
50.3 Пример реализации	339
50.4 Заключение	342
51 Классический SVM	343
51.1 Описание лекции	343
51.2 Линейная классификация	343
51.3 Интуиция метода опорных векторов	347
51.4 Метод опорных векторов для линейно-разделимой выборки	347
51.5 Метод опорных векторов для линейно-неразделимой выборки	348
51.6 Решение задачи метода опорных векторов	350
51.7 Ядерный переход	352
51.8 Плюсы и минусы SVM	353
52 О блоке “Квантовые нейронные сети”	355
53 Квантовые нейронные сети	357
53.1 Описание лекции	357
53.2 Введение	357
53.3 Применение	357

53.4	Архитектура	359
53.5	Анзац	359
53.6	Функция потерь	360
53.7	Сеть от начала до конца	360
53.8	Что мы узнали из лекции	361
54	Сверточные вариационные квантовые схемы	363
54.1	Описание лекции	363
54.2	Сверточные нейронные сети	363
54.3	Квантовая свертка	363
54.4	Подготовка модели и данных	365
54.5	Случайная квантовая цепь	367
54.6	Реализация квантовой свертки	367
54.7	Подготовка датасета	368
54.8	Гибридная квантово-классическая модель	370
54.9	Тренировка моделей	370
54.10	Сравнение результатов	371
54.11	Замечания	372
54.12	Что мы узнали из лекции	372
55	О блоке “Проблемы квантового машинного обучения”	373
56	Квантовая химия. Введение.	375
56.1	Описание лекции	375
56.2	Введение	375
56.3	Атом водорода	376
57	Квантовая химия. Теория самосогласованного поля.	379
57.1	Описание лекции	379
57.2	Теория самосогласованного поля	383
57.3	Что мы узнали?	389
58	Модель Изинга	391
58.1	Задача Изинга в одномерном случае	391
58.2	Задача Изинга как задача о максимальном разрезе в графе	392
58.3	Модель Изинга на чистом NumPy	394
58.4	Заключение	396
59	Точное аналитическое решение одномерной модели Изинга. Метод трансфер-матрицы.	397
59.1	Модель Изинга	397
59.2	Матричный подход	398
59.3	Решение одномерной модели	401
59.4	Заключение	402
60	Задачи комбинаторной оптимизации	403
60.1	Задача о максимальном разрезе в графе	403
60.2	Задача о выделении сообществ в графах	404
60.3	Задача о Гамильтоновых циклах	406
60.4	Задача коммивояжера	408
60.5	Задача о рюкзаке	409
60.6	Классические алгоритмы	410
60.7	Заключение	416
61	О блоке “От задач к квантовым алгоритмам”	417

62 Формулировка задач оптимизации в терминах модели Изинга	419
62.1 QUBO матрица	419
62.2 QUBO как квантовый гамильтониан	420
62.3 Поиск основного состояния	420
62.4 Статья “Ising formulations of many NP problems”	421
62.5 Задача о максимальном разрезе в графе (повторение)	421
62.6 Задача коммивояжера	422
62.7 Задача о выделении сообществ в графе	423
62.8 Заключение	424
63 Преобразование Жордана-Вигнера	427
63.1 Описание лекции	427
63.2 Введение	427
63.3 Спины, фермионы и бозоны	428
63.4 Вторичное квантование	428
63.5 Переход от спинов к фермионам	429
64 Проблема собственных значений	433
64.1 Введение	433
64.2 О проблеме (повторение)	433
64.3 Итеративные алгоритмы	433
64.4 Вариационные алгоритмы	436
64.5 Заключение	441
65 О блоке “VQE”	443
66 Variational Quantum Eigensolver	445
66.1 VQE	445
66.2 Интуиция алгоритма	446
66.3 Пример	446
66.4 Заключение	449
67 Практика VQE	451
67.1 Импорт библиотек	451
67.2 Случайный взвешенный граф	451
67.3 Решение задачи TSP в NetworkX	453
67.4 Матрица σ_{mod}^z	453
67.5 Гамильтониан задачи TSP	455
67.6 VQE на Pennylane	458
67.7 Симулляция	460
67.8 Заключение	463
68 О блоке “D-Wave – квантовый отжиг и семплирование”	465
69 D-Wave: квантовый отжиг, оптимизация и семплинг	467
69.1 Описание лекции	467
69.2 Введение в квантовый отжиг	467
69.3 Задачи оптимизации и семплирования	467
69.4 Квантовая физика под капотом	468
69.5 Пример: задача раскраски графа	470
69.6 Пример: задача электронной структуры молекулы	474
70 О блоке “Quantum Approximate Optimization Algorithm”	477
71 Quantum Approximate Optimization Algorithm	479

71.1	Введение	479
71.2	Пример задачи оптимизации	481
71.3	Quantum Alternating Operator Ansatz	484
72	Академия и индустрия	487
73	Список литературы	489
74	Список авторов курса	491
74.1	Создатели курса	491
74.2	Основные авторы	491
74.3	Основные рецензенты	492
74.4	Техническая поддержка	492
74.5	Редакторы	492
75	Благодарности	493
76	Глоссарий	495
76.1	А	495
76.2	Б	495
76.3	В	495
76.4	Г	495
76.5	И	496
76.6	К	496
76.7	М	496
76.8	Н	496
76.9	О	497
76.10	П	497
76.11	С	497
76.12	У	497
76.13	Ш	497
76.14	Э	498
Bibliography		499

Этот курс позволит вам погрузиться в удивительный мир квантового машинного обучения!

Почему именно этот курс?

Наш курс отличается от других курсов по квантовым вычислениям:

- он адаптивный и содержит лекции разных уровней сложности и глубины;
- он практический, а все объяснения подкрепляются кодом;
- он про реальные методы, которые будут актуальны ближайшие 10-15 лет.

Основатель курса [Семён Синченко](#):

Я очень люблю квантовые технологии и являюсь их большим энтузиастом. При этом моя основная работа связана с машинным обучением, поэтому идею сделать курс по QML я вынашивал очень давно. В сообществе ODS мне удалось найти единомышленников, и вместе мы подготовили для вас набор лекций и домашних заданий, которые проведут вас через удивительный мир кубитов, гейтов и квантовых нейросетей. Мы очень хотели сделать курс одновременно простым и понятным, но также строгим и глубоким с технической точки зрения и, надеюсь, у нас это получилось. В основном мы будем обсуждать темы, которые будут перспективны в ближайшие годы, а также максимально ориентироваться на практической стороне вопроса. Я лично верю, что квантовые технологии по мере их развития произведут не меньшую революцию, чем, например, распространение вычислений на видеокартах, а также позволят по-другому подойти к задачам, которые мы раньше не умели решать. Надеюсь, что после прохождения этого курса поверите и вы! В любом случае, я уверен, что тема QML и квантовых вычислений как минимум стоит того, чтобы с ней познакомиться. Надеюсь, что вам будет интересно!

Как устроен курс?

Наш курс разделен на логические блоки, каждый из которых содержит лекции разных уровней сложности:

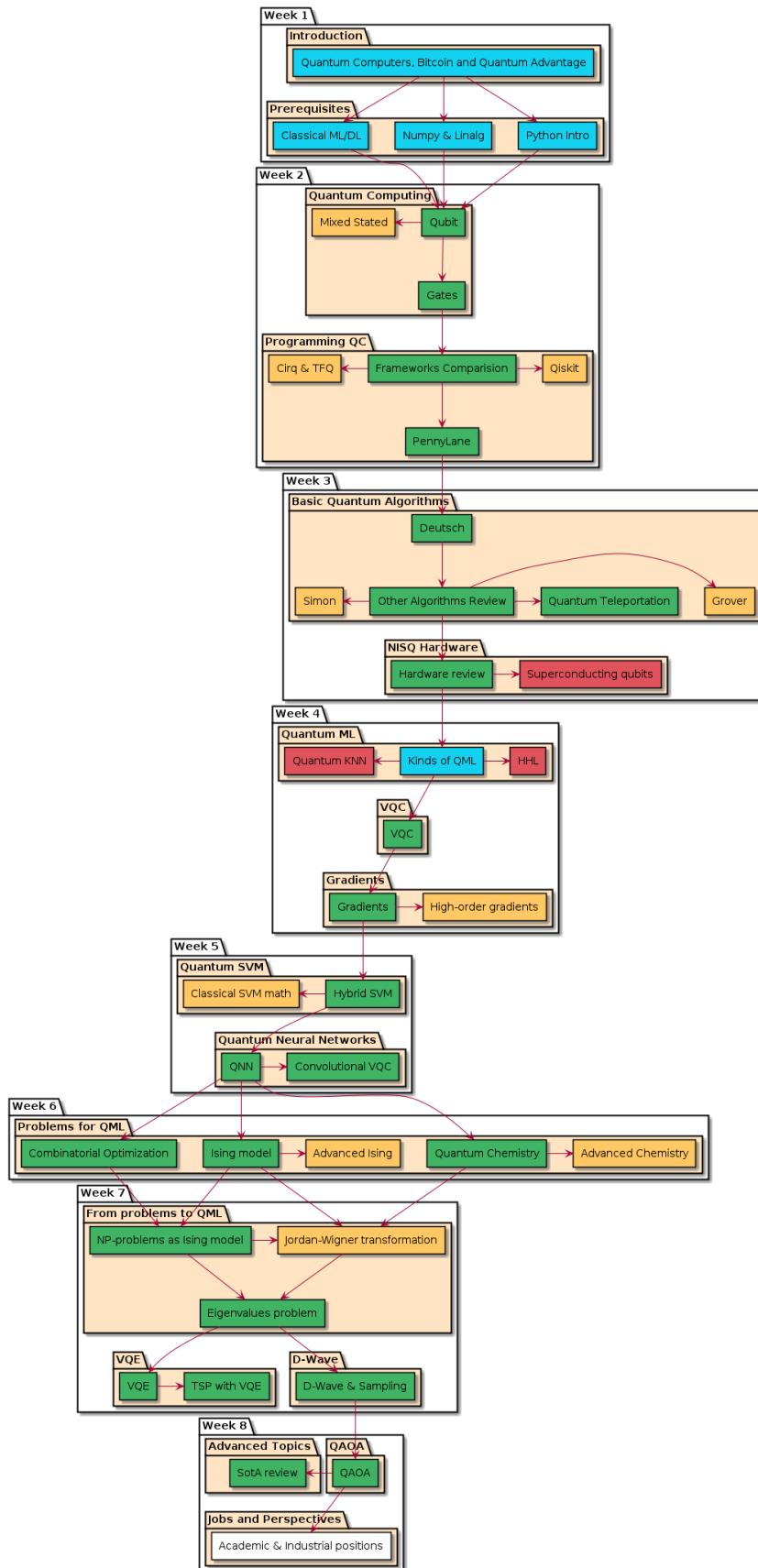
- **ГОЛУБОЙ** – вводные лекции;
- **ЗЕЛЕНЫЙ** – лекции “основного” блока курса;
- **ЖЕЛТЫЙ** – лекции, глубже раскрывающие темы блоков;
- **КРАСНЫЙ** – лекции про физику и математику, которая стоит за всем этим;
- **БЕЛЫЙ** – факультативные лекции.

Как проходить этот курс?

Рекомендуется проходить курс в том порядке, который обозначен на схеме. У курса есть [пререквизиты](#) – программирование на Python, основы линейной алгебры и машинного обучения, советуем с ними ознакомиться и (опционально) проверить себя с помощью [входного теста](#).

Курс обсуждается в [Slack ODS.ai](#) в канале #quantum_computing.

Как можно помочь?



Помочь можно словом и делом. Если знаете людей, интересующихся квантовыми вычислениями – замолвите словечко. Курс разрабатывается сообществом OpenDataScience, такими же «больными», как и организаторы (@sem @yorko @vtrohimenko). Если горите желанием в чем-то помочь, то создавайте Issue, а лучше сразу Pull Request в репозитории курса ([Contributing guide](#)).

Желаем успехов!

О КВАНТОВЫХ КОМПЬЮТЕРАХ, БИТКОИНЕ И ПРЕВОСХОДСТВЕ

1.1 Описание лекции

Эта лекция не несет в себе образовательного смысла, а лишь пытается ответить на вопросы, которые обычно возникают у тех, кто впервые сталкивается с темой квантовых вычислений. А именно:

- что это за вычисления такие?
- зачем вообще это все нужно?
- и когда взломают биткоин?
- что за превосходство, о котором все говорят?

1.2 Что это вообще за компьютеры такие?

1.2.1 Количественная эволюция компьютеров

Сегодня классические компьютеры, построенные на идеях Тьюринга, фон Неймана и Шокли, стали неотъемлемой частью нашей жизни. Все мы привыкли к тому, что с каждым годом наши компьютеры становятся все мощнее и мощнее. И то, что сегодня является бюджетным ноутбуком 15 лет назад было аналогом суперкомпьютера!

Так называемый закон Мура, сформулированный Гордоном Муром еще в конце 60-х годов, говорит нам о том, что число транзисторов на кристалле интегральной схемы будет удваиваться каждые два года. И этот закон стабильно выполняется.

1.2.2 Качественная эволюция компьютеров

Но, к сожалению, по сугубо физическим причинам, выполнение закона Мура не может длиться вечно — рано или поздно, но прямое увеличение количества транзисторов станет невозможным. Поэтому сегодня все чаще можно услышать слова о том, что современным компьютеры ждет качественная революция. Кто-то говорит о переходе на новые материалы для изготовления транзисторов. Кто-то говорит о создании транзисторов на новых принципах, например, об оптических компьютерах. Но часто можно услышать слова о том, что следующим революционным прорывом станет создание квантовых компьютеров. О них мы и будем говорить.

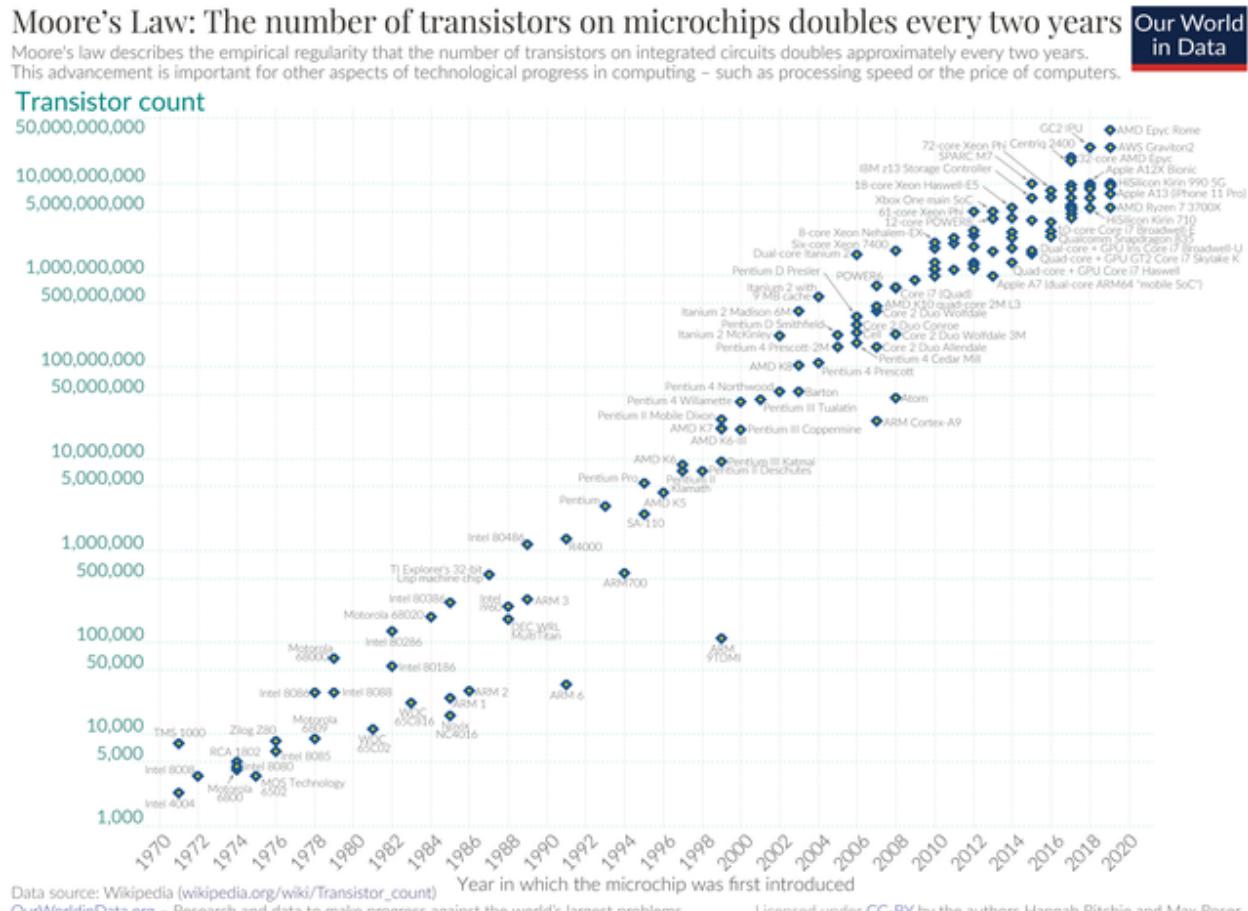


Fig. 1.1: Иллюстрация закона Мура — рост числа транзисторов с 1970-х

1.2.3 Идея о квантовом компьютере

Сегодня существует несколько версий о том, кто же первым высказал идею о квантовом компьютере. Как это часто бывает, сразу несколько ученых одновременно и независимо пришли к одной и той же идее. Одним из таких ученых был Ричард Фейнман.



Fig. 1.2: Ричард Фейнман, 1918-1988

В 1981-м году, когда шло очень активное развитие одновременно классических компьютеров и вычислительной квантовой механики, он высказал идею о том, что для решения задач квантовой физики нам нужен квантовый компьютер.

1.2.4 Что это за компьютер такой?

Этот вопрос на самом деле крайне сложный и именно ему будет посвящена первая половина нашего курса. Кажется странным, если вопрос, которому будет посвящено несколько полноценных лекций с формулами можно было бы раскрыть в одном абзаце. Было бы ошибкой пытаться сказать, что обычные компьютеры работают на законах классической физики, а квантовые на основе квантовой механики — ведь нормально объяснить работу транзистора можно лишь с привлечением уровня Ферми и прочих сложных штук из квантовой физики. Также неправильно было бы говорить о том, что в отличии от классических компьютеров, где есть лишь 0 и 1 в квантовых есть все состояния сразу. Ведь ничего не мешает сделать так называемую вероятностную машину Тьюринга, другими словами, классический компьютер, который оперирует многими состояниями сразу. Особенно не хочется сразу сыпать на читателя кучу непонятных терминов, типа квантовой суперпозиции, кубита или запутанности, ведь для тех, кто не знает что такое квантовые компьютеры эти термины, вероятнее всего, тоже ничего не дадут. Для начала, давайте просто условимся, что квантовые компьютеры это, в отличии от фотонных, графеновых, или других перспективных “новых” компьютеров, не только использование новых материалов или технической базы, а еще и использование новой, отличной от заложенной Тьюрингом концепции вычислений, представления и обработки информации.

1.3 А зачем это вообще нужно?

1.3.1 Факторизация больших чисел

Мне кажется, что именно открытие алгоритма Шора для эффективного решения задачи факторизации послужило наибольшим толчком в популяризации квантовых вычислений. Именно после этого большое число специалистов устремилось в эту область, военные и корпорации начали вкладывать деньги, а журналисты стали писать о будущем крахе банковских платежей и вообще всего мира. Вероятно, алгоритм Шора является самым раскрученным квантовым алгоритмом.

Дело в том, что большая часть всей современной криптографии держится на одном простом предположении о невозможности эффективно решать задачу факторизации больших чисел. Ну или по простому, если у нас есть число, которое является произведением двух относительно больших простых чисел, то мы будем до бесконечности искать эти числа и скорее всего так и не найдем их. Но это для классического компьютера. А вот для квантового компьютера в 1994-м Питером Шором был предложен алгоритм, который решает эту задачу эффективно, за относительно короткий промежуток времени.



Fig. 1.3: Питер Шор, тот, кто переполошил весь мир своим алгоритмом

Именно этот алгоритм в будущем взломает биткоин и обрушит банковскую систему, но не все так плохо – развитие квантовых компьютеров подтолкнуло ученых в области криптографии к созданию новых, так называемых *пост-квантовых* алгоритмов шифрования, которые нельзя взломать за разумное время и на классическом, и на квантовом компьютере.

1.3.2 Комбинаторные и NP-трудные задачи

Помимо уже озвученных проблем с выполнением закона Мура, есть также и другая проблема. А именно то, что существуют задачи, которые скорее всего никогда не получится эффективно решать на классическом компьютере Тьюринга. Даже на фотонном или графеновом. Хороший пример это задача о рюкзаке. Когда у нас есть рюкзак ограниченного объема, а также есть много предметов разного веса и стоимости. И нам надо наполнить наш рюкзак так, чтобы предметы внутри него имели максимальную суммарную стоимость. Задача кажется легкой, но она относится к так называемым *NP-полным задачам*. Такие задачи, например, в случае большого рюкзака и набора предметов, невозможно точно решить за разумное время. Да и вообще их решить с приемлемой точностью, пусть даже *приближенно* это сегодня большая проблема!

Note: Здесь я не зря написал “скорее всего”. Дело в том, что этот вопрос является одним из [так называемых вопросов тысячелетия](#). Так, для известной задачи о наполнении рюкзака, мы не знаем сегодня эффективного алгоритма решения на классическом компьютере. Но мы также и не можем пока доказать, что такого алгоритма не существует. Ну то есть скорее всего такого алгоритма и правда не существует, а также скорее всего $P \neq NP$, но доказать это пока ни у кого не вышло. Но это скорее лирическое отступление.

Так вот, дело в том, что для квантовых компьютеров уже сегодня известны алгоритмы, которые позволяют потенциально эффективно, пусть и *приближенно* решать такие задачи на квантовом компьютере. Это задача коммивояжера, задача о рюкзаке, задача кластеризации графа и много других задач комбинаторной оптимизации. В нашем курсе будет целый блок, посвященный таким квантовым алгоритмам как *Variational Quantum Eigensolver* и *Quantum Approximate Optimization Algorithm*.

1.3.3 Симуляция квантовой механики

Это то, ради чего Фейнман предложил создать квантовые компьютеры. Это отдельная большая тема, где много квантовой механики. Ей будет посвящено сразу несколько отдельных лекций нашего курса. Но попробуем объяснить в двух словах, не вдаваясь в детали.

Дело в том, что задачи квантовой механики не получается решать аналитически. Казалось бы, в чем проблема, законы Ньютона уже для трех тел тоже аналитически не решаются, но это не мешает нам летать в космос, ведь такую задачу можно решить *численно*. Но ту приходит вторая проблема, а именно, что явно интегрировать уравнение Шредингера по времени, или, по простому, решать квантовую механику *численно* тоже вычислительно почти невозможно более чем для двух частиц.

Казалось бы, что нам с этого. Ведь квантовая механика это удел теоретиков. Но вот проблема, квантовая механика лежит в основе квантовой химии, а та, в свою очередь, лежит в основе вообще всей химии и таких ее прикладных направлений, как создание новых лекарств, разработка новых аккумуляторов для автомобилей Tesla и многоного другого. И сегодня мы вынуждены использовать лишь очень приближенные решения и концепции, точности которых часто не хватает.

Квантовые компьютеры в этом случае могут сделать реальный прорыв. Ведь в силу своей физической природы, квантовый компьютер идеально подходит для симуляции квантовой механики, а значит и решения столь важных сегодня задач из области разработки лекарств и дизайна новых материалов.



Fig. 1.4: Визуализация решения задачи коммивояжера – кратчайший путь, чтобы обехать 12 немецких городов – очень важная задача современной логистики



Fig. 1.5: Эрвин Шредингер, 1887-1961, создатель знаменитого уравнения и мема про кота

1.3.4 Машинное обучение и искусственный интеллект

За последние 10-15 лет машинное обучение достигло поистине небывалых высот в своем развитии. Многие задачи, решение которых силами компьютера, раньше казалось невозможным сегодня успешно решаются при помощи машинного обучения. Примеры таких задач это, например, игра в Go, различение пород чихуахуа по фотографии, распознавание лиц в видеопотоке, составление относительно осмысленных текстов и генерация картин в стиле Пикассо из простых фотографий. Но оно все еще очень далеко от возможностей человеческого мозга. Так, наиболее масштабные искусственные нейронные сети, по примерным оценкам, имеют сегодня размер, эквивалентный 15 миллионам нейронов, в то время как человеческий мозг имеет порядка 85 миллиардов! Вызывает вопросы также и скорость обучения современных нейронных сетей. Так, самые большие языковые модели сегодня обучаются неделями на кластерах из тысяч видеокарт, в то время как человек с его относительно скромными вычислительными возможностями учится говорить всего 2-3 года.

И тут тоже на помощь могут прийти квантовые компьютеры. В данном случае, квантовые аналоги нейронных сетей, а также их комбинации с классическими нейронными сетями уже сегодня показывают впечатляющие результаты. Так, есть работы, где показано, что 4 квантовых нейрона по своей выразительности эквивалентны классической искусственной нейронной сети с ~ 250 нейронами!

Именно квантовому машинному обучению, а также способам его применения и будет посвящена большая часть нашего курса. Мы постараемся рассмотреть все вопросы по этой теме, начиная от теории того, как можно строить квантовые алгоритмы машинного обучения и заканчивая тем, как их можно запрограммировать на современных языках квантового программирования. Если эта тема вам интересна, то этот курс точно для вас!

1.4 Ну и когда взломают биткоин?

Наверное это один из главных вопросов, которые возникают при чтении подобных статей. И ответим сразу: взломают нескоро, времени еще много, 10 лет точно есть.



Fig. 1.6: Биткоин, как и многие другие электронные средства вынуждены будут перейти на пост-квантовую криптографию

1.4.1 Сколько нужно кубитов под разные задачи?

Наверное сразу стоит оценить тот размер, который квантовый компьютер должен иметь для эффективного решения описанных выше задач. Примерно цифры такие:

- Алгоритм Шора и взлом современной криптографии (включая биткоин): $\sim 20 \cdot 10^6$ (20 миллионов) кубит
- Задачи оптимизации: $\sim 100 \cdot 10^3$ (100 тысяч) кубит
- Первые полезные задачи в квантовой химии: $\sim 1 \cdot 10^3$ (1 тысяча) кубит
- Квантовое машинное обучение: $\sim 100 - 500$ кубит

Это кстати одна из причин, почему наш курс посвящен по большей части именно квантовому машинному обучению.

1.4.2 Логические vs Физические кубиты

Есть еще такая проблема, что вся квантовая механика вероятностная. А еще, что квантовые компьютеры работают в области микромира и очень чувствительны к любым шумам извне. Это ведет к совершенно недопустимому уровню ошибок в вычислениях и их низкой детерминированности. Например, сегодня хорошим уровнем точности для квантовых компьютеров является 99% на одну операцию. Но ведь каждый алгоритм включает в себя сотни или даже тысячи операций! И тогда уровень ошибок становится совсем печальным.

Но есть и хорошие новости. Сегодня существует очень много классных алгоритмов коррекции ошибок, которые позволяют используя несколько физических кубит с высоким уровнем ошибок создать один логический кубит, имеющий очень низкий уровень ошибок. То есть программист будет писать код, который производит операции над одним кубитом, а на физическом уровне это будет операция над несколькими кубитами. В общем вопрос вполне решаемый. Вот только для создания одного качественного логического кубита может потребоваться до тысячи физических кубит! А те оценки, которые мы привели выше, они как раз про логические кубиты, то есть кубиты с очень высокой точностью операций на уровне классических компьютеров.

1.4.3 Сколько кубит есть сегодня?

Скажем сразу, сегодня уже существуют квантовые компьютеры. Вот только все производители, когда пишут о новом рекорде, имеют в виду чаще всего именно физические кубиты.

Есть квантовые компьютеры с разной архитектурой. Одни имеют больше кубит, но и более высокий уровень ошибок. Другие имеют низкий уровень ошибок, но их трудно масштабировать. Теме квантового железа в нашем курсе будет посвящен отдельный блок из нескольких лекций. Но если кратко, то можно назвать примерно такие цифры:

- рекорд в относительно легко масштабируемых, но шумных квантовых компьютерах это ~ 55 кубит
- рекорд в относительно точных, но медленных и плохо масштабируемых компьютерах это ~ 20 кубит
- рекорд в точных и масштабируемых, но очень трудно программируемых компьютерах это ~ 25 кубит

Note: Тут мы имеем ввиду соответственно:

- сверхпроводящие кубиты, которые сегодня проще всего масштабировать
- ионы в ловушках, которые имеют одну из самых высоких точностей
- фотоны, которые вроде всем хороши, кроме того, что на них программирование это юстировка линз и лазеров на оптическом столе

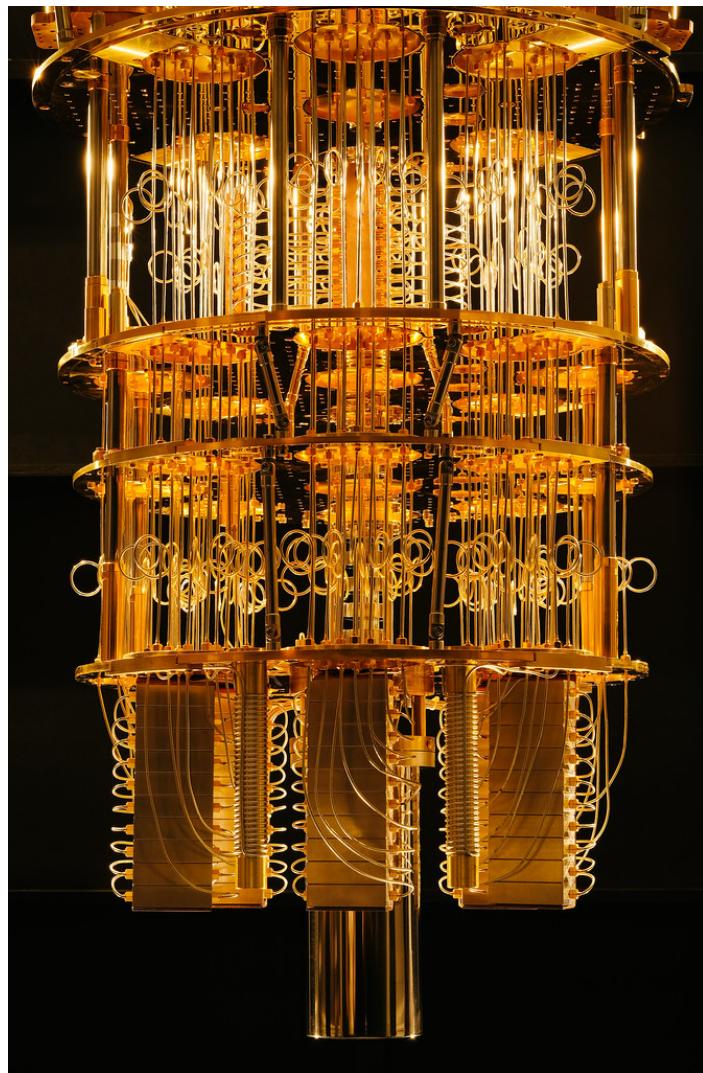


Fig. 1.7: Квантовый компьютер компании IBM выглядит примерно так

Стоит добавить, что рекорд в точных и масштабируемых, а также программируемых (топологических) кубитах сегодня это ровно 2 кубита. Серьезно, взаимодействие двух логических кубит было опубликовано в *Nature* в этом году.

1.4.4 Какие планы имеют ведущие игроки на этом рынке?

Казалось бы, с такими масштабами биткоину боятся нечего, да и в целом область выглядит не самой перспективной. Но есть один нюанс. Все крупные игроки на рынке создания квантовых компьютеров (*Google Quantum*, *IBM Quantum*, *IonQ*, *Xanadu*) озвучили планы к 2030-му году иметь порядка одного миллиона физических кубит, что эквивалентно порядка тысячи логических кубит. Для криптографии это еще не страшно, но вот многие полезные задачи уже можно будет попробовать решать. Ну и стоит еще раз посмотреть на график закона Мура для классических компьютеров, которые каждые десять лет показывают примерно такой же прогресс!

1.5 О квантовом превосходстве

Очень часто можно услышать разговоры о том, что достигнуто или опровергнуто квантовое превосходство. Попробуем под конец лекции разобраться, что же это такое и почему это важно (или не важно).

1.5.1 Понятие квантового превосходства

Само понятие было сформулировано еще в 2012-м году известным физиком теоретиком Джоном Прескиллом.

Квантовое превосходство это решение на квантовом компьютере задачи, которую нельзя решить на классическом компьютере за разумное время (10 тысяч лет разумным временем не считается). Достижение квантового превосходства это однозначно новый уровень в развитии квантовых вычислений. Но есть один подвох. Дело в том, что речь идет о совершенно любой задаче, независимо от того, насколько она полезна или бесполезна.

Так что когда кто-то заявляет о достижении квантового превосходства, то это важный повод для ученых и разработчиков квантовых компьютеров, но скорее всего это очень малозначимый факт, с точки зрения простого обывателя.

1.5.2 Хронология событий

Ну и в конце приводим краткую хронологию событий.

- 2019 год, компания *Google* заявляет о достижении квантового превосходства. Задача выбрана максимально удобная для квантового компьютера и полностью лишенная практического смысла. По словам разработчиков из *Google* их квантовый компьютер за 4 минуты решил задачу, которую классический суперкомпьютер решал бы 10 тысяч лет. Их квантовый компьютер имел 54 кубита;
- 2019 год, компания *IBM* заявляет, что *Google* не учли, что их задачу можно решать на классическом компьютере более оптимально, но без экспериментов;
- 2020 год, компания *Alibaba* реализует алгоритм *IBM* на своем суперкомпьютере и решает задачу за ~ 20 дней;
- 2021 год, группа китайских ученых оптимизирует классический алгоритм и решает задачу на 60 видеокартах *NVIDIA* за 7 дней;



Fig. 1.8: Джон Прескил, который и придумал этот термин. Еще он известен своим знаменитым пари с другим физиком Стивеном Хокингом (которое Хокинг проиграл)

- 2021 год, группа других китайских ученых заявляет, что достигла нового превосходства на квантовом компьютере из 56 кубит;

В общем сейчас идет довольно интересный процесс войны меча и щита. Пока одни ученые строят более мощные квантовые компьютеры, другие придумывают более продвинутые алгоритмы их симуляции. Хотя конечно все ученые говорят, что уже где-то на 60-70 кубитах эта история окончательно закончится в пользу квантовых компьютеров.

1.6 А как это вообще выглядит? И сколько стоит?

На сегодня почти все известные технологии создания квантовых компьютеров требуют чего-то из:

- сверхнизкие температуры
- сверхвысокий вакуум
- сверхточная юстировка лазеров на оптическом столе

Или даже всего сразу. Поэтому сегодня почти все квантовые компьютеры продаются через облачные сервисы. Например, относительно недавно ведущий поставщик облачных технологий – компания *Amazon* добавила в свой сервис AWS новый продукт *Amazon Braket*. Этот продукт позволяет взять в аренду самый настоящий компьютер точно также, как мы привыкли брать в аренду процессоры, видеокарты или жесткие диски. Аналогичные продукты сейчас предоставляют и другие крупные игроки на рынке облачных услуг. Хотя это все пока исключительно для целей исследования. Ведь как мы уже поняли, сегодня квантовые компьютеры еще не способны решать реальные задачи. Стоит такое развлечение не очень дорого, например, можно запустить свою квантовую программу на 32-х кубитном компьютере *Aspen-9* всего за \$0.3 (per-task).

Некоторые производители идут дальше и предлагают относительно компактные решения. Так, недавно было представлено 24-х кубитное решение, которое помещается в две стандартных серверных стойки. Но масштабируемость таких устройств вызывает вопросы.

В любом случае, в ближайшие 15-20 лет точно не стоит ждать появление карманного квантового компьютера, или хотя бы квантового сопроцессора в домашнем ПК. Да и в этом нет особого смысла, ведь мало кому дома нужно взламывать биткоин, решать логистическую проблему или разрабатывать высокотемпературный сверхпроводник.

1.7 Заключение

Это вводная лекция, она не даст вам каких-то особых знаний. Скорее, ее цель заинтересовать читателя. Самое интересное будет в основной части курса, где мы будем разбирать квантовые алгоритмы, пытаться симулировать квантовую механику и обучать самые настоящие квантовые нейросети! Ждем вас на курсе!

ПРЕРЕКВИЗИТЫ ДЛЯ ПРОХОЖДЕНИЯ КУРСА QML

Настоящий курс совмещает в себе как теорию для глубинного понимания материала, так и большое количество практики для закрепления навыков и проработки неочевидных ситуаций/проблем. Поэтому предполагается, что некоторые базовые знания у вас уже есть. Проверить себя можно с помощью опционального [входного теста](#), который поможет понять, нужно ли проходить вводные “голубые” лекции – по Python, математике и машинному обучению (15 вопросов всего, по 5 на блок). Тест довольно простой, так что если вы его прошли без ошибок или почти без ошибок – это мало о чём говорит, все равно дальше может быть сложно. А вот если ошибок много, тогда вам стоит обратиться ко вводным “голубым” блокам.

2.1 Python, Poetry и среда разработки

Для успешного прохождения курса вы должны уметь делать и понимать такие вещи, как:

- копирование файлов с GitHub;
- установка Poetry и библиотек Python;
- как запустить Jupyter Notebook;
- основные сочетания клавиш в Jupyter Notebook;
- как запускать код в Jupyter Notebook;
- какие простые типы данных существуют в Python;
- как работать со сложными типами-контейнерами (`list`, `dict`, `tuple`), включая самые популярные методы каждого из них;
- как работает арифметика Python и как переменные разных типов соотносятся друг с другом;
- что такое условные конструкции и ветвление логики;
- что такое блок кода в Python и зачем нужны отступы (табуляции);
- что такое циклы и как их можно использовать применительно к сложным типам данных;
- как писать функции в Python, что такое аргументы функции и какими они бывают;
- что такое декоратор и как его использовать;
- базовый ООП: объявление классов, понятие объекта класса, умение писать методы класса.

Если какая либо из тем вызывает у вас непонимание, то настоятельно рекомендуется освежить знания (или даже их приобрести), пройдя соответствующие уроки в блоке “Основы Python”. Даже если у вас есть опыт работы с ним и написания кода, то предлагается ознакомиться с процессом установки всех необходимых зависимостей и библиотек (урок 2 блока), а также горячими клавишами и разнообразными функциями в Jupyter Notebook (урок 3).

2.2 Линейная алгебра и NumPy

Для понимания базовых квантовых алгоритмов и алгоритмов квантового машинного обучения, как и в классическом машинном обучении, не обойтись без линейной алгебры. Если есть время взяться за математику по-серьезному, рекомендуем курсы [Gilbert Strang](#) или [Д.В. Беклемишева](#), а также [задачник А.И. Кострикина](#). А если Вы это когда-то изучали линейную алгебру, но подзабыли ее, то поможет блок нашего курса. Хотется, чтобы у Вас не было проблем с такими понятиями как:

- Вектор-строка и вектор-столбец
- Линейная зависимость векторов
- Системы линейных уравнений
- Матрицы, определитель и след матрицы
- Линейный оператор
- Комплексное и эрмитово сопряжение
- Скалярное и векторное произведения
- Гильбертово пространство

Список явно не исчерпывающий и скорее всего явно так и не перечислишь все то, что может встретиться по мере изучения квантовых алгоритмов.

2.3 Машинное обучение

Наконец, чтобы говорить о квантовом машинном обучении, надо иметь неплохое представление о собственно машинном обучении. [Блок нашего курса](#) рассказывает о том, что общего у разных задач машинного обучения и как из основных компонентов, подобно пазлу, складываются различные применения моделей машинного обучения. Этот блок, впрочем, не заменяет хорошей книги или курса по машинному обучению (как [курс Andrew Ng](#), [специализация Яндекса и МФТИ](#) или [mlcourse.ai](#)), покрывающего такие темы как:

- обучение с учителем и без учителя, обучение с подкреплением
- метрики качества в задачах обучения с учителем
- функции потерь и сведение задач машинного обучения к задачам оптимизации
- обработка и построение признаков
- оценка качества модели и кросс-валидация
- алгоритмы классификации и регрессии: деревья решений, лес, бустинг, логистическая регрессия и др.
- алгоритмы ранжирования, метрики качества ранжирования
- обучение без учителя: снижение размерности, кластеризация, поиск аномалий
- байесовский взгляд на машинное обучение
- обучение представлению

Опять же, это не исчерпывающий список, к тому же покрывающий только “классику”, игнорируя глубокое обучение. Но если что-то из перечисленного вызывает недоумение или если со входным [тестом](#) не задалось, возможно, стоит обратиться к упомянутым курсам.

CHAPTER
THREE

О БЛОКЕ “PYTHON”

Этот блок включает в себя:

- описание процесса установки среды разработки для прохождения курса;
- описание синтаксиса языка программирования Python;
- разбор типовых синтаксических конструкций при программировании на Python;
- примеры анализа возникающих ошибок, помогающие в дальнейшем прохождении курса.

ВВОДНАЯ ЛЕКЦИЯ ПРО PYTHON

Добро пожаловать во вводный блок курса! Если у вас есть опыт программирования на Python, рекомендуем ознакомиться *с пререквизитами курса* – так вы сможете понять, какие разделы стоит изучить или освежить в памяти перед прохождением курса. Тест можно игнорировать, если вы только начинаете свое знакомство с Python.

Итак, Python – это такой язык программирования, который позволяет сообщить компьютеру о том, что нужно сделать, чтобы достичь некоего результата. За последнее десятилетие он получил быстрое распространение и сейчас является одним из самых популярных языков программирования в мире. Входной порог для его использования достаточно низок: вы можете использовать Python для решения своих задач даже если никогда не имели дела с программированием.

4.1 Что такое Python?

Python – это язык программирования *общего назначения*, используемый во многих приложениях. Например:

- разработка веб-приложений;
- создание игр;
- продвинутая аналитика данных, в том числе с использованием нейронных сетей;
- компьютерная графика;
- геофизика;
- психология;
- химия;
- теория графов.

Python используют практически все крупные компании, о которых вы слышите каждый день: Google, Yandex, YouTube, Dropbox, Amazon, Facebook ... список можно продолжать часами.

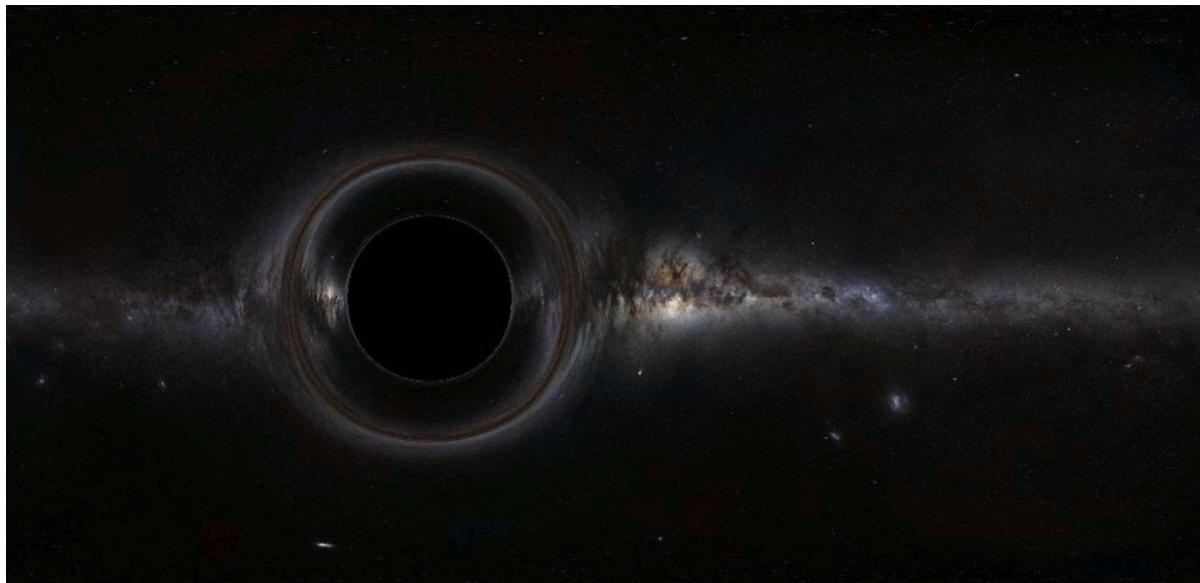


Fig. 4.1: Рассчитанная в Python симуляция преломлений света черной дырой

4.1.1 Чем примечателен Python?

В основе разностороннего применения и популярности лежит **простота изучения**: все чаще люди начинают свой путь в программировании с Python, поскольку он очень **дружелюбен к новичкам** и позволяет максимально быстро перейти к решению целевой задачи.

Сюда же можно отнести **многообразие библиотек** (или *расширений функциональности*, то есть кода, написанного другими людьми, который вы можете переиспользовать). Хотите изучить физику небесных тел и симулировать их взаимодействия? Можно найти и скачать библиотеку, позволяющую за один вечер провести вычисления, о которых в прошлом веке можно было лишь мечтать. Хотите создать прототип мобильного приложения? И на этот случай есть библиотека. Вам нравится квантовая физика и вы хотите использовать ее вместе с умными компьютерными алгоритмами? Что ж, тогда вы снова по адресу.

Python – это **высокоуровневый язык для быстрой разработки и/или прототипирования**, на нем очень удобно проверять гипотезы и идеи. “Высокоуровневый” означает, что вам не нужно вникать в устройство компьютера и тонкости взаимодействия с ним, чтобы перейти к задаче. Многое «сделано за вас»: вы работаете с простыми *абстракциями* (или удобными представлениями), а не боретесь с компьютером из-за непонимания сложностей его устройства.

Еще один плюс в копилку популярности языка – это **элегантность и краткость синтаксиса** (принципов написания кода, как будто это абзацы в тексте или колонки в газете). Вместе с вышеупомянутым обилием библиотек вы можете буквально за 5 минут и 10 строк кода – а это меньше половины листа А4 – воспроизвести научную статью, в которую вложено несколько человеко-лет. А еще такой синтаксис делает **код легким для чтения, запоминания и понимания**.

Стоит отметить, что Python – это **интерпретируемый** язык, а значит, компьютер каждый раз перед выполнением программы читает код строчку за строчкой и определяет (интерпретирует), что нужно сделать дальше, не проводя никаких оптимизаций и предварительных расчетов. Это негативно влияет на общую скорость работы: Python является одним из самых медленных языков. Тем не менее он отлично подходит для академических целей, например, исследовательской работы или других задач, где скорость работы не является критически важной. Настоящая сила Python заключается в том, что это **“язык-клей”**: он обеспечивает удобный доступ к различным библиотекам, написанным на высокоеффективных языках, например, на C/C++, Fortran, CUDA C и других.

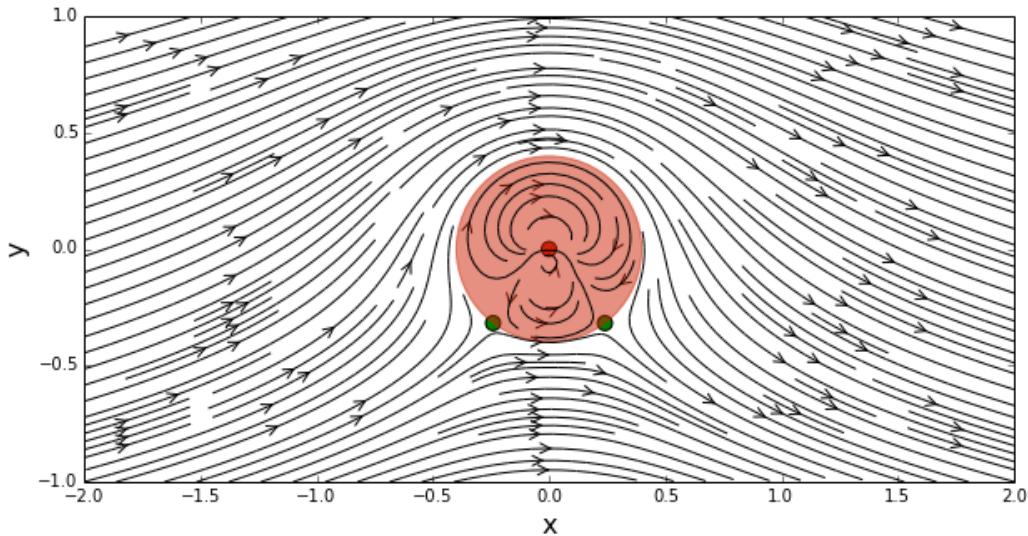


Fig. 4.2: Пример моделирования аэродинамики в Python с помощью библиотеки AeroPython

4.1.2 И в чем подвох?

В простоте языка и его доступности для быстрого старта таится одна из проблем: вы *можете не понимать, что происходит внутри*, поэтому иногда бывает сложно разобраться в причинах ошибок и неточностей, возникающих по ходу работы над задачей. В целом к Python применим следующий принцип: “**Easy to learn, hard to master**”. Возвращаясь к примеру элегантности кода, когда 10 строк кода выполняют всю работу: важно понимать, что за ними стоят еще *сотни* или даже *тысячи строк кода*, а это может приводить к ситуациям, когда поиск ошибки в минимальном наборе команд растягивается на несколько дней.

4.2 Но не пугайтесь!

В данном блоке мы постараемся дать вам всю необходимую интуицию и теорию для успешного прохождения настоящего курса, ответим на основные вопросы, покажем типовые примеры использования Python и разберем классические ошибки.

Дополнительно отметим, что Python хорош и для квантового машинного обучения (QML), ради которого весь курс и затяян, в особенности – для классического машинного обучения (ML). В области ML этот язык программирования стал де-факто стандартом, который используют практически все специалисты.

4.2.1 Интересные факты

Еще немного дополнительной информации про Python:

- Разработчик языка Гвидо ван Россум назвал его в честь популярного британского комедийного телешоу 1970-х «Летающий цирк Монти Пайтона».
- Актуальной версией Python считаются версии 3.6 и выше (3.7, 3.8.12...). Долгое время (до 2020 года существовал Python 2, который ныне не поддерживается и не обновляется. Если вы видите кусок кода на Python 2 и вам предстоит работать с ним, возможно, сначала придется его переписать, хотя

большая часть кода имеет совместимость и работает корректно. В этом курсе мы не будем изучать Python 2.

- У Python огромное сообщество: большинство проблем, с которыми вы можете столкнуться, уже было озвучено и даже решено. Это означает, что используя поисковик, вы можете решить практически все проблемы в течение 10-30 минут. Главное – научиться правильно формулировать свои вопросы.
- При работе с Python следует придерживаться принципа “должен существовать один и, желательно, только один очевидный способ сделать это”. Другие принципы (Дзен Питона) на русском языке – [по ссылке](#).
- Python – это *открытый проект*, в который каждый может внести изменения (но они должны быть предварительно одобрены), например, [тут](#).
- Есть целый набор рекомендаций и предложений по улучшению кода (PEP, Python Enhancement Proposals). Они содержат указания на то, как следует писать код и чего стоит избегать, а также дискуссии о будущих изменениях в языке.
- Язык постоянно развивается, в нем появляются новые возможности, улучшается производительность (скорость выполнения).
- Сборник всех существующих в открытом доступе библиотек [находится тут](#).
- Если вы столкнетесь с багом (системной ошибкой, вызванной внутренним механизмом языка), то сообщить об этом можно [на специальном сайте](#).



Fig. 4.3: Кот для привлечения внимания и в благодарность за то, что вы начали проходить курс и сделали самый сложный шаг – прошли первую лекцию!

УСТАНОВКА НЕОБХОДИМЫХ ИНСТРУМЕНТОВ И НАСТРОЙКА СРЕДЫ

5.1 Описание лекции

Эта лекция расскажет:

- как скачать курс себе;
- как собрать свой экземпляр книги;
- как скачать сотню библиотек за раз;
- как проверить, корректно ли прошла установка.

5.2 Git и получение файлов курса

Git - это система контроля версий. Этот инструмент часто используется для совместной разработки программ (и не только!) группой людей, каждый из которых работает над своей отдельной проблемой. Так, например, этот курс был создан с помощью Git, и каждый автор работал с отдельной копией, создавая набор лекций. А затем все копии собрались в одну книгу, которую вы и читаете. Для вашего удобства, а также возможности получить (“собрать” на сленге) свой экземпляр книги (если он вам зачем-то нужен – ведь все доступно онлайн в полном объеме). К сожалению, знакомство с этим инструментом не входит в программу настоящего курса, и единственное, что вам потребуется - это скачать файлы себе для дальнейшей работы. Хороший туториал на английском можно найти по [этой ссылке](#).

Однако сейчас ограничимся лишь простой операцией: для этого надо зайти на страницу с репозиторием (папкой) по [ссылке на github](#) (одну из разновидностей Git). Затем найдите зеленую кнопку с текстом <> Code и кликните по ней. В открывшемся окне убедитесь, что выбрана вкладка HTTPS, а не SSH или GitHub CLI. В самом низу окошка будет кнопка Download ZIP - она нам и нужна, жмите!

Tip: Если вы проследуете инструкциям из документации по установке Git, то действия выше можно проделать одной командой из консоли: `git clone https://github.com/SemyonSinchenko/qmlcourse`. Вообще Git - очень полезный инструмент, и рекомендуется уделить время прочтению инструкций по его установке и настройке, это пригодится в будущем.

Затем скачанный файл необходимо разархивировать любым удобным способом. В современных версиях Windows можно выбрать соответствующий пункт в выпадающем меню при клике на скачанный архив правой кнопкой мыши, на Mac OS достаточно просто кликнуть дважды, а на Ubuntu ... что вы делаете на этой странице? Нужно поставить программу, если ее еще нет, а затем разархивировать по любому удобному пути: `sudo apt install unzip && unzip qmlcourse-master.zip -d <путь/к/вашей/папке/>`.

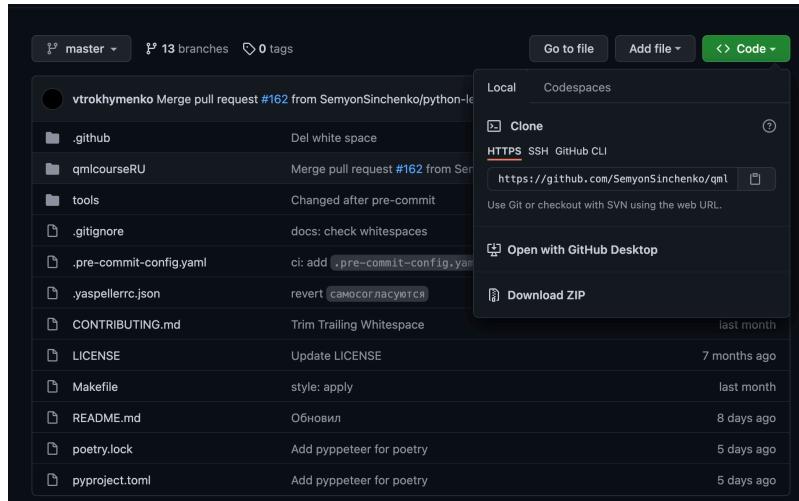


Fig. 5.1: Не пугайтесь темного цвета - это просто цветовая схема в браузере. На светлой странице все, что написано выше, применимо в полном объеме.

Теперь в папке `qmlcourse-master` вы увидите содержимое курса, над которым трудилась наша команда: это и картинки, и текст, и задания.

5.3 Что делать дальше

В этом блоке мы сделаем две вещи: соберем книгу и установим на ваш компьютер все необходимое для прохождения курса, от Python и его библиотек до специальных программ, которые занимаются отслеживанием этих самых библиотек. Буквально после пары строк на вашем компьютере появится сотня библиотек, которыми мы хвастались в первой лекции – они позволят как проводить математические расчеты, так и писать программы для квантовых компьютеров. Итак, дальнейшие шаги зависят от вашей операционной системы:

5.3.1 Инструкция для Windows

Для дальнейшей работы необходимо установить менеджер пакетов для Windows - chocolatey. Официальная инструкция находится [по этой ссылке](#). В данной инструкции рассмотрена установка без администраторских прав, то есть вы можете установить все программы даже на рабочий ноутбук или компьютер - дополнительных соглашений или паролей для установки не требуется.

Итак, необходимо перейти по ссылке выше и скопировать код файла `ChocolateyInstallNonAdmin.ps1`. Создайте где-нибудь (хоть на рабочем столе) пустой файл с указанным именем, вставьте в него скопированное содержимое.

Tip: Важно, чтобы расширение файла было именно `ps1`. Если вы не знаете, как создавать файлы произвольных расширений, то картинка ниже поможет.

Если вы все сделали правильно, то иконка файла изменится:

Поздравляем, вы создали исполняемый файл! В своем роде это скрипт, который и запустит установку. Для этого нужно открыть PowerShell (терминал в рамках операционной системы Windows). Его вы можете найти очень просто - достаточно набрать название в меню “Пуск”. После этого выполните команду (скопируйте, вставьте и нажмите Enter):

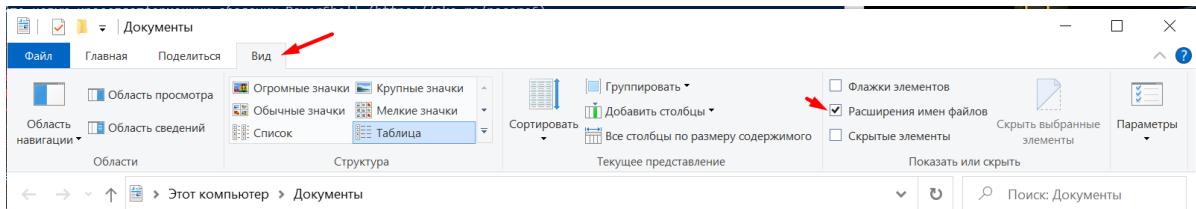


Fig. 5.2: Необходимо зайти в “Проводник” (любая папка с документами), затем перейти во вкладку “Вид”, после чего поставить галочку у соответствующего пункта. Теперь при создании файла, к примеру, `my_file.abc` расширение будет `abc`, а не `.txt` (хотя файл может оставаться текстовым). Такой файл можно открыть через блокнот (нажав правой кнопкой мыши и выбрав программу для запуска).

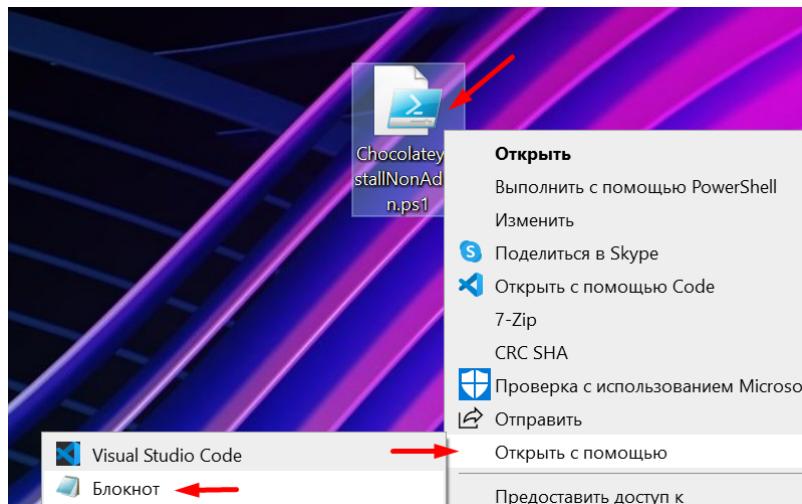


Fig. 5.3: Иконка изменилась (она соответствует PowerShell, см. ниже). Файл можно редактировать через блокнот, как показано выше.

```
Set-ExecutionPolicy Bypass -Scope Process -Force;
```

После этого перетащите в терминал файл (прямо мышкой с рабочего стола) и отпустите левую кнопку: в терминале должен появиться путь до файла. Снова выполните команду (**Enter**). Дождитесь окончания установки.

И, наконец, закончим установкой вспомогательных инструментов. Запустите команду

```
choco install make curl miniconda3
```

Все сопроводительное обеспечение установлено, осталось непосредственно собрать окружение и собственный экземпляр книги - в этом нам поможет запуск двух команд (**но перед этим нужно перейти в директорию со скачанным курсом** . Для этого нужно воспользоваться командой `cd` (change directory, сменить директорию) и указать путь к директории. Самый простой способ это сделать - перетащить папку из окна проводника с файлами (или рабочего стола) в терминал: путь подставится автоматически. Пример правильной команды: `cd /home/ivan.petrov/Downloads/course/qmlcourse-master/`):

```
make install-windows  
make build-windows
```

5.3.2 Инструкция для MacOS

Для дальнейшей работы необходимо установить менеджер пакетов для MacOS - `brew`. [По ссылке](#) вы найдете инструкцию и более детальное описание, чем является `brew`. Вам необходимо выполнить в терминале (который доступен как приложение на экране со всеми приложениями – просто начните вводить `term` и поиск подскажет верный вариант) команду:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Attention: Скорее всего, для выполнения этой команды, нужно задействовать команду `sudo`, это вполне нормальная история.

Затем выполнить простую команду (тоже в терминале), предварительно перейдя в папку с курсом, которую вы извлекли из архива. Для этого нужно воспользоваться командой `cd` (change directory, сменить директорию) и указать путь до папки. Самый простой способ это сделать - перетащить папку из окна Finder в терминал: путь подставится автоматически. Например:

```
cd /home/ivan.petrov/Downloads/course/qmlcourse-master/
```

И из этой папки вызовите последовательность из двух команд:

```
make install-macos-latest  
make build-linux-macos
```

5.3.3 Инструкция для Ubuntu

Эта часть подойдет также для всех “родственных” систем: Debian, Linux Mint, Deepin Linux (который можно встретить на ноутбуках компании Huawei, к примеру) и многие другие системы, использующие менеджер пакетов apt. Для начала необходимо установить дополнительные программы. Выполните код ниже в терминале:

```
sudo apt-get update
sudo apt-get install build-essential -y
sudo apt-get install git wget -y
```

Attention: При использовании sudo терминал может попросить вас ввести пароль для установки и обновления пакетов программ. Это абсолютно нормально.

Затем выполнить простую команду (тоже в терминале), предварительно перейдя в папку с курсом, которую вы извлекли из архива. Для этого нужно воспользоваться командой cd (change directory, сменить директорию) и указать путь до папки. Самый простой способ это сделать - перетащить папку из окна Finder в терминал: путь подставится автоматически. Например:

```
cd /home/ivan.petrov/Downloads/course/qmlcourse-master/
```

И из этой папки вызовите последовательность из двух команд:

```
make install-ubuntu-latest
make build-linux-macos
```

Hint: Первая команда установит Python, `poetry` - это еще один менеджер, который отвечает за управление библиотеками Python и их версиями, а также сами библиотеки, необходимые для прохождения курса; вторая – соберет `jupyter book`, то что по ссылке https://semyonsinchenko.github.io/qmlcourse/_build/html/book/index.html (так что можете не выполнять если не хотите, трава и так будет расти)

5.4 Проверка корректности установки

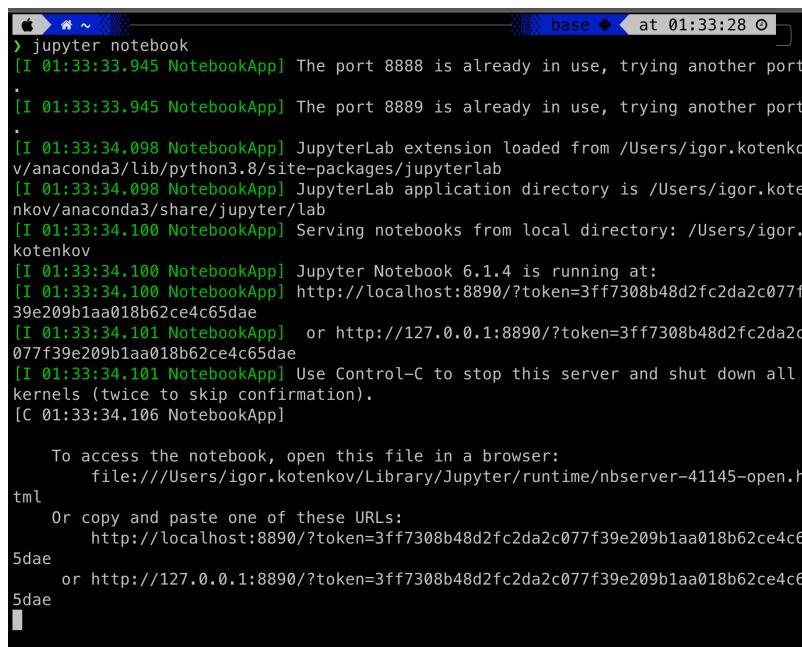
Чтобы убедиться в том, что все отработало штатно, и у вас полный комплект, запустим очень мощный инструмент для работы в машинном обучении - Jupyter Notebooks. С ним вы познакомитесь ближе в следующей лекции, а пока введите в терминале команду

```
poetry run jupyter notebook
```

Hint: Для того, чтобы запустить что-либо в текущем окружении, вам нужно начинать писать команду с `poetry run` (как в примере выше), ибо все ваши установленные библиотеки (которые в свою очередь устанавливаются как `poetry add <name_lib>`) не будут подгружаться

В терминале должна появиться надпись о том, что запущен сервер, а также будет предоставлено две или три ссылки.

Перейдите по одной из ссылок, в которых есть слово `token` – они начинаются с `http://localhost` или `http://127.0.0.1`. В вашем браузере откроется страница с обзором папки, из которой был запущен Jupyter Notebook. Если вы в дальнейшем хотите хранить все свои результаты в другой папке, то



```

> jupyter notebook
[I 01:33:33.945 NotebookApp] The port 8888 is already in use, trying another port
.
[I 01:33:33.945 NotebookApp] The port 8889 is already in use, trying another port
.
[I 01:33:34.098 NotebookApp] JupyterLab extension loaded from /Users/igor.kotenkov/anaconda3/lib/python3.8/site-packages/jupyterlab
[I 01:33:34.098 NotebookApp] JupyterLab application directory is /Users/igor.kotenkov/anaconda3/share/jupyter/lab
[I 01:33:34.100 NotebookApp] Serving notebooks from local directory: /Users/igor.kotenkov
[I 01:33:34.100 NotebookApp] Jupyter Notebook 6.1.4 is running at:
[I 01:33:34.100 NotebookApp] http://localhost:8890/?token=3ff7308b48d2fc2da2c077f39e209b1aa018b62ce4c65dae
[I 01:33:34.101 NotebookApp] or http://127.0.0.1:8890/?token=3ff7308b48d2fc2da2c077f39e209b1aa018b62ce4c65dae
[I 01:33:34.101 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 01:33:34.106 NotebookApp]

To access the notebook, open this file in a browser:
file:///Users/igor.kotenkov/Library/Jupyter/runtime/nbserver-41145-open.html
Or copy and paste one of these URLs:
http://localhost:8890/?token=3ff7308b48d2fc2da2c077f39e209b1aa018b62ce4c65dae
or http://127.0.0.1:8890/?token=3ff7308b48d2fc2da2c077f39e209b1aa018b62ce4c65dae

```

Fig. 5.4: Примерно так будет выглядеть терминал со ссылками.

перед запуском команды с помощью уже указанной инструкции в терминале `cd <путь/до/директории>` перейдите к ней.

Для создания нового файла – блокнота, как его еще называют (почему? узнаете в следующей лекции) – кликните по кнопке `new` в правом верхнем углу, а затем – по `Python 3` (возможно будет какая-нибудь приставка, это штатно).



Fig. 5.5: Таким образом можно создать новый файл с кодом.

Для завершения проверки скопируйте код ниже в тетрадку в браузере, а затем нажмите `CTRL + Enter` или `command + Enter` (это заставит код выполниться, подробнее дальше в курсе). Если увидите график - то все в полном порядке!

```

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Fixing random state for reproducibility
np.random.seed(19680801)

# Compute pie slices
N = 20
theta = np.linspace(0.0, 2 * np.pi, N, endpoint=False)

```

(continues on next page)

(continued from previous page)

```
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)
colors = plt.cm.viridis(radii / 10.)

ax = plt.subplot(111, projection='polar')
ax.bar(theta, radii, width=width, bottom=0.0, color=colors, alpha=0.5)

plt.show()
```

Не переживайте, этот код не нужно разбирать сейчас – мы просто убеждаемся, что все работает согласно задумке. Если что-то не так, пересмотрите все ли сделали согласно инструкции; если да и не воспроизводится, то задавайте вопрос в [канале курса](#)

5.5 Что мы узнали в лекции

- что такое Git и как скачать проект на локальный компьютер;
- как становить необходимые библиотеки, Python и прочие прелести;
- как запустить Jupyter Notebook для набора кода;

ЗНАКОМСТВО С ИНСТРУМЕНТАРИЕМ: JUPYTER

6.1 Описание лекции

Эта лекция расскажет:

- что такое `Jupyter` и чем он хорош;
- о видах ячеек и режимах работы в `Jupyter`;
- о самых необходимых горячих клавишах;
- что такое ядро и почему вас это должно волновать.

6.2 Введение

Как любой мастер должен знать свой инструмент, так и любой человек, решивший пройти курс QML, должен понимать тонкости рабочей среды. Как вы уже могли понять по прошлому занятию, всю (или большую часть) работы мы будем делать в `Jupyter Notebook`. Но бояться нечего: по сути, `Jupyter` – это *продвинутый текстовый редактор* с функцией запуска кода и получения результатов вычислений. Настолько продвинутый, что позволяет вам не только рисовать картинки и писать формулы, но даже строить целые интерактивные карты:

На курсе, конечно, работы с гео-данными не будет, однако очень пригодятся вывод информации в виде *таблиц*, *создание и отрисовка простых графиков*, а главное, все это происходит в **браузере**. Редактирование кода в браузере не вызывает лишних проблем со средами разработки и в то же время оно доступно максимально широкому кругу людей. В этом редакторе можно запускать `Python`-код, что очень похоже на интерактивный редактор MATLAB, если вы с ним знакомы.

Благодаря удобству использования и доступности `Jupyter` в настоящее время стал крупным игроком в нише научных вычислений и быстрого прототипирования. Вдобавок он безумно удобен для обучения и передачи знаний. Почему? Давайте разбираться.

6.3 Типы ячеек

В `Jupyter` существует несколько типов **ячеек**, мы поговорим о двух основных: **Code** и **Markdown**. В прошлом уроке мы создали пустой **ноутбук**, чтобы проверить установку `Jupyter`. **Ноутбуком** это называется потому, что в переводе с английского **notebook** – это тетрадка (альтернативное название на русском языке). В тетрадке можно писать что-то осмысленное, черкаться, оставлять пометки. Сейчас вы должны видеть вот такой экран:

Здесь верхней красной чертой выделено поле с **названием** ноутбука. Можете кликнуть по нему, переименовать во что-то осмысленное и нажать `Enter`, чтобы применить изменения. Нижней же чертой обозначен

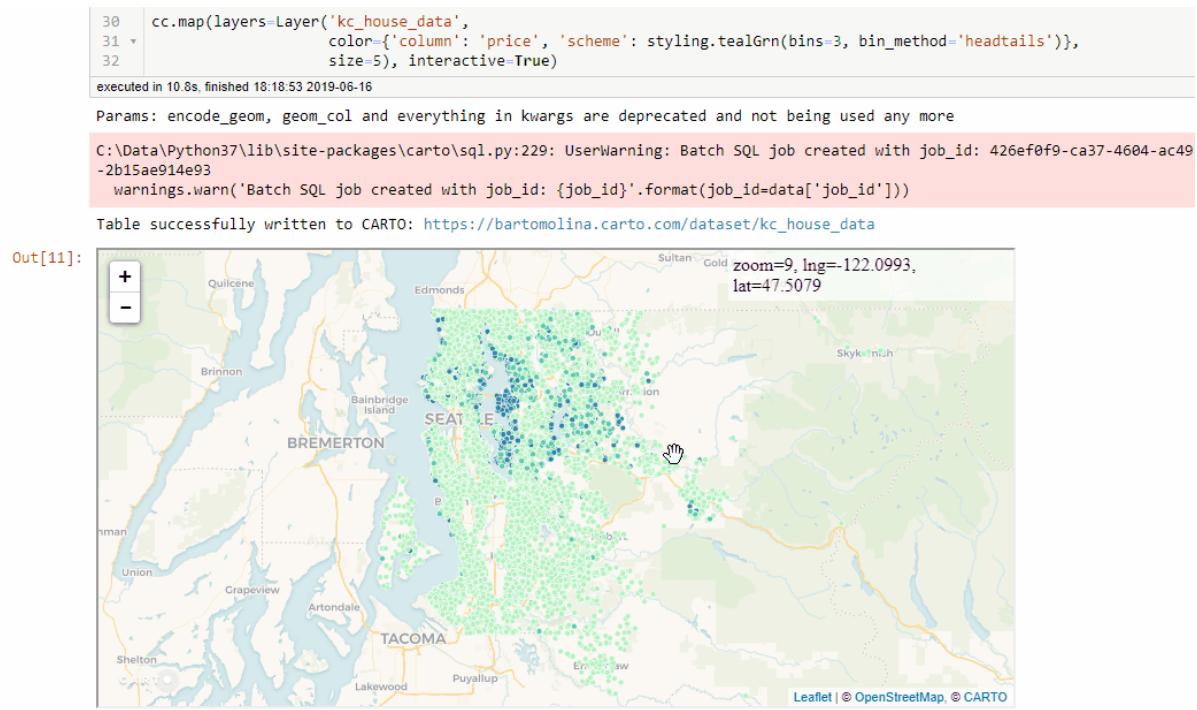


Fig. 6.1: Пример интерактивной визуализации прямо внутри рабочего файла с кодом

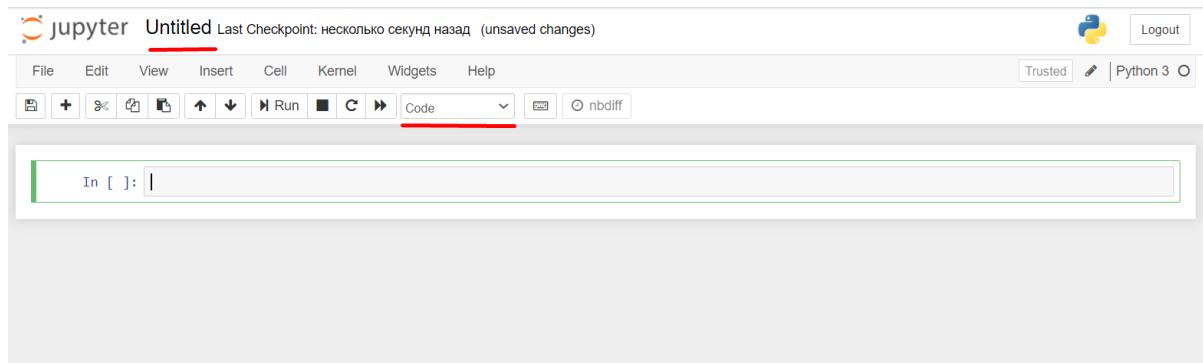


Fig. 6.2: Пример пустой только что созданной тетрадки.

выпадающий список переключений **типов ячеек**. По умолчанию создана одна **code** ячейка – в ней в будущем мы будем писать Python-код. Попробуйте кликнуть по списку и выбрать **Markdown** – визуально ячейка немножко изменится.

Что такое Markdown?

Markdown (произносится маркдаун) – облегченный язык разметки, созданный с целью обозначения **форматирования** в простом тексте с максимальным сохранением его читаемости человеком.

Пример: Text attributes `_italic_, **bold**`, `monospace` .

С помощью Markdown можно разнообразить код, вставить формулы (в том числе в LaTeX формате, если он вам знаком), ссылки на статьи и многое другое. Попробуйте скопировать код из примера выше в Markdown-ячейку и нажать кнопку Run:

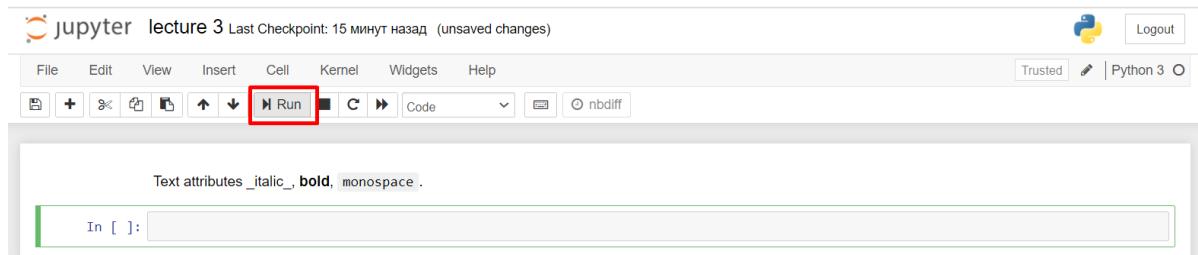


Fig. 6.3: Пример выполненной ячейки. Красным квадратом выделена кнопка Run.

Произошло следующее: ваша ячейка выполнилась и Jupyter отобразил ее содержимое. С помощью такого форматирования можно писать целые статьи с выкладками, формулами, графиками, то есть сопроводительной информацией. Поэтому, как уже было сказано, тетрадки очень удобны, особенно если соблюдать структуру, то есть писать сверху вниз с разделением на логические блоки. Также стоит отметить, что создалась новая Code-ячейка прямо под первой.

Note: Самая прекрасная часть тетрадок: **ячейки разных типов можно смешивать по порядку**, таким образом сначала описывая какую-то логику, а затем непосредственно реализовывая ее в коде и выполняя.

Вам не так часто придется писать Markdown-заметки самостоятельно, основная причина их создания – ваше желание и дальнейшее удобство использования ноутбука. Есть также другая причина, по которой мы акцентируем на них внимание. Может так произойти, что вы *случайно* изменили тип ячейки и не заметили этого. Теперь, если в Markdown-ячейку вставите Python-код, то ничего не произойдет или возникнет ошибка. Если вы заметили что-то странное при выполнении кода в тетрадке – **проверьте, корректен ли тип ячейки**. Для выполнения кода нужно выставить тип **Code**.

Tip: Обратите внимание, как визуально отличаются эти две ячейки. Одна из них имеет прозрачный фон, другая – серый. У **Code** ячейки также есть странная надпись слева (про нее еще поговорим).

Каждый раз вручную запускать код (или Markdown) через кнопку Run не очень-то удобно, поэтому можно запомнить две комбинации клавиш. **CTRL+Enter** выполнит текущую ячейку и оставит “курсор” (указатель на ячейку) **на том же месте**, не создавая лишнюю строчку в ноутбуке. **Shift+Enter** повторит функциональность кнопки Run: выполнит ячейку, а затем **перейдет на следующую** (или **создаст новую**, если текущая ячейка является **последней**).

Первая комбинация (CTRL+Enter) будет полезна в том случае, если вы что-то написали и знаете, что будете вносить изменения (например, менять цвет линии на графике в попытках добиться визуальной красоты), а

Multivariate Normal Distribution Equation

Recall the equation for the normal distribution from the [Gaussians](#) chapter:

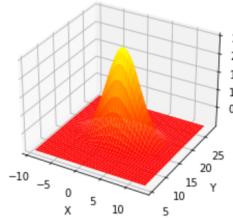
$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma^2} \exp\left[-\frac{1}{2}(x - \mu)^2/\sigma^2\right]$$

Here is the multivariate normal distribution in n dimensions.

$$f(\mathbf{x}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{(2\pi)^n |\boldsymbol{\Sigma}|}} \exp\left[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right]$$

The multivariate version merely replaces the scalars of the univariate equations with matrices. If you are reasonably well-versed in linear algebra this equation should look quite manageable. If not, don't worry, both FilterPy and SciPy provide functions to compute it for you. Let's ignore the computation for a moment and plot it to see what it looks like.

```
In [14]: import kf_book.mkf_internal as mkf_internal
mean = [2., 17.]
cov = [[10., 0.],
       [0., 4.]]
mkf_internal.plot_3d_covariance(mean, cov)
```



This is a plot of multivariate Gaussian with a mean of $\boldsymbol{\mu} = \begin{bmatrix} 2 \\ 17 \end{bmatrix}$ and a covariance of $\boldsymbol{\Sigma} = \begin{bmatrix} 10 & 0 \\ 0 & 4 \end{bmatrix}$. The three dimensional shape shows the probability density for any value of (X, Y) in the z-axis. I have projected the variance for X and Y onto the walls of the chart - you can see that they take on the Gaussian bell curve shape. The curve for X is wider than the curve for Y , which is explained by $\sigma_x^2 = 10$ and $\sigma_y^2 = 4$. The highest point of the 3D surface is at the means for X and Y .

Fig. 6.4: Пример грамотно оформленной Jupyter-тетрадки. Такую можно скинуть коллегам – и всем все будет понятно!

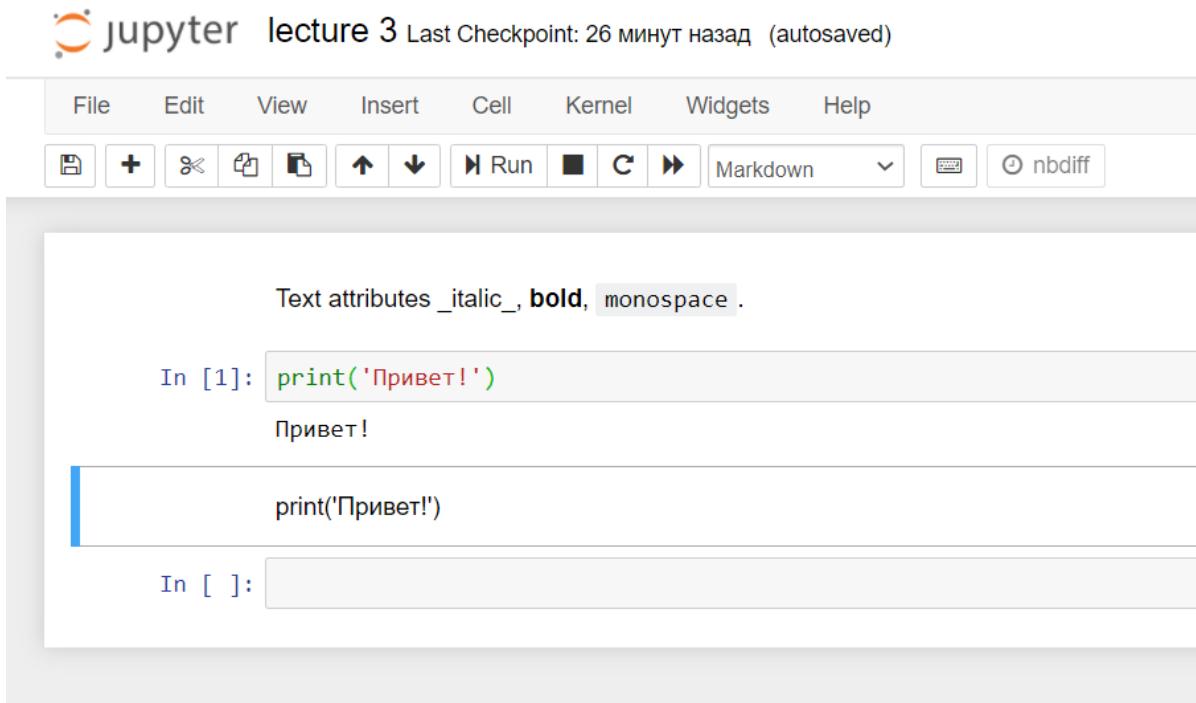


Fig. 6.5: Верхняя ячейка с `print` написана в Code-режиме и корректно выполняется, печатая строчку с приветствием. Нижняя ячейка же содержит текст, а не код, поэтому ничего работать не будет (точнее, код отобразится как текст, но не будет выполнен).

значит, придется менять код в этой же ячейке.

Вторая (`Shift+Enter`) пригодится тогда, когда вы хотите запустить много-много идущих подряд ячеек (можете представить, что коллега скинул вам свою тетрадку с 30 клетками и вы хотите ее запустить, чтобы получить данные).

Не беспокойтесь, буквально к концу первого блока лекций у вас выработается мышечная память и вы будете использовать сочетания клавиш на автомате.

6.4 Режимы работы

Пришло время разобраться с цветом курсора, выделяющего ячейки. Он может быть **синим** или **зеленым**.



Fig. 6.6: Пример разного цвета указателя клетки.

Никакой тайны за этим нет, это два режима: **режим редактирования и командный режим**. Зеленый цвет сигнализирует о том, что вы работаете с **текстовым содержимым ячейки**, то есть редактируете его! Можете писать код, вставлять формулы, что угодно. Но как только вы нажмете ESC на клавиатуре, цвет сменится на синий, что означает **возможность редактирования всего ноутбука, а не отдельных ячеек в нем**. Можно передвигать ячейки, удалять их (полностью, а не только текст в них), добавлять новые. Стрелочками на клавиатуре можно выбирать ячейки (скакать вверх и вниз). Как только доберетесь до нужной (а вместо этого можно просто кликнуть по ней мышкой, что полезно в ситуации, когда клетка *очень* далеко, в самом низу страницы) – жмите Enter, чтобы вернуться к редактированию.

Tip: Можно осуществлять переходы между режимами кликом мышки (внутри блока кода либо где-нибудь в стороне, слева или справа от ячейки, где ничего нет).

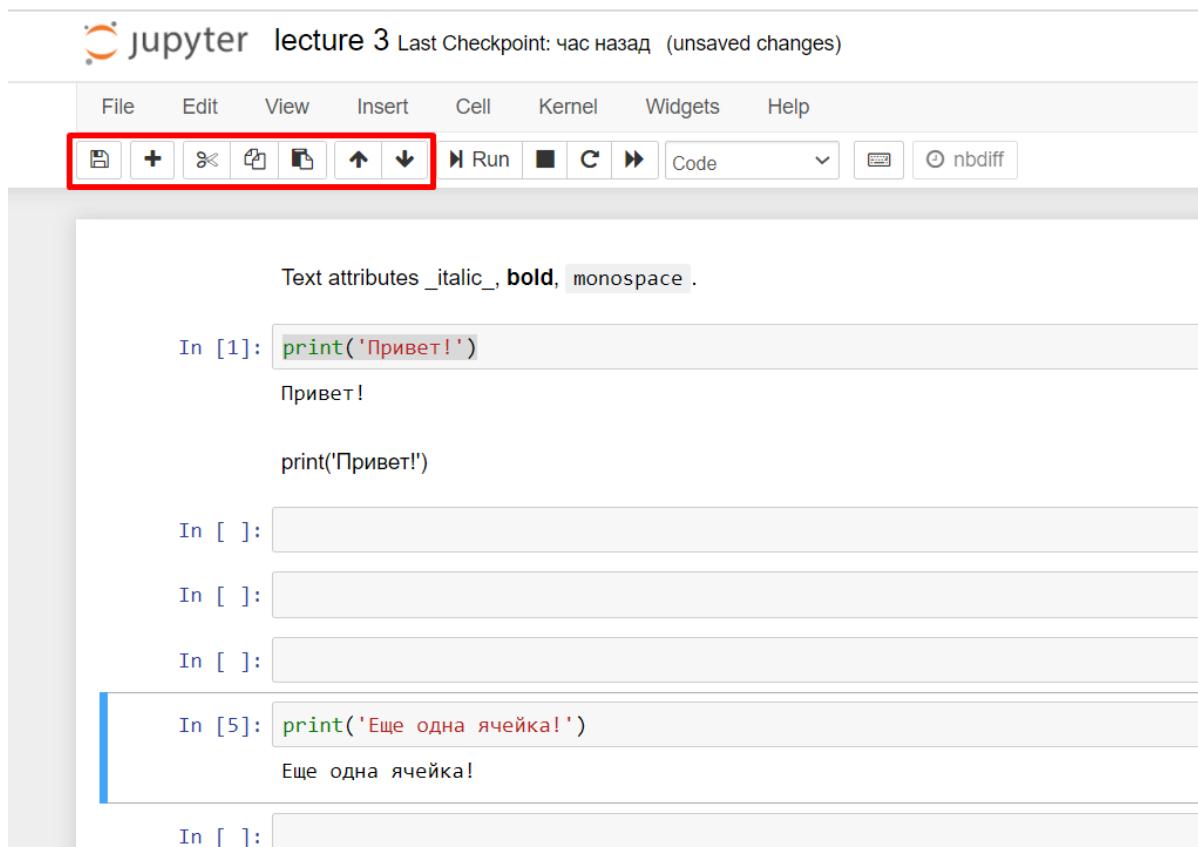


Fig. 6.7: Кнопки управления ноутбуком.

Выполнять описанные выше операции можно с помощью горячих клавиш (или хоткеев), либо через интерфейс. Описание выделенного блока кнопок для картинки выше (в порядке слева направо, с указанием сочетаний клавиш):

1. Сохранение ноутбука (CTRL+S) – делайте его **помоще**, чтобы не потерять результаты работы!
2. Создание ячейки ниже текущей (B) – В потому, что создается клетка снизу, то есть Below. Логика для A и Above аналогична.
3. Вырезать ячейку (X) – применимо и к целому блоку ячеек (можно выделить с зажатой клавишей Shift). Функциональность как и в Excel/Word: убрать в одном месте, чтобы вставить в другом.
4. Копировать ячейку (C).

5. Вставить ячейку из буфера (**V**) – после вырезания или копирования ячейки.
6. Переместить текущую выделенную ячейку **вверх**.
7. Переместить текущую выделенную ячейку **вниз**.

Описание всех доступных команд (и соответствующих им хоткеев) доступно при нажатии на кнопку с клавиатурой в правой части выделенного блока меню (вне красного прямоугольника).

Attention: Попробуйте потратить 5-7 минут на практику использования этих кнопок и сочетаний клавиш.

Первое время можете пользоваться только элементами UI-интерфейса – это нормально, главное, сопоставить кнопки и стоящую за ними функциональность.

6.5 Оставшиеся кнопки на панели

Про кнопку Run (и хоткей Shift+Enter) мы уже поговорили, а что с остальными?

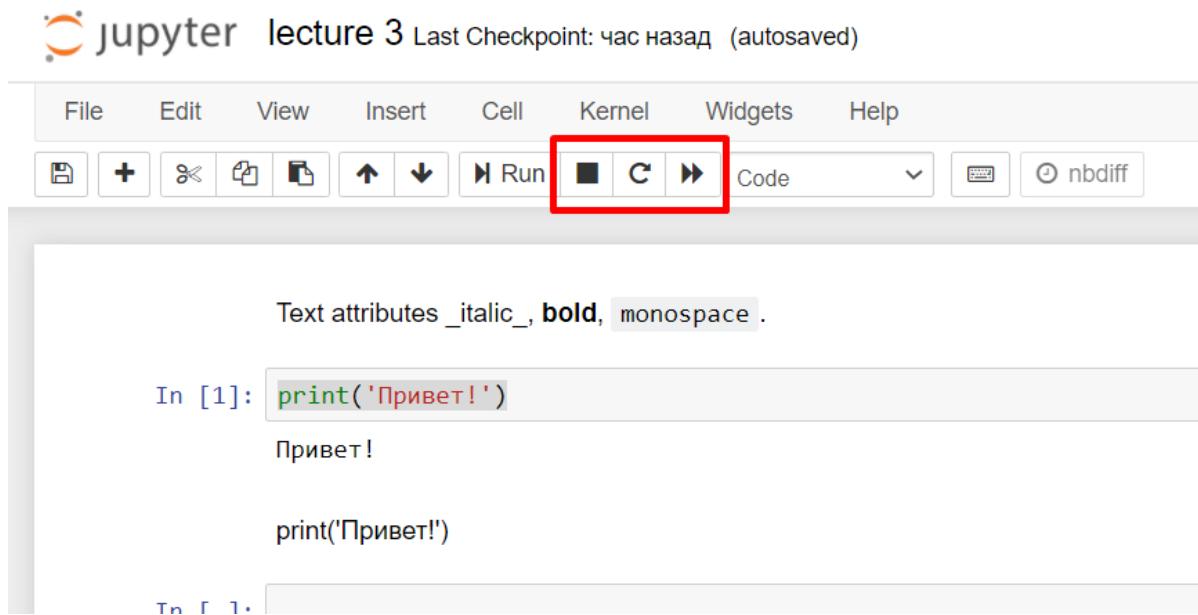


Fig. 6.8: Кнопки управления ЯДРОМ ноутбука.

Для того, чтобы вы могли запускать код Python, запускается так называемое “ядро” (или kernel), то есть приложение, которое непосредственно выполняет (запускает) ваш код и передает результаты обратно в Jupyter-ноутбук. За это отвечает как раз Run.

Справа от нее расположен Stop, который **прерывает выполнение программы**. Он может быть полезен в случаях, когда вы запустили расчеты на час, но заметили ошибку – и поэтому нужно и код переписать, и ячейку с кодом снова запустить. И в этой ситуации вы сначала **останавливаете** выполнение, редактируете код, затем жмете Run – и все готово!

Но случается беда и код не останавливается, потому что ядро Python зависает. В таких случаях нужно **перезапустить ядро** – и кнопка с закругленной стрелочкой Restart поможет осуществить задуманное. Будьте аккуратны – **вы потеряете ВСЕ несохраненные данные** (значения переменных, результаты расчетов,

данные для построения графиков). Сама тетрадка останется без изменений, то есть **написанное сохранится**. Концепция “ядра” и запуска кода станет более понятна, когда мы перейдем к практике.

Tip: Пока стоит держаться правила: “Накосячил? Попробуй остановить (Stop) ядро. Не получается? Тогда перезапускай (Restart) его!”

Нужно понимать, что ядро “помнит” все предыдущие выполненные ячейки (пока не будет перезагружено или выключено), а значит, вы можете позже в коде переиспользовать те части, которые были описаны ранее (например, переменные или физические константы). Иными словами, состояние ядра сохраняется во времени и между ячейками – **оно относится к документу в целом**, а не к отдельным ячейкам.

Последняя кнопка из выделенного блока имеет говорящее название: **Re-start and run all**. Ядро будет **перезапущено** (все переменные и данные удалятся), а затем **каждая ячейка будет выполнена в порядке сверху вниз**. Поэтому рекомендуется соблюдать структуру, чтобы запускать код с нуля (после возвращения к ноутбуку на следующий день, но с новым ядром, так как компьютер был выключен) – и он отрабатывал.

6.6 Что это за In [*]?

Та самая надпись слева от запущенной Code-ячейки. Это вспомогательная информация о том, что происходит с **кодовой ячейкой** (**In** означает **Input**, то есть ввод кода). Возможно несколько вариантов заполнения.

The screenshot shows three code cells in a Jupyter Notebook:

- In []:** print('Я еще не запущена()')
- In [6]:** print('Я уже не запущена()')
Я уже не запущена()
- In [*]:** while True:
 continue
 print('А я в бесконечном цикле...')

Fig. 6.9: Пример трех видов информации о статусе ячейки.

В первом случае в квадратных скобках **ничего нет** – это значит, что **ячейка еще не была запущена**. Возможно, вы забыли, а быть может, она просто ждет своего часа.

Во втором случае **ячейка была запущена** шестой по счету (да-да, **ячейки выполняются по порядку, который задаете вы сами!**) и она **успешно отработала и завершилась**.

В последней строчке умышленно был сделан бесконечный цикл. Это означает, что **код никогда не сможет выполниться** и будет висеть до тех пор, пока вы не остановите (Stop) ядро. Поэтому там выведен **индикатор выполнения ячейки** – в скобках указана звездочка *. Обратите внимание: **это не всегда плохой сигнал**. Если ваш код должен выполняться 2-3 минуты, то все это время будет выводиться [*]. Когда код отработает и результат будет получен, отрисуется цифра (например, [7]).

6.7 Самая полезная клавиша

Пришло время программировать! Скопируйте себе в ноутбук кусок кода ниже и попробуйте его запустить. Не переживайте, он может показаться сложным и непонятным, но сейчас не требуется понимание всех деталей.

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Fixing random state for reproducibility
np.random.seed(19680801)

# Compute pie slices
N = 20
θ = np.linspace(0.0, 2 * np.pi, N, endpoint=False)
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)
colors = plt.cm.viridis(radii / 10.)

ax = plt.subplot(111, projection='polar')
ax.bar(θ, radii, width=width, bottom=0.0, color=colors, alpha=0.5)
plt.show();
```

Вы увидите ошибку. По сообщению видно (стрелочка в левой части указывает на проблемное место), что во второй строчке используется слово `mat`, при этом Python жалуется на отсутствие такого модуля. Все дело в том, что в коде выше производится попытка рисования графика и для этого используется библиотека `matplotlib`. Но в одной из строк написано только `mat`. Это не дело, давайте исправлять. Однако всех библиотек не запомнишь – и это не нужно. Попробуйте поставить курсор после буквы `t` (и перед точкой) и нажать TAB. Вы должны увидеть **список подсказок** и из него выбрать нужный вариант. Этот список не только сокращает время написания кода (за счет автоматического дополнения), но и позволяет избежать ошибок в написании. Обязательно пользуйтесь этим инструментом.

Если вы все сделали правильно, воспользовавшись подсказкой, то после очередного запуска (Run) кода появится рисунок.

Fig. 6.10: Другой пример удачного использования: есть несколько переменных со сложным, но очень похожим названием. Не стоит их перепечатывать – достаточно нажать TAB!

Что ж, большое количество новой информации позади, давайте подведем итоги!

6.8 Что мы узнали из лекции

- Существует два основных типа ячеек, один предназначен для программирования (написания кода), другой – для описания (формул, определений).
- Существует два режима работы: `edit` – редактирование конкретной ячейки и `control` – работа со всей структурой тетрадки/ноутбука.
- Режимы работы можно различить по цвету рамки вокруг активной ячейки, а тип ячейки – по прозрачности фона и наличию надписи `In []` слева.
- Между режимами можно переключаться с помощью `Esc` и `Enter`.
- Чтобы запустить ячейку (с кодом или текстом), нужно нажать на кнопку `Run` сверху, либо воспользоваться сочетаниями клавиш: `Shift+Enter`, `CTRL+Enter`.
- Нужно не забывать сохранять ноутбук (`CTRL+S`), а быстро добавить ячейку кода можно с помощью плюсика слева сверху (или клавиши `B`).
- Клавиши `C` / `V` позволяют копировать и вставлять ячейки в `control`-режиме.
- `In [3]` указывает на порядок выполнения ячеек с кодом, `In [*]` – на процесс выполнения.
- Если вы долго ждете выполнения Code-блока, можно `Stop`нуть ядро, если не помогает – `Restart`нуть.
- Ядро – это процесс, выполняющий код, и после перезагрузки оно не сохраняет переменные.
- `TAB` – ваш друг, позволяющий избегать опечаток, а также реже пользоваться документацией.

6.9 Бонус-материал

Полная официальная документация по Jupyter находится по [ссылке тут](#).

Если вам захочется узнать больше о трюках в ноутбуках, о недостатках и преимуществах по сравнению с альтернативами, предлагаем посмотреть выступление Joel Grus:

ПЕРЕМЕННЫЕ И ВЫВОД ИНФОРМАЦИИ В PYTHON

7.1 Описание лекции

Из этой лекции вы узнаете о:

- базовых типах данных и переменных;
- простейших операциях с числами и строками;
- функции print;
- выводе информации с подстановкой значений.

7.2 Суть переменных в Python

Настало время приступить к изучению непосредственно Python, ведь прошло три лекции, а мы об языке программирования и не говорили вовсе! И поскольку наш курс посвящен физике, то начнем со знакомой всем по школьным карандашам формулы $E = mc^2$. По ней можно вычислить полную энергию физического объекта E с помощью известной массы объекта m и константы c . Эта постоянная, указывающая на скорость света в вакууме, используется настолько часто, что для нее выделили **отдельное обозначение в виде буквы латинского алфавита**, как и для многих других аналогичных величин. Если в формуле встречается c (в известном контексте), то вы всегда уверены, что именно нужно подставить при расчетах.

Этот пример полностью описывает концепцию **переменных** в языках программирования, и Python не исключение. Запись $x = 3$ означает, что везде по тексту далее под иксом подразумевается именно тройка, и ничего другого (пока не будет введено новое определение). Этой же логике подчиняется Python. Сначала указывается **имя переменной**, а затем – ассоциируемое с ней значение.

```
c = 299_792_458 # запишем константу, м/с
m = 0.5 # масса некоторого абстрактного объекта, кг
E = m * (c ** 2) # вычисляем энергию, Дж

some_variable_1 = 10.2 # какая-то другая переменная
m = 12
```

Пример кода выше иллюстрирует сразу несколько базовых концепций, которые нужно запомнить:

1. В объявлении переменной нет ничего сложного. Синтаксис и правила интуитивно понятны: это можно делать как в физике/математике, как в учебниках и статьях.
2. `#` означает комментарий, то есть произвольный текст, который не воспринимается Python (все до конца строки кода полностью игнорируется). Служит исключительно для создания подсказок в коде, объяснения происходящего, то есть для удобства.

3. Числа могут быть **целыми и вещественными**. Разряды в целых числах для удобства визуального восприятия можно разделять нижней чертой.
4. **Значение переменной может быть вычислимым**, то есть являться производной от других переменных (как E , ведь это результат перемножения). На самом деле значение вычисляется в момент объявления переменной (при сложной формуле расчета процесс может занимать некоторое время).
5. Операция возведения в степень реализуется с помощью `**`.
6. В качестве названия переменных можно использовать **буквы и цифры**, а также некоторые символы. Однако **имя переменной не может начинаться с цифры**.
7. Переменные можно переопределять (и даже менять тип). Однако **старое значение в этом случае будет безвозвратно утрачено**. В данном примере после выполнения последней строчки нельзя установить, чему было равно `t` до того, как переменной было присвоено значение дюжины.

Если говорить менее строго и более абстрактно, то **переменная – это контейнер** (или коробка), в котором что-то лежит, и на самой коробке на приклеенном листочке бумаги указано содержимое. Чем понятнее надпись, тем легче найти и использовать объект (поэтому переменные с названием из одной буквы воспринимаются плохо, особенно если таких переменных очень много).

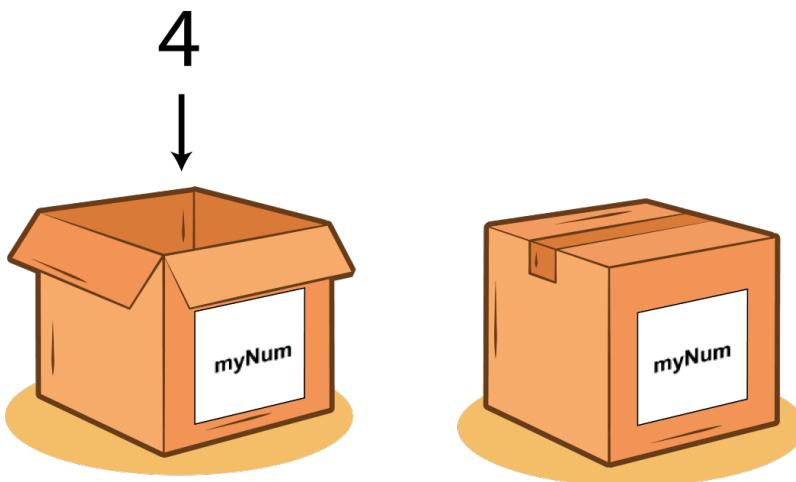


Fig. 7.1: Объявить переменную – значит положить объект в коробку с подписью.

7.3 Типы переменных

В листинге кода выше важно заметить, что существует разница между двумя типами численных переменных: **целые и вещественные**. При сугубо математических расчетах и арифметических операциях тип переменной не имеет значения. Однако для некоторого функционала в Python нужно быть аккуратным. Мы поговорим подробно об этом в следующих лекциях, а пока стоит запомнить, что вещи, которые необходимо посчитать – в том числе и **длину** чего-то *счетного* – должны быть целочисленными (как и в жизни: первый, второй, третий...).

Attention: Целочисленный тип называется `int` (от `Integer`), вещественный – `float`. Эти типы можно переводить из одного в другой. При переводе вещественного числа в целое теряется часть информации.

Тип переменной – и это относится не только к числам, но и к **любому** объекту – можно узнать с помощью функции `type`. Для вывода информации в Python используется функция `print`. Что именно представляет собой функция мы рассмотрим в более поздних лекциях, пока стоит думать об этом как о некотором объекте,

который зависит (рассчитывается) от других объектов и выдает некоторый результат. Для передачи аргументов используются круглые скобки (аналогично математике: $y = F(x)$). Давайте скомбинируем эти знания и рассмотрим пример:

```
first_variable = 10
second_variable = 10.0

# запишем в переменные значения типов данных
type_of_first_variable = type(first_variable)
type_of_second_variable = type(second_variable)

# и распечатаем сами типы, чтобы посмотреть глазами и сравнить
print(type_of_first_variable)
print(type_of_second_variable)

# перезапишем переменные
first_variable = 12.9
second_variable = int(first_variable)
third_variable = float(second_variable)

# в print() можно передавать несколько переменных
print(first_variable, second_variable, third_variable)
```

```
<class 'int'>
<class 'float'>
12.9 12 12.0
```

Внимательно проанализируйте код выше – в нем продемонстрирован базовый синтаксис **преобразования типов и вывода информации**. Легко увидеть подтверждение высказанных ранее тезисов: `second_variable` действительно потеряла часть информации (дробную часть числа), которую нельзя вернуть, если преобразовать переменную обратно во `float`. Преобразование типов в языках программирования называется **приведением** (типов, то есть привести одно к другому, а не из-за страшилок про духов).

7.4 Арифметические операции с числами

Математика Python максимально близка к естественной: `+-*` и `**` (рассмотренное ранее возведение в степень) работают в точности как ожидается. С делением `/` есть нюанс: **возвращаемое значение всегда вещественное**.

```
a = 3
b = 12.1

c = a + b

# можно объединять вызовы функций print и type
# без создания лишней переменной
print(type(c))

# и даже трех функций, включая приведение типа
print(type(int(c)))

# деление числа на само себя дает единицу, но...
print(a / a)
print(b / b)
```

(continues on next page)

(continued from previous page)

```
print(c / c)
print(12 / 4)
```

```
<class 'float'>
<class 'int'>
1.0
1.0
1.0
3.0
```

Note: Обратите внимание, что операции не изменяют переменную саму по себе (то есть операция `a + b` не меняет ни `a`, ни `b`). Чтобы сохранить получаемое значение, нужно присвоить его некоторой переменной (в примере выше это `c`). Если вы хотите изменить непосредственно саму переменную, то можно переприсвоить ей значение на основе расчета: `a = a + b` или `c = c + 12`.

Даже несмотря на то, что кейс с делением числа на само себя очевиден (всегда получается единица, кроме деления на нуль), будет выведено вещественное значение. Сами же вещественные значения можно складывать, вычитать, умножать и возводить в степень как с целыми, так и с вещественными числами (и наоборот). Если в таком выражении используется хотя бы одна `float`-переменная, то и результат будет не целочисленным. Однако:

```
a = 3
b = 2

print(a + b, type(a + b))
print(a * b, type(a * b))
print(a ** b, type(a ** b))
```

```
5 <class 'int'>
6 <class 'int'>
9 <class 'int'>
```

Это *практически* все тонкости, которые необходимо знать, чтобы не совершать базовые ошибки.

7.4.1 Примечание

Возможно, у вас родился вопрос относительно расстановки пробелов в коде выше. Обязательно ли соблюдать такой синтаксис? Нужно ли ставить пробелы до и после знаков операций? На самом деле нет: это делается исключительно для удобства чтения кода и **настоятельно рекомендуется не удаляться от стандартов языка**. Код ниже выполнится без ошибок, однако ухудшается читаемость:

```
a=      3
b     =2

print(a +b, type(a+ b))
print(a      * b, type(a *b))
print(a**b, type(a      ** b))
```

```
5 <class 'int'>
6 <class 'int'>
9 <class 'int'>
```

7.5 Строковые переменные

Мы разобрались в том, как описывать и хранить числа, как производить арифметические расчеты. Базовый математический язык освоен, но мы же люди, и хочется общаться словами! Конечно, Python позволяет это делать. Благодаря **строковым переменным** можно хранить и соединять текстовую информацию:

```
text_variable = 'тут что-то написано'
another_text_variable = "Вася, впиши сюда что-нибудь перед публикацией курса!"

long_text = """
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut
labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit
esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non
proident, sunt
in culpa qui officia deserunt mollit anim id est laborum.

"""

print(another_text_variable)
```

Вася, впиши сюда что-нибудь перед публикацией курса!

В примере выше рассмотрено три способа создания текстовых переменных. Первые два не отличаются между собой с точки зрения Python, то есть неважно, используете ли вы одинарные кавычки ' или двойные ". Однако стоит понимать, что если ваша строка содержит в себе такой символ, то кавычка должна быть изменена:

```
error_string = 'Chillin' kid'
another_error_string = "И тут он мне говорит: "у тебя нет ног!""

print(error_string)
print(another_error_string)
```

Механизм ошибки таков, что Python неясно: это вы закончили строчку и дальше идет какая-то команда, или же строчка продолжается. В обоих случаях **нужно сменить способ создания строки** – и тогда все будет хорошо:

```
error_string = "Chillin' kid"
another_error_string = 'И тут он мне говорит: "у тебя нет ног!"'

print(error_string)
print(another_error_string)
```

Chillin' kid
И тут он мне говорит: "у тебя нет ног!"

Если необходимо сохранить какой-либо объемный текст или сообщение, можно воспользоваться мультистрочным объявлением переменной, как в первом примере блока.

Строки можно объединять для удобства вывода информации:

```
first_string = 'Результат вычислений: '
second_string = ". Это не так много!"
```

(continues on next page)

(continued from previous page)

```
a = 12
b = 2
result = a * b

# два способа вывода:
print(first_string, result, second_string)

# либо через склейку строк вручную
# обратите внимание на приведение типа int к str
result_string = first_string + str(result) + second_string
print(result_string)
```

Результат вычислений: 24 . Это не так много!
 Результат вычислений: 24. Это не так много!

Warning: Будьте аккуратны со сложением строк. Объединение строк "3" и "5" даст результат "35", а не 8 – и тип результирующего значения **будет строковый**. **Сложить строку и число нельзя**: вы получите ошибку и никакого приведения типов не произойдет. Здесь возникнет двусмысленность – нужно привести число к строке и затем сконкатенировать или же строку к числу (а вдруг это невозможно?), после чего сложить.

Склейивание строк называется **конкатенацией**. Попробуйте в Jupyter-ноутбуке объединить строковые, целочисленные и вещественные переменные в разных комбинациях. Разберитесь, что означает ошибка, которая будет выведена в случае, если не делать приведение типов (то есть без `str` в `str(result)`).

Note: Обратите внимание на пробел между числом и точкой в первом случае. Они добавлены автоматически функцией `print` – это сделано для того, чтобы разные объекты при последовательном выводе не “склеивались” друг с другом. Во втором случае этого не происходит, так как мы напрямую склеиваем строки и только затем передаем результат конкатенации на печать в `print`.

Но на практике это не совсем удобно, поэтому в Python придумали F-строки. Их суть в том, что переменная из кода напрямую подставляется (с автоматическим приведением типа к строке) в саму строку! Вот:

```
a = 12
b = 2
result = a * b

result_string = f'Результат вычислений: {result}. Это не так много!'

# и без f
wrong_result_string = 'Результат вычислений: {result}. Это не так много!'

print(result_string)
print(wrong_result_string)
```

Результат вычислений: 24 . Это не так много!
 Результат вычислений: {result}. Это не так много!

Для объявления F-строки нужно, во-первых, использовать одинаковые кавычки на концах текста. Во-вторых, нужно указать литеру `f` перед самой строкой. И последнее – нужно обрамить название конкретной переменной (`result` в данном случае) в фигурные скобки.

Когда переменная одна, а также нет текста после ее использования, то выгода F-строк не так очевидна (относительно простого `print(some_string, some_variable)`). Однако представьте, что вам нужно вывести координаты точки в трехмерном пространстве, значение времени, параметры системы и значение некоторой функции от всех переменных выше!

```
# так тоже можно!
x, y, z = 12.1, 0, 13
# скобки, как и в математике, задают порядок выполнения вычислений
func_val = (x * y) ** z

current_time = 30.113412

# а вот так можно писать длинные F-строки (но работает и для обычных)
out_string = (f'В точке с координатами X={x}, Y={y}, Z={z} значение функции '
              f'равно {func_val}. Состояние системы указано на момент '
              'времени t=' + str(current_time))

print(out_string)
```

В точке с координатами X=12.1, Y=0, Z=13 значение функции равно 0.0. Состояние системы указано на момент времени t=30.113412

7.6 Что мы узнали из лекции

- **Переменные** – это “контейнеры”, в которые можно что-то положить и дать название.
- Математика в Python не имеет сложных правил, процесс вычислений максимально интуитивен.
- Арифметические операции могут **менять тип** результирующей переменной.
- `type()`, `print()` – базовые функции, с помощью которых можно делать **самопроверки** по ходу написания кода.
- **Сменить тип** переменной можно вызовом функций `int()`, `float()`, `str()`.
- Строки могут обрамляться как `'`, так и `"` (но этих символов **не должно быть внутри** текста).
- F-строки облегчают комплексный вывод, содержащий как текст, так и переменные Python (и автоматически приводит типы).

УСЛОВНЫЕ КОНСТРУКЦИИ, БУЛЕВА ЛОГИКА И СРАВНЕНИЯ

8.1 Описание лекции

В этой лекции мы расскажем про:

- if/else-конструкции и условия;
- тип bool;
- операторы сравнения;
- блоки кода и отступы.

8.2 Ветвление логики

В прошлых лекциях мы рассмотрели программы с линейной структурой: сначала выполнялась первая конструкция (например, объявление переменной), затем вторая (преобразование переменной или расчет по формуле), после – третья (print для вывода результатов). Можно сказать, что происходило последовательное исполнение команд, причем каждая инструкция выполнялась **обязательно**. Но что делать, если хочется опираться на обстоятельства и принимать решения о том, выполнять одну часть кода или другую?

Допустим, по числу x нужно определить его абсолютную величину, то есть модуль. Программа должна напечатать значение переменной x , если $x > 0$ или же величину $-x$ в противном случае ($-(-5) = 5$). Этую логику можно записать следующим образом:

```
x = -3 # попробуйте поменять значение переменной

if x > 0:
    print("Исходный x больше нуля")
    print(x)
else:
    print("Исходный x меньше или равен нулю")
    print(-x)
```

```
Исходный x меньше или равен нулю
3
```

В этой программе используется условная инструкция `if` (в переводе с английского “если”). `if` – это ключевое зарезервированное слово (так нельзя называть свою переменную), указывающее на **условную конструкцию**. После `if` следует указать вычислимое выражение, которое **можно проверить на истинность** (то есть можно сказать, правда это или нет). Общий вид конструкции следующий:

```
if (Условие) :
    <Блок инструкций 1>
else:
    <Блок инструкций 2>
```

else – тоже ключевое слово (в переводе – “иначе”). Таким образом, можно в голове придерживаться такой интерпретации: “**если** условие верно (истинно), **то** выполнни первый блок команд, **иначе** выполнни второй блок”.

Условная инструкция содержит как минимум ключевое слово **if** (единожды), затем может идти любое количество (включая ноль) блоков с условием **else if** <условие> (*иначе если*, то есть будет выполнена проверка нового условия в случае, если первая проверка в **if** не прошла), затем – optional – конструкция **else**. Логика чтения и выполнения кода сохраняет порядок **сверху вниз**. Как только одно из условий будет выполнено, выполнится соответствующая инструкция (или набор инструкций), а все последующие блоки будут проигнорированы. Это проиллюстрировано в коде:

```
x = -3.8 # попробуйте поменять значение переменной

if x > 0:
    print('x больше нуля')
elif x < 0: # можно написать "else if x < 0:"
    print('x меньше нуля')
else:
    print('x в точности равен нулю')
print('Такие дела!')
```

```
x меньше нуля
Такие дела!
```

Понятно, что **x** не может одновременно быть и больше нуля, и меньше (или равен ему). Среди всех трех **print**-блоков будет выполнен **только один**. Если **x** действительно больше нуля, то второе условие (**x < 0**) даже не будет проверяться – Python сразу же перейдет к последней строке и выведет надпись “Такие дела!”.

Чтобы лучше разобраться в том, как работает код, можно использовать **визуализаторы** – например, **такой**. Прогоняйте через него весь код (даже в несколько строк) и сверяйте со своими ожиданиями от его работы.

8.3 А что вообще такое эти ваши условия?

Выше было указано, что после конструкций **if/else if** необходимо указать **условие**, которое еще и должно быть истинным или ложным (“правда или нет”). Давайте попробуем определить необходимый **тип** переменной.

```
x = -3.8

condition_1 = x > 0
condition_2 = x < 0

print(condition_1, type(condition_1))
print(condition_2, type(condition_2))
```

```
False <class 'bool'>
True <class 'bool'>
```

Видно, что оба условия имеют один и тот же тип – **bool**, то есть **boolean**. По **определению**:

Boolean (Булев, Логический тип данных) – примитивный тип данных в информатике, которые могут принимать **два возможных значения**, иногда называемых истиной (True) и ложью (False).

Оказывается, что в коде выше мы получили **ВСЕ** возможные варианты булевой переменной – это истина (True, пишется только с заглавной буквы) и ложь (False, аналогично). Никаких других значений быть для условия не может. Вот такой это простой тип данных.

8.4 Способы получения bool

Какими вообще могут быть условия? Как с ними можно обращаться? Согласно [официальной документации](#), в Python есть такие операторы сравнения:

Операция	Значение
<	строго меньше чем
<=	меньше или равно
>	строго больше, чем
>=	больше или равно
==	равный
!=	не равный

Fig. 8.1: Все операции сравнения работают нативно (так же, как и в математике)

```
print(3.0 > 3)
print(3.0 == 3)
```

```
False
True
```

Здесь практически нечего рассматривать, операторы сравнения они и в Python операторы. Куда интереснее принцип **объединения различных условий в одно** – для создания комплексной логики.

Пусть стоит задача определения четверти точки по ее координатам на двумерной плоскости. Решение такой задачи может быть записано следующим образом:

```
x = -3.6
y = 2.5432

if x > 0:
    if y > 0:
        # x > 0, y > 0
        print("Первая четверть")
    else:
        # x > 0, y < 0
```

(continues on next page)

(continued from previous page)

```

    print("Четвертая четверть")
else:
    if y > 0:
        # x < 0, y > 0
        print("Вторая четверть")
    else:
        # x < 0, y < 0
        print("Третья четверть")

```

Вторая четверть

Пример показывает, что выполняемым блоком кода может быть любой блок Python, включая новый логический блок с `if-else` конструкцией. Однако его можно сократить с помощью **логических операторов** `and`, `or` и `not`. Это стандартные логические операторы **Булевой алгебры**.

Логическое **И** является бинарным оператором (то есть оператором с двумя operandами: левым и правым) и имеет вид `and`. Оператор `and` возвращает `True` тогда и только тогда, когда **оба его операнда имеют значение True**.

Логическое **ИЛИ** является бинарным оператором и возвращает `True` тогда и только тогда, когда **хотя бы один операнд равен True**. Оператор “логическое ИЛИ” имеет вид `or`.

Логическое **НЕ** (отрицание) является унарным (то есть **с одним операндом**) оператором и имеет вид `not`, за которым следует единственный operand. Логическое **НЕ** возвращает `True`, **если operand равен False и наоборот**.

Эти правила необходимо запомнить для успешного создания сложных условий с целью разделения логики, заложенной в Python-коде.

Проиллюстрируем правила в коде на простых примерах. Обратите внимание на то, как можно объявлять `bool`-переменные – это не сложнее, чем создание целочисленного значения:

```

true_value = True
false_value = False

# False потому, что один из operandов является False
some_value = true_value and false_value
print(some_value)

# True потому, что хотя бы один из operandов равен True
some_value = true_value or false_value
print(some_value)

# отрицание True (истины) есть False (ложь)
some_value = not true_value
print(some_value == false_value)

# пример сложного условия – порядок лучше в явном виде задавать скобками
hard_condition = (not true_value or false_value) or (true_value != false_value)
print(hard_condition)

```

False
True
True
True

Теперь попробуем их применить на приближенных к практике примерах:

```

x = -3.6
y = 2.5432

if x > 0 and y > 0: # конструкция заменяет два вложенных if
    print("Первая четверть")
elif x > 0 and y < 0:
    print("Четвертая четверть")
elif y > 0:
    print("Вторая четверть")
else:
    print("Третья четверть")

# определим, большое ли число x (в терминах модуля)
x_is_small = (x < 3) and (x > -3)
# число большое, если оно не маленькое (по модулю)
x_is_large = not x_is_small # можно отрицать факт малости x

print('Is x small? ', x_is_small)
print('Is x large? ', x_is_large)

# так тоже можно писать - на манер неравенств в математике
another_x_is_small = -3 < x < 3
print(another_x_is_small)
print(another_x_is_small == x_is_small)

```

```

Вторая четверть
Is x small? False
Is x large? True
False
True

```

Так как вторая переменная `x_is_large` – это отрицание (`not`) первой (`x_is_small`), то они **никогда** не будут равны.

8.5 Блоки кода и отступы

В примерах выше вы наверняка заметили упоминание “блок кода”, а также откуда-то взявшимся отступы после условий, и это не случайно. Во-первых, давайте признаем, что так условные конструкции (особенно вложенные!) читать куда легче, и глаза не разбегаются. Во-вторых, это особенность языка Python – здесь не используются скобки `{}` для указания блоков, все форматирование происходит с помощью отступов. Отступы **всегда** добавляются в строки кода **после двоеточия**.

Для выделения блока инструкций (строк кода, выполняющихся подряд при любых условиях), относящихся к инструкциям `if`, `else` или другим, изучаемым далее, в языке Python используются **отступы**. Все инструкции, которые относятся к одному блоку, должны иметь **равную величину отступа**, то есть одинаковое число **пробелов в начале строки**. В качестве отступа PEP 8 рекомендует использовать **отступ в четыре пробела** и не рекомендует использовать символ табуляции. Если нужно сделать еще одно вложение блока инструкций, достаточно добавить еще четыре пробела (см. пример выше с поиском четверти на плоскости).

Tip: Хоть и не рекомендуется использовать символ табуляции для создания отступов, кнопка Tab на вашей клавиатуре в Jupyter-ноутбуке (при курсоре, указывающем на начало строки кода) создаст отступ в четыре пробела. Пользуйтесь этим, чтобы не перегружать клавишу пробела лишними постукиваниями :).

8.6 Что мы узнали из лекции

- Для задания логики выполнения кода и создания нелинейности используются **условные инструкции**, поскольку они следуют некоторым условиям.
- Условная инструкция задается ключевым словом `if`, после которого может следовать несколько (от нуля) блоков `else if/elif`, и – дополнительно – в конце добавляется `else`, если ни один из блоков выше не сработал.
- Условия должны быть **булевого типа** (`bool`) и могут принимать **всего два значения** – `True` и `False`. Выполнится тот блок кода, который задан истинным (`True`) условием (и только первый!).
- Условные конструкции можно вкладывать друг в друга, а также объединять с помощью **логических операторов** `and`, `or` и `not`.
- **Блок кода** – это несколько подряд идущих команд, которые будут выполнены последовательно.
- Чтобы выделить блок кода после условия, используйте **отступы** – четыре пробела.
- Чтобы создать отступ в `Jupyter`, нужно нажать `Tab` в начале строки кода.

СПИСКИ И ЦИКЛЫ В PYTHON

9.1 Описание лекции

На этом занятии мы разберем следующие темы:

- списки (`list`) и их методы;
- индексация списков;
- что такое срезы и зачем они нужны;
- цикл `for` и функция `range`;
- итерация по спискам, `list comprehensions`.

9.2 Введение в списки объектов

В предыдущих лекциях мы оперировали малым количеством переменных. Для каждого блока логики или примера кода вводилось 3-5 объектов, над которыми осуществлялись некоторые операции. Но что делать, если объектов куда больше? Скажем, вам необходимо хранить информацию об учащихся класса – пусть это будет рост, оценка по математике или что-либо другое. Не знаю, как вы, но я нахожу крайне неудобным создание 30 отдельных переменных. А если еще и нужно посчитать среднюю оценку в классе!

```
average_grade = petrov_math + kosareva_math + zinchenko_math + kotenkov_math + ...
average_grade = average_grade / 30
```

Такой код к тому же получается крайне негибким: если количество студентов, как и их состав, изменится, то нужно и формулу переписать, так еще и делитель – в нашем случае 30 – изменять.

Часто в программах – даже в (квантовом) машинном обучении – приходится работать с большим количеством **однотипных** переменных. Специально для этого придуманы **массивы** (по-английски `array`). В Python их еще называют **списками** (`list`). В некоторых языках программирования эти понятия отличаются, но не в Python. Список может хранить переменные **разного** типа. Также списки называют “контейнерами”, так как они хранят какой-то набор данных. Для создания простого списка необходимо указать квадратные скобки или вызвать конструктор типа (`list` – это отдельный тип, фактически такой же, как `int` или `str`), а затем перечислить **объекты через запятую**:

```
# разные способы объявления списков
first_list = []
second_list = list()
third_list = list([1, 2, "stroka", 3.14])
fourth_list = [15, 2.2, {"another_list": False}]
```

(continues on next page)

(continued from previous page)

```
print(type(second_list), type(fourth_list))
print(first_list, fourth_list)
```

```
<class 'list'> <class 'list'>
[] [15, 2.2, ['another_list', False]]
```

Tip: Хоть список и хранит переменные разного типа, но так делать без особой необходимости не рекомендуется – вы сами скорее запутаетесь и ошибитесь в обработке объектов списка. В большинстве других языков программирования массив может хранить только объекты одного типа.

Для хранения сложных структур (скажем, описание студента – это не только оценка по математике, но и фамилия, имя, адрес, рост и так далее) лучше использовать классы – с ними мы познакомимся в будущем. А еще могут пригодиться **кортежи**, или `tuple`. Про них в лекции не рассказано, самостоятельно можно ознакомиться [по ссылке](#).

Теперь можно один раз создать список и работать с ним как с единым целым. Да, по прежнему для заведения оценок студентов придется разово их зафиксировать, но потом куда проще исправлять и добавлять! Рассмотрим пример нахождения средней оценки группы, в которой всего 3 учащихся, но к ним присоединили еще 2, а затем – целых 5:

```
# базовый журнал с тремя оценками
math_journal = [3, 3, 5]

# добавим новопришедших студентов
math_journal.append(4)
math_journal.append(5)

# и сразу большую группу новых студентов
math_journal.extend([2, 3, 4, 5, 5])

print(f"math_journal = {math_journal}")

# найдем среднюю оценку как сумму всех оценок, деленную на их количество
avg_grade = sum(math_journal) / len(math_journal)
print(f"avg_grade = {avg_grade}")
```

```
math_journal = [3, 3, 5, 4, 5, 2, 3, 4, 5, 5]
avg_grade = 3.9
```

В коде выше продемонстрировано сразу несколько важных аспектов:

1. Добавлять по одному объекту в конец списка можно с помощью метода списка `append`;
2. Метод `append` принимает в качестве аргумента один Python-объект;
3. Слияние списков (конкатенация, прямо как при работе со строками) нескольких осуществляется командой `extend` (расширить в переводе с английского);
4. Для списков определена функция `len`, которая возвращает целое число `int` – количество объектов в списке;
5. Функция `sum` может применяться к спискам для суммирования всех объектов (если позволяет тип – то есть для `float`, `int` и `bool`). Попробуйте разобраться самостоятельно, как функция работает с последним указанным типом);

6. Для методов `append` и `extend` не нужно приравнивать результат выполнения какой-то переменной – изменится сам объект, у которого был вызван метод (в данном случае это `math_journal`);
7. Списки в Python **упорядочены**, то есть объекты сами по себе места не меняют, и помнят, в каком порядке были добавлены в массив.

Tip: В тексте выше встречается термин **метод**, который, быть может, вам не знаком. По сути метод – это такая же **функция**, о которых мы говорили раньше, но она принадлежит какому-то объекту с определенным типом. Не переживайте, если что-то непонятно – про функции и методы мы поговорим подробно в ближайших лекциях! `print`, `sum` – функции, они существуют сами по себе; `append`, `extend` – методы объектов класса `list`, не могут использоваться без них.

9.3 Индексация списков

Теперь, когда стало понятно, с чем предстоит иметь дело, попробуем усложнить пример. Как узнать, какая оценка у третьего студента? Все просто – нужно воспользоваться **индексацией** списка:

```
# базовый журнал с пятью оценками
math_journal = [1, 2, 3, 4, 5]

third_student_grade = math_journal[3]
print(third_student_grade)
```

4

И снова непонятный пример! Давайте разбираться:

1. Для обращения к *i*-тому объекту нужно в квадратных скобках указать его индекс;
2. **Индекс** в Python начинается **С НУЛЯ** – это самое важное и неочевидное, здесь чаще всего случаются ошибки;
3. Поэтому `[3]` обозначает взятие **четвертой** оценки (и потому выводится четверка, а не тройка);
4. Всего оценок 5, но так как индексация начинается с нуля, то строчка `math_journal[5]` выведет ошибку – нам доступны лишь индексы `[0, 1, 2, 3, 4]` для взятия (так называется процедура обращения к элементу списка по индексу – взятие по индексу).

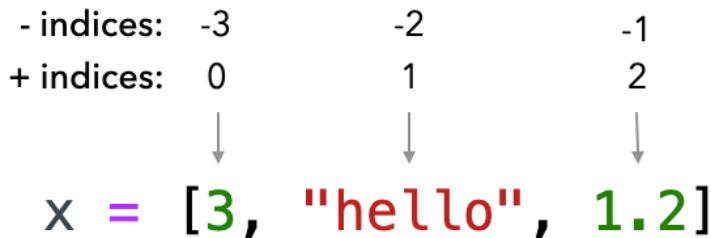


Fig. 9.1: Пример списка из трех объектов. Сверху показаны их индексы, включая отрицательные

Также в Python существуют отрицательные индексы (-1, -2 ...). Они отсчитывают объекты списка, начиная с конца. Так как нуль уже занят (под первый объект), то он не используется.

```
# базовый журнал с пятью оценками
math_journal = [1, 2, 3, 4, 5]

# возьмем последнюю оценку
last_grade = math_journal[-1]
print(f"Последняя оценка: {last_grade}")

# а теперь -- предпоследнюю
prev = math_journal[-2]
print(f"Предпоследняя оценка: {prev}")

# конечно, взятие по индексам можно использовать в ранее разобранном синтаксисе

if math_journal[-1] < math_journal[-2]:
    math_journal[-1] += 1
    print("Последняя оценка меньше предпоследней. Натянем студенту?")
else:
    math_journal[-2] = 2
    print("Последний студент сдал очень хорошо, на его фоне предпоследний просто_"
          "двоечник!" )
```

```
Последняя оценка: 5
Предпоследняя оценка: 4
Последний студент сдал очень хорошо, на его фоне предпоследний просто двоечник!
```

Все это важно не только для грамотного оперирования конкретными объектами, но и следующей темы -

9.4 Срезы

Срезы, или slices – это механизм обращения сразу к нескольким объектам списка. Для создания среза нужно в квадратных скобках указать двоеточие, слева от него – индекс начала среза (по умолчанию 0, можно не выставлять) **включительно**, справа – границу среза **не включительно** (пустота означает “до конца списка”). Может показаться нелогичной такая разнородность указания границ, но на самом деле она безумно удобна – особенно вместе с тем, что индексация начинается с нуля. Быстрее объяснить на примере:

```
# базовый журнал с пятью оценками
math_journal = [1, 2, 3, 4, 5]

# как взять первые 3 оценки?
first_3_grades = math_journal[:3]
print(f"{first_3_grades = }")

# как взять последние две оценки?
last_2_grades = math_journal[-2:]
print(f"{last_2_grades = }")

# сделаем срез на 4 оценки, начиная со второй (с индексом 1)
start_index = 1
some_slice = math_journal[start_index : start_index + 4]
print(f"{some_slice = }")

# возьмем столько объектов из начала, сколько объектов в some_slice
yet_another_slice = math_journal[:len(some_slice)]
```

(continues on next page)

(continued from previous page)

```
# а вот так можно проверить, попадает ли объект в список
print("Верно ли, что единица входит в some_slice? {1 in some_slice}")
print("Верно ли, что единица входит в yet_another_slice? {1 in yet_another_slice}")
```

```
first_3_grades = [1, 2, 3]
last_2_grades = [4, 5]
some_slice = [2, 3, 4, 5]
Верно ли, что единица входит в some_slice? {1 in some_slice}
Верно ли, что единица входит в yet_another_slice? {1 in yet_another_slice}
```

Tip: Можно сделать пустой срез, и тогда Python вернет пустой список без объектов. Можете проверить сами: `["1", "2", "3"] [10:20]`

Давайте проговорим основные моменты, которые **крайне важно понять**:

1. Так как индексация начинается с нуля (значение по умолчанию) и правая граница не включается в срез, то берутся объекты с индексами `[0, 1, 2]`, что в точности равняется трем первым объектам;
2. Срез `[-2:]` указывает на то, что нужно взять все объекты до конца, начиная с предпоследнего
3. Значения в срезе могут быть **вычисляемы** (и задаваться сколь угодно сложной формулой), но должны оставаться **целочисленными**;
4. Если нужно взять k объектов, начиная с i -го индекса, то достаточно в качестве конца среза указать $k+i$;
5. Для проверки вхождения какого-либо объекта в список нужно использовать конструкцию `x_obj in some_list`, которая вернет `True`, если массив содержит `x_obj`, и `False` в ином случае;
6. Самый простой способ сделать копию списка - это сделать срез по всему объекту: `my_list[:]`. Однако будьте внимательны – в одних случаях копирование происходит полностью (по значению), а в некоторых сохраняются ссылки (то есть изменив один объект в скопированном списке вы измените объект в исходном). Связано это с типом объектов (`mutable/immutable`), подробнее об этом будет рассказано в следующей лекции. В общем, если вы работаете с простыми типами (`int/str`), то срез вернет копию, и ее изменение не затронет исходный список. Однако для хранения новых данных нужна память, поэтому при копировании десятков миллионов объектов можно получить ошибку, связанную с нехваткой памяти.

9.5 Циклы

До сих пор в примерах мы хоть и обращались к нескольким объектам, добавляли и меняли их, все еще не было рассмотрено взаимодействие сразу с несколькими. Давайте попробуем посчитать, сколько студентов получили оценку от 4 и выше. Для этого интуитивно кажется, что нужно **пройтись по всем оценкам от первой до последней**, сравнить каждую с четверкой. Для прохода по списку, или **итерации**, используются **циклы**. Общий синтаксис таков:

```
example_list = list(...)
for item in example_list:
    <> блок кода внутри цикла (аналогично блоку в if)
    ... что-то сделать с item
    <>
```

Здесь `example_list` – это некоторый итерируемый объект. Помимо списка в Python существуют и другие итерируемые объекты, но пока будем говорить о массивах.

Этот цикл работает так: указанной **переменной item** присваивается **первое значение из списка**, и выполняется **блок кода** внутри цикла (этот блок, напомним, определяется отступом). Он выполняется весь от начала отступа и до конца, как и было объяснено в пятой лекции). Этот блок еще иногда называют **телом цикла**. Потом переменной item присваивается следующее значение (второе), и так далее. Переменную, кстати, можно называть как угодно, не обязательно item.

Итерацией называется каждый **отдельный проход** по телу цикла. Цикл всегда повторяет команды из тела цикла несколько раз. Два примера кода ниже аналогичны:

```
math_journal = [3, 4, 5]
counter = 0

for cur_grade in math_journal:
    if cur_grade >= 4:
        counter += 1

print(f"Всего хорошистов и отличников по математике {counter} человека")
```

Всего хорошистов и отличников по математике 2 человека

```
math_journal = [3, 4, 5]
counter = 0

cur_grade = math_journal[0]
if cur_grade >= 4:
    counter += 1

# не забываем менять индекс с 0 на 1, так как каждый раз берется следующий элемент
cur_grade = math_journal[1]
if cur_grade >= 4:
    counter += 1

# и с единицы на двойку
cur_grade = math_journal[2]
if cur_grade >= 4:
    counter += 1

print(f"Всего хорошистов и отличников по математике {counter} человека")
```

Всего хорошистов и отличников по математике 2 человека

Понятно, что первый кусок кода обобщается на любой случай – хоть оценок десять, хоть тысяча. Второе решение не масштабируется, появляется **много одинакового кода, в котором легко ошибиться** (не поменять индекс, к примеру).

Движемся дальше. Так как каждый элемент списка закреплен за конкретным индексом, то в практике часто возникают задачи, логика которых связана на индексах. Это привело к тому, что появилась альтернатива для итерации по списку. Функция range принимает аргументы, аналогичные срезу в списке, и возвращает итерируемый объект, в котором содержатся целые числа (индексы). Так как аргументы являются аргументами функции, а не среза, то они соединяются запятой (как print(a, b) нескольких объектов). Если подан всего один аргумент, то нижняя граница приравнивается к нулю. Посмотрим на практике, как сохранить номера (индексы) всех хорошо учащихся студентов:

```
math_journal = [4, 3, 4, 5, 5, 2, 3, 4]
good_student_indexes = []
```

(continues on next page)

(continued from previous page)

```

for student_index in range(len(math_journal)):
    current_student_grade = math_journal[student_index]
    if current_student_grade >= 4:
        good_student_indexes.append(student_index)

print(f"Преуспевающие по математике студенты находятся на позициях: {good_student_
    indexes}")

```

Преуспевающие по математике студенты находятся на позициях: [0, 2, 3, 4, 7]

В примере `student_index` принимает последовательно все значения от 0 до 7 включительно. `len(math_journal)` равняется 8, а значит, восьмерка сама не будет включена в набор индексов для перебора. На каждой итерации `current_student_grade` меняет свое значение, после чего происходит проверка. Если бы была необходимость пробежаться только по студентам, начиная с третьего, то нужно было бы указать `range(2, len(math_journal))` (двойка вместо тройки потому, что индексация с нуля, ведь мы перебираем индексы массива).

Выше описаны основные концепции обращения со списками. Их крайне важно понять и хорошо усвоить, без этого писать любой код будет безумно сложно. Скопируйте примеры к себе в .ipynb-ноутбук, поиграйтесь, поменяйте параметры цикла и проанализируйте изменения.

9.6 List comprehensions

Некоторые циклы настолько просты, что занимают 2 или 3 строчки. Как пример – привести список чисел к списку строк:

```

# грубый вариант
inp_list = [1, 4, 6, 8]
out_list = []

for item in inp_list:
    out_list.append(str(item))

# list comprehension
out_list = [str(item) for item in inp_list]
print(out_list)

```

['1', '4', '6', '8']

Две части кода идентичны за вычетом того, что нижняя – с непонятной конструкцией в скобках – короче. Python позволяет в рамках одной строки произвести какие-либо простые преобразования (помним, что `str()` – это вызов функции, а значит если у вас есть сложная функция, которая делает квантовые вычисления, то ее также можно применить!). Фактически самый частый пример использования – это паттерн “**применение функции к каждому объекту списка**”.

9.7 Что мы узнали из лекции

- `list` – это **объект-контейнер, который хранит другие объекты разных типов**. Запись происходит упорядочено и последовательно, а каждому объекту присвоен **целочисленный номер, начиная с нуля**;
- для добавления одного объекта в `list` нужно использовать метод объекта `list.append`, а для расширения списка сразу на несколько позиций пригодится `extend`;
- проверить, сходит ли объект в список, можно с помощью конструкции `obj in some_list`;
- индексы **могут быть отрицательными**: `-1, -2 ...` В таком случае нумерация начинается от последнего объекта;
- можно получить часть списка, сделав **срез** с помощью конструкции `list[start_index : end_index]`, при этом объект на позиции `end_index` не будет включен в возвращаемый список (т.е. **срез работает не включительно по правую границу**);
- часто со списками используют **циклы, которые позволяют итерироваться по объектам массива** и выполнять произвольную логику в рамках отделенного отступом блока кода;
- для итерации по индексам можно использовать `range()`;
- простые циклы можно свернуть в **list comprehension**, и самый частый паттерн для такого преобразования – это **применение некоторой функции к каждому объекту списка** (если `x` это функция, то синтаксис будет таков: `[x(item) for item in list]`).

СЛОВАРИ

10.1 Описание лекции

В этой лекции мы расскажем про:

- то, что такое `dict`;
- методы и примеры использования;
- хранение изменяемых и неизменяемых типов;
- хэширование.

10.2 Что такое `dict`

`dict` (от английского “*dictionary*”, словарь) – еще один тип данных в Python. Словари хранят пары ключ: значение. То есть в списках можно достать элемент, если указать его позицию в виде целого числа, а в словарях – тот самый ключ. Обратите внимание, `dict` – **неупорядоченный** тип данных, поэтому достать элемент по номеру не получится, но отображение содержимого будет в **порядке добавления** элементов. **Уникальность** ключей должна поддерживаться, чтобы всегда можно было быстро найти одно единственное значение.

В некоторых языках программирования можно встретить ассоциативные массивы – полную аналогию `dict`. Также вспомним базы данных, широко использующиеся во всем мире для хранения информации: в таблице можно установить первичный ключ, который уникально идентифицирует запись, как и ключ соответствует значению в словаре. Самый простой пример ключа - порядковый номер объекта!

10.2.1 Создание словаря

Использовать словарь стоит, когда нужно сохранять объекты с какими-то ключами и обращаться к объектам по известным ключам. Один из способов определения словаря: указание пар `ключ: значение` через **запятую** внутри фигурных скобок `{ }`. Напоминает `set`, правда? `{ }` позволяет создать **пустой** словарь, но не пустое множество.

Например, вы решили упростить себе жизнь и больше не запоминать дни рождения коллег. Вместо этого, лучше хранить их в одном месте:

```
# локально попробуйте поменять значение переменной dates
dates = {"Кунг Фьюри": "1968-09-09", "Наташа Романова": "1985-03-15"}
print(dates)
```

```
{'Кунг Фьюри': '1968-09-09', 'Наташа Романова': '1985-03-15'}
```

В примере `dates` имеет две пары значений. В первой паре строка "Кунг Фьюри" является *ключом*, а "1968-09-09" – его *значением*.

10.2.2 Получение значения по ключу

Чтобы получить значение по ключу, необходимо обратиться к переменной, содержащей словарь, и указать ключ в квадратных скобках []:

```
dates["Кунг Фьюри"]
```

```
'1968-09-09'
```

Если указать неверный ключ в [], Python будет ругаться: выбросит ошибку `KeyError` и перестанет выполнять код. Чуть ниже посмотрим, как можно избежать таких ситуаций.

```
# пока этого ключа нет в словаре, будет ошибка при обращении
# поэтому используем перехват ошибок
try:
    print(dates["Капитан Ямайка"])

except KeyError as e:
    print(f"Ключа действительно нет: {e}")
```

```
Ключа действительно нет: 'Капитан Ямайка'
```

10.2.3 Изменение и добавление значений

Синтаксис изменения значения по ключу и добавления нового ключа со значением одинаковый: в [] нужно указать ключ, поставить = и указать значение, которое теперь будет соответствовать ключу.

```
# этот ключ уже был в примере
dates["Кунг Фьюри"] = "1960-09-09"

# а такого не было
dates["Капитан Ямайка"] = "1930-10-04"

print(dates)
```

```
{'Кунг Фьюри': '1960-09-09', 'Наташа Романова': '1985-03-15', 'Капитан Ямайка': '1930-10-04'}
```

Если ключ уже был в словаре, значение по нему изменится на новое, а **старое будет удалено**. Указание нового ключа со значением добавляет пару в словарь.

10.3 Основные методы словаря

10.3.1 Проверка вхождения и get ()

Помните, ранее говорили, что обращение к несуществующему ключу приводит к ошибке? Пришло время посмотреть пару способов борьбы!

Можно проверить, есть ли интересующий ключ среди множества ключей словаря. Это делается при помощи бинарного оператора `in`. Слева должен быть указан ключ, справа – переменная со словарем:

```
# еще способ создания: пары можно передавать как аргументы dict через '='
marks = dict(линал=100, английский=92)

# False
print(f'{"матан" in marks = }')

# True
print(f'{"линал" in marks = }')
```

```
"матан" in marks = False
"линал" in marks = True
```

В коде проверку можно использовать в условной конструкции `if`, чтобы принимать решение в зависимости от наличия ключа:

```
if "матан" in marks:
    print(marks["матан"])

else:
    print("Нет оценки по матану :(")
```

```
Нет оценки по матану :(
```

Теперь о методе `get ()`: при помощи него тоже можно получать значения из словаря по ключу. `KeyError` никогда не появится: если ключа нет, по умолчанию возвращается `None`:

```
empty_dict = {}

# None
print(empty_dict.get("ключ"))
```

```
None
```

Вторым аргументом метода `get ()` можно указать значение, которое должно возвращаться вместо `None`, когда ключ не был найден:

```
# теперь будет возвращено значение '-1'
print(empty_dict.get("ключ", -1))
```

```
-1
```

10.3.2 Что такое “длина словаря”?

Функция `len()` для словаря будет возвращать количество пар `ключ: значение` (их столько же, сколько ключей), которое в нем содержится:

```
# empty_dict -- пустой словарь, поэтому длина равна 0
print(f"{len(empty_dict) = }")

# а вот словарь marks уже содержит две пары, поэтому длина 2
print(f"{len(marks) = }")
```

```
len(empty_dict) = 0
len(marks) = 2
```

10.3.3 Удаление из словаря

Есть несколько способов очистки в словаре: можно убирать по ключу, а можно сразу удалить все!

- при помощи инструкции `del` (от английского “*delete*”) можно удалить пару `ключ: значение` (*замечу, что удаление ключа эквивалентно удалению пары `ключ: значение`, так как мы теряем возможность найти то самое значение*), в общем виде:

```
# таким образом из словаря "словарь" будет удален ключ "название_ключа"
# и соответствующее ему значение
del словарь[название_ключа]
```

Предположим, коллега из самого первого примера уволился и больше нет смысла хранить его день рождения:

```
# из словаря dates удаляется ключ "Наташа Романова"
del dates["Наташа Романова"]
print(dates.get("Наташа Романова"))
```

```
None
```

- `pop()` – метод, который достает значение, хранящееся по переданному ключу, и **сразу** удаляет ключ из словаря:

```
# еще один способ создания словаря из последовательности пар
holidays = dict([("January", [1, 2, 3, 4]), ("Feburary", [23]), ("March", [8])])

# pop() возвращает значение, соответствующее ключу, значит его можно присвоить
# переменной
january_days = holidays.pop("January")

# напечатается соответствующий массив
print(january_days)
```

```
[1, 2, 3, 4]
```

Для метода `pop()` есть возможность указать значение, которое будет возвращено при обращении к несуществующему ключу. Почти как `get()`, но все таки, без указания этого значения, `pop()` выбрасывает `KeyError`

- `popitem()` имеет схожее название, но не путайте с предыдущим методом: этот на вход не принимает ключ, а возвращает пару `ключ: значение`, которая была добавлена последней (*такое поведение гарантируется с Python 3.7*).

```
# в результате -- последняя добавленная пара
print(holidays.popitem())
```

```
('March', [8])
```

- `clear()` позволяет удалить сразу все ключи словаря, то есть полностью его очистить:

```
# вернемся к предыдущему примеру
# словарь становится пустой
holidays.clear()

# значит, длина равна 0
print(len(holidays))
```

```
0
```

Обратите внимание на то, как работают методы `pop()`, `popitem()` и `clear()`: как только вызываются, словарь меняет свой состав (*изменения происходят in place*, то изменить по месту без копирования).

10.3.4 Обновление и добавление ключей

Мы уже видели, что значения в словарь можно добавлять или менять, обращаясь по ключу. Python предоставляет возможность не писать кучу присваиваний, а использовать лаконичный метод `update()`, который на вход может принимать либо

- другой словарь
- пары ключ: значение в какой-то последовательности (например, кортежи по два значения в списке: первое – ключ, второе – значение)

Tip: Кортеж (tuple) - еще одна стандартная структура данных. Очень похожа на list (поддерживает индексацию), но является неизменяемой: как только tuple создан, поменять его нельзя

```
# создадим два словаря: в первом уже есть два ключа
quidditch_team = {"Fred Weasley": "3rd year", "George Weasley": "3rd year"}

# во втором -- один ключ
new_members = {"Harry Potter": "1st year"}

# добавим пары из new_members
# метод update() также работает in place, поэтому после выполнения данной
# строки кода, в словаре quidditch_team станет три ключа
quidditch_team.update(new_members)

print(quidditch_team["Harry Potter"])
```

```
1st year
```

А что, если в `update()` передать пары, ключ которых уже был в словаре? Значения по дублирующимся ключам будут **перезаписаны на новые**:

```
# данный ключ (то, что записано первым в кортеже) уже есть в quidditch_team
member_update = [("Harry Potter", "2nd year")]
```

(continues on next page)

(continued from previous page)

```
# значение, соответствующее "Harry Potter", будет переписано
quidditch_team.update(new_members)

print(quidditch_team["Harry Potter"])
```

```
1st year
```

10.4 Доступ к ключам и значениям

В Python можно без проблем извлекать отдельно по ключам или значениям, а также итерироваться по элементам словарей в цикле `for`. Осталось разобраться, как это работает.

10.4.1 Ключи

По умолчанию, в конструкциях вида

```
# после in указано название переменной, хранящей словарь
for key in dict_var:
    ...
```

переменные цикла (тут – `key`) будут принимать значения из множества **ключей** словаря. Аналогично можно использовать метод `keys()` (*позволяет достать все ключи*), который явно говорит, что ваш цикл идет по ключам, например:

```
# словарь в качестве ключей хранит имена игроков
for player in quidditch_team:

    # на каждой итерации будет напечатан ключ и значение
    print(player, quidditch_team[player])
```

```
Fred Weasley 3rd year
George Weasley 3rd year
Harry Potter 1st year
```

10.4.2 Значения

При помощи метода `values()` можно получить все значения, хранящиеся по всем ключам словаря:

```
# можно создать переменную со всеми значениями словаря
school_years = quidditch_team.values()
```

Приведем пример с циклом:

```
# словарь в качестве значений хранит годы обучения
for year in quidditch_team.values():

    # на каждой итерации будет год обучения игрока
    print(year)
```

```
3rd year
3rd year
1st year
```

Напрямую по значению получить ключ нельзя

10.4.3 Все и сразу

Существует метод `items()`, который достает пары ключ: значение в виде последовательности кортежей. Его же часто удобно использовать в циклах, чтобы не тащить длинную запись в виде названия словаря и квадратных скобок с ключом при обращении к значению:

```
# сразу две переменные: первая последовательно будет ключами,
# вторая -- значениями
for player, year in quidditch_team.items():

    # items() избавляет от необходимости обращаться quidditch_team[player],
    # чтобы получить значение. Оно уже в year
    print(f"Player {player} is in {year}")
```

```
Player Fred Weasley is in 3rd year
Player George Weasley is in 3rd year
Player Harry Potter is in 1st year
```

10.5 Сортировка

Функция `sorted()` доступна и для словарей. По умолчанию ключи словаря поддерживают порядок, в котором были добавлены, но можно отсортировать их в нужном направлении (в зависимости от типа):

```
# вспомним про рабочие дни, ключи - целые числа по номеру дня недели
week = {7: "weekend", 6: "weekend", 1: "workday"}

# в sorted_week окажутся ключи, отсортированные в порядке возрастания
sorted_week = sorted(week)
print(f"Порядок возрастания: {sorted_week}")
```

```
Порядок возрастания: [1, 6, 7]
```

Не забудьте, когда в функции передается просто название переменной со словарем, работа идет только над множеством ключей.

Можно ли отсортировать словарь по значениям? Да, можно попробовать самостоятельно разобраться с аргументами функции `sorted()`.

10.6 Что можно хранить

Теперь добавим немного технических подробностей: возможно, вы уже заметили самостоятельно, что dict может принимать в качестве ключа не всякое значение. На самом деле только **хэшируемые** объекты (*можно вызвать функцию hash () и получить значение*) могут быть ключами словаря, на значения это ограничение не распространяется. В dict и set значение хэша от объекта используется для поиска внутри структуры.

Ключом словаря нельзя сделать объект **изменяемого** типа, например, list, set или **сам dict**, так как значение их хэша может измениться со временем. Неизменяемый кортеж может быть ключом только если не содержит внутри изменяемые объекты.

10.6.1 Изменяемость и неизменяемость

В англоязычной литературе изменяемые типы называют **mutable**, а неизменяемые – **immutable**, [почитать документацию](#)

В Python все – объект. Когда пользователь присваивает значение переменной, она начинает ассоциироваться с ячейкой памяти, где лежит это значение. Переменная знает адрес, откуда можно получить значение. `id()` и `hex()` показывают адрес в памяти компьютера. `_id()` – адрес в десятичном виде, а `hex()` поможет перевести в шестнадцатеричный.

По адресу лежит так называемое **внутреннее состояние** переменной:

- **неизменяемые** типы не позволяют менять внутреннее состояние, значение переменной может поменяться только вместе с адресом
- **изменяемые** типы позволяют менять внутреннее состояние переменной при сохранении адреса (возвращаемое `id()` значение не меняется, но *значение* переменной каким-то образом преобразовывается). Изменение по ссылке называется изменением *in place*

Неизменяемые типы

Из стандартных неизменяемыми являются:

- int
- float
- bool
- str
- tuple

Давайте сразу рассмотрим пример:

```
counter = 100

# полученное вами значение адреса может отличаться
print(counter, hex(id(counter)))
```

```
100 0x7f6f3be69d80
```

А теперь поменяем значение counter:

```
counter = 200
print(f"counter = {counter}, {hex(id(counter))} = )")
```



Fig. 10.1: counter указывает на 100 (цифры в нижней правой части как раз и показывают тот самый адрес)

```
counter = 200, hex(id(counter)) = '0x7f6f3be6aa00'
```

Кажется, что раз значение переменной counter поменялось, то и содержимое по предыдущему адресу изменилось? Нет, на самом деле counter теперь указывает в другое место:

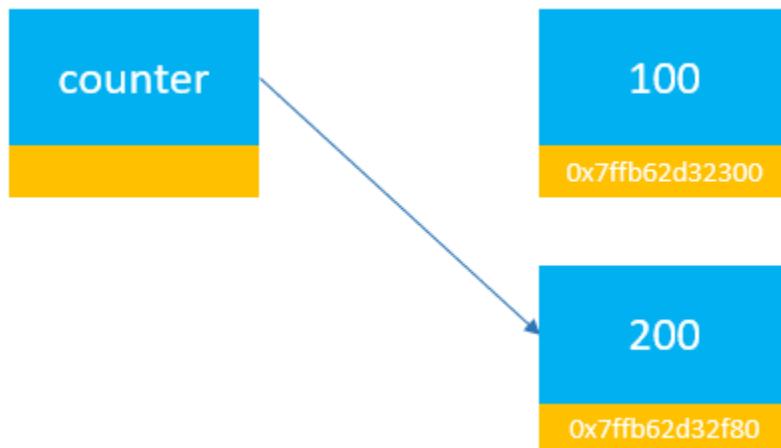


Fig. 10.2: counter указывает на новое значение 200 с другим адресом

Из интересного: Python заранее создает объекты для чисел от -5 до 256, поэтому для переменных со значением из этого диапазона берутся заранее готовые ссылки.

```
# создадим две переменные с одинаковыми значениями в диапазоне от -5 до 256
a = 20
b = 20

# a и b указывают на одно и то же место в памяти
# попробуйте у себя поменять значение a и b на число больше 256 или меньше -5
id(a) == id(b)
```

```
True
```

Изменяемые типы

Стандартные изменяемые типы это:

- list
- set
- dict

У списков есть метод `append()`, позволяющий добавить в него значение:

```
# создадим список и напечатаем его адрес
ratings = [1, 2, 3]
print(f"Было: {hex(id(ratings))}")

ratings.append(4)
print(f"Стало: {hex(id(ratings))} - ничего не поменялось!")
```

```
Было: 0x7f6f34b2f580
Стало: 0x7f6f34b2f580 - ничего не поменялось!
```



Fig. 10.3: Начальное состояние списка содержало три элемента

После добавления еще одного, адрес `ratings` не изменился.



Fig. 10.4: После добавления элемента поменялось лишь внутреннее состояние

10.7 Что мы узнали из лекции

- Новый тип данных – **словарь!** Позволяет хранить соответствие ключ: значение;
- Несколько способов создания `dict`, примеры:

```
# при помощи "литерала" - фигурных скобок {}
flowers = {"roses": "red", "violets": "blue"}

# при помощи вызова
# dict() и последовательности с парами значений
```

(continues on next page)

(continued from previous page)

```
anime = dict([
    ("Ведьмак", "Кошмар волка"),
    ("Призрак в доспехах", ["Призрак в доспехах", "Синдром одиночки",
    "Невинность"])
])

# dict() и "ключ=значение"
literature = dict(poem_flowers=flowers)

print(f"flowers = {flowers}\nanime = {anime}\nliterature = {literature}")
```

```
flowers = {'roses': 'red', 'violets': 'blue'}
anime = {'Ведьмак': 'Кошмар волка', 'Призрак в доспехах': ['Призрак в доспехах',
    'Синдром одиночки', 'Невинность']}
literature = {'poem_flowers': {'roses': 'red', 'violets': 'blue'}}
```

- методы для изменения состояния или получения доступа к элементам:

```
# доступ к элементу, если ключа нет - ошибка
print(f'{flowers["violets"] = }')

# при помощи get()
print(f'{flowers.get("magnolias") = }')
```

```
flowers["violets"] = 'blue'
flowers.get("magnolias") = None
```

```
days = ["Пн", "Вт", "Ср", "Чт", "Пт"]

# создадим пустой словарь
numbered_days = {}

# будем добавлять в него элементы в цикле
for num in range(len(days)):
    numbered_days[num] = days[num]

# получим отдельно ключи и значения
# пары из tuple можно сразу получить при помощи метода items()
keys = numbered_days.keys()
values = numbered_days.values()

print(f"Ключи: {keys}\nЗначения: {values}")
```

```
Ключи: dict_keys([0, 1, 2, 3, 4])
Значения: dict_values(['Пн', 'Вт', 'Ср', 'Чт', 'Пт'])
```

- Требование к ключу: возможность хэширования, свойство ключа внутри словаря: уникальность;
- Разобрали, что есть изменяемые (dict, set, list) и неизменяемые (int, float, bool, tuple, str) стандартные типы данных.

ФУНКЦИИ

11.1 Описание лекции

В этой лекции мы расскажем про:

- функции и их смысл;
- локальность переменных;
- параметры и аргументы;
- декораторы.

11.2 Что такое “функция” в Python

В общем случае функцией можно назвать набор связанных инструкций, которые выполняют определенную задачу. Функции во всех языках программирования помогают:

- структурировать код и улучшить читаемость
- переиспользовать код и не изобретать велосипед
- уменьшать количество мест, в которых можно ошибиться при копировании и вставке кода

В Python функции можно разделить на три типа:

- встроенные ([список built-in функций](#))
- именованные (определенные пользователем при помощи `def`)
- анонимные (`lambda`-функции)

Все функции являются объектами типа `function`.

На самом деле, вы уже использовали встроенные функции, например:

- `print()` для вывода данных на экран
- `str()` для создания объектов строкового типа
- `type()` для определения типа объекта

Ими можно пользоваться как черным ящиком, который принимает что-то на вход и творит свою магию. О том, что готовые функции ожидают получить, написано в документации вместе с описанием принципа работы.

Tip: После разбора лекции советую открыть документацию `print()`, например, и разобраться с подробностями работы

Очевидно, стандартные функции дают лишь базовый инструментарий. Реализовать более сложную логику можно самостоятельно.

Пример:

Представим, что необходимо находить количество корней квадратного уравнения - это легко можно реализовать на Python!

```
def count_roots(a, b, c):  
  
    d = b**2 - 4 * a * c  
    if d == 0:  
        return 'один'  
  
    elif d < 0:  
        return 'нет корней'  
  
    else:  
        return 'два'
```

11.3 Именованные функции

11.3.1 Определение

Определение функции позволяет **создать объект** функции. После определения к функции можно будет обратиться по заданному имени и получить результат ее работы.

В Python существует особый синтаксис определения именованных функций:

1. в начале - ключевое слово `def`
2. название функции

Существуют правила и рекомендации по именованию функций:

- **правила:**
 - название не может начинаться с числа
 - можно использовать нижнее подчеркивание `_`, цифры и буквы
- **рекомендации:**
 - `snake_case`: в названии только буквы нижнего регистра, слова разделяются `_`

3. круглые скобки и, дополнительно, параметры внутри (*о них ниже*)
4. двоеточие, переход на новую строку
5. тело функции, выделенное отступом - набор инструкций, который “выполняет работу”

Код в Python организован в блоки и именно отступы дают понять, где у блоков начало и конец. Все тело функции должно располагаться минимум в одном отступе от начала строк.

```
def название_функции(позиционные, *доп_позиционные, именованные, **доп_именованные_
    ↪параметры):
    инструкция 0
    инструкция 1
    ...
    инструкция N
```

Давайте напишем простейшую функцию, которая будет печатать две фразы:

```
# первым делом - def
# далее - имя функции "print_theme"
# после - круглые скобки, сейчас пустые. Потом : и переход на новую строку
def print_theme():

    # тело функции из двух вызовов print(), оба с единообразным отступом
    print('Лекция про функции!')
    print('Тело кончилось')
```

11.3.2 Вызов

После определения функции появляется возможность к ней обращаться (вызывать). Делается это просто: указывается имя функции, круглые скобки и, дополнительно, аргументы в них.

```
# выше была определена print_theme, ее и вызовем
print_theme()
```

```
Лекция про функции!
Тело кончилось
```

11.3.3 Возвращаемое значение

Функция может не только производить какие-то действия, но возвращать наружу результат своей работы. Для этого используется ключевое слово `return`:

```
# определим функцию, которая принимает на вход x
def add_two(x):

    # переменная result - сумма x и 2
    result = x + 2

    # "наружу" возвращается полученное значение
    return result
```

На самом деле, любая функция что-то возвращает:

- указанное значение при наличии `return`
- `None`, когда `return` отсутствует

Возвращение результата означает, что его можно использовать вне функции, например, присвоить полученное значение переменной. Давайте посмотрим, что возвращает `print_theme()` и `add_two()`:

```
# Присвоим результат выполнения функции переменной и посмотрим, что в ней
from_print_theme = print_theme()
print(from_print_theme)
```

```
Лекция про функции!
Тело кончилось
None
```

Как видите, несмотря на отсутствие `return` в коде функции, она действительно возвращает `None`.

Теперь посмотрим на `add_two()`, где возвращаемое значение задано нами:

```
# add_two при вызове ожидает получить число: сложим 2 и 2
from_add_two = add_two(2)
print(from_add_two)
```

4

11.4 Пространства имен и области видимости

Чуть выше была определена `add_two()`, внутри которой инициализировалась переменная `result`. Можно ли получить значение, обратившись к ней вне функции? Нет:

```
# ошибочка!
# тут магическая конструкция try - except, которая "ловит" исключения и не
# дает коду перестать работать из-за ошибки
try:
    print(result)
except NameError as ne:
    print(ne)
```

```
name 'result' is not defined
```

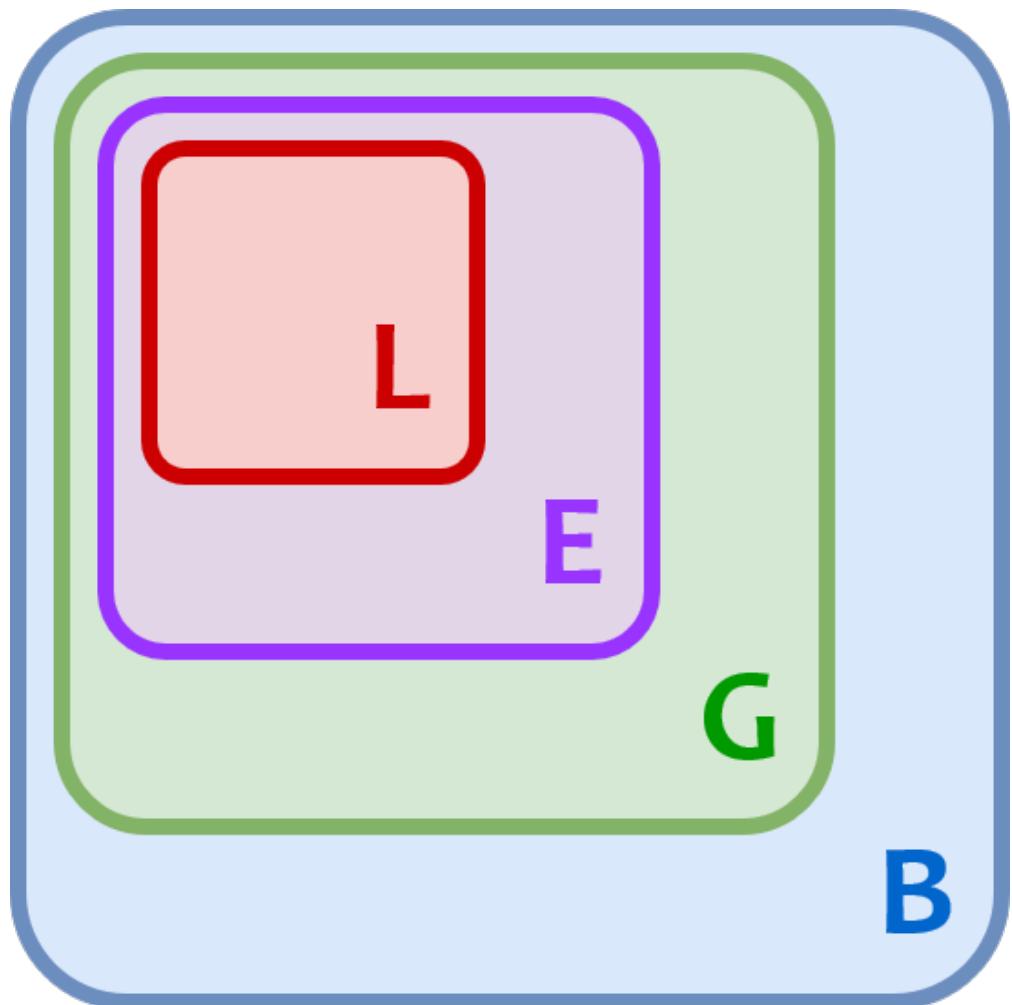
Почему так происходит? Функции обладают своим пространством имен: переменные, которые определены внутри, не видны извне. Однако, функция может получать доступ к переменным, которые определены снаружи. Давайте опишем чуть более формально.

В Python объектам можно давать имя и по этому имени обращаться, эти названия организованы в пространства имен или неймспейсы (*от английского 'namespace'*). Инициализация переменной добавляет в неймспейс название объекта. Неймспейс - набор имен определенных на текущий момент объектов. Представьте себе `dict`, где ключом является строка, а значением - ссылка на объект. Область видимости определяет, переменные из каких пространств имен сейчас доступны. Разберем, как и где Python ищет объекты, к которым обращаются.

Неймспейсы и области видимости можно разделить на несколько типов:

- локальные (Local) - локальные переменные функции, данная область существует до тех пор, пока функция не завершит работу после вызова
- локальные пространства могут быть вложены друг в друга, в таком случае область N (Enclosing) содержит “более глубокую” N + 1. Поиск имени, вызванного на уровне N, начинается с локального неймспейса этого уровня и продвигается “наружу”, то есть на уровни выше
- глобальные (Global) - все объекты, определенные на уровне скрипта
- встроенные (Built-in) - неймспейс содержит все встроенные объекты (функции, исключения и т.д.), поиск в нем осуществляется последним

Если имя не было разрешено даже на уровне Built-in, будет выброшено исключение `NameError`.



```
z = 'name'

def foo():
    # в foo не определена своя z, но она успешно найдется в глобальной области
    print(z)
    a = 10
    b = -5

    def bar():
        # bar успешно напечатает a, при этом значение будет найдено в
        # локальном неймспейсе foo
        # как думаете, где находится print?
        # в built-in, то есть самой "внешней" области
        print(a)

        # создадим переменную b внутри bar
        b = 20

        # будет напечатано 20, так как поиск увенчается успехом в локальном
        # неймспейсе
        print(b)

    bar()

    # если тут раскомментировать следующую строку, будет ошибка: foo не знает
    # ничего про внутренности bar
    # print(b)

foo()
```

```
name
10
20
```

Tip: Хотим заметить, что нужно быть аккуратными при использовании вложенных функций и следить за тем, где и какие переменные определены и меняются. В рамках курса вообще не советуем так делать, чтобы не запутаться. Единственное исключение - декораторы, описанные ниже.

11.5 Параметры

Наша функция `add_two()` или, например, `type()` ожидают, что на вход будут переданы какие-то аргументы для успешной работы, а вот `print()` можно вызвать и с ними, и с пустыми скобками. В начале лекции был представлен скелет функции, сейчас разберем, что же находится в скобках.

Для начала немного формализма:

- при определении функции в скобках пишутся параметры, которые функция может принять
- при вызове функции в скобках указываются аргументы, которые задают значения параметров внутри функции

То есть имеется отображение: аргументы, с которыми вызывается функция -> значения параметров.

Переменные с названиями параметров могут быть использованы внутри тела функции, как будто их значения известны.

11.5.1 Позиционные параметры

Позиционные параметры выглядят как перечисленные внутри скобок названия переменных, используемых внутри функции:

```
# в данном случае есть два позиционных параметра
def foo(arg1, arg2):
    print(arg1, arg2)
```

Данный тип характеризуется следующим:

- позиционные параметры идут первыми после открытых скобок, все именованные строго после них
- важен порядок: отображение аргументов в параметры будет последовательным
- при вызове функции все аргументы, которые ожидает функция, должны быть переданы (откуда иначе Python возьмет значения? вот именно!)

Разберем пример, который суммирует два числа:

```
# в данном случае есть два позиционных параметра
def two_var_sum(var1, var2):

    # функция возвращает вычисленное значение суммы
    return abs(var1) + var2

# порядок важен!
print(two_var_sum(-1, 2), two_var_sum(2, -1))

# можно явно задавать переменные при вызове, в таком случае порядок не играет
# роли. Указывается название параметра и значение после =
print(two_var_sum(var2=2, var1=-1))
```

```
3 1
3
```

А что, если количество входных переменных очень большое или вы заранее не знаете, сколько аргументов будет передано при вызове? Например, вам нужно сложить не 2 числа, а 102? В Python есть специальный синтаксис со звездочками. После позиционных аргументов можно указать `list`, элементами которого станут неограниченное количество переданных позиционных аргументов.

Синтаксис: `имя_функции([поз0, ..., позN,] *поз_список): ... - []` в данном случае обозначают необязательность.

```
# тут позиционных аргументов нет (такое тоже может быть), поэтому сразу
# используется синтаксис со *: список, куда попадут все переданные позиционные
# аргументы, тут называется vars
def many_var_sum(*vars):

    # функция возвращает вычисленное значение суммы
    return sum(vars)

many_var_sum(1, 2, 3, 4, 5, 6)
```

```
21
```

Tip: Совет: лучше передавать все в списках или векторах (о которых расскажут позже)

11.5.2 Именованные параметры

Данные параметры имеют значения по умолчанию.

Синтаксис: имя_функции(все позиционные и *, им0=значение0, им1=значение1, ..., имK=значениеK, **им_словарь): ...

характеризуются следующим:

- в определении идут строго после позиционных параметров
- в определении дано значение по умолчанию через = после имени
- при вызове необязательно передавать - тогда будет использовано значение по умолчанию

```
# тут есть один позиционный параметр и один именованный
def hello(name, phrase='Привет'):

    print(f'{phrase}, {name}')

# передавать значения аргументов можно как в исходном порядке, тогда
# параметру будет присвоено соответствующее значение
hello('Саша', 'Приветствую')

# так и указывая названия параметров
hello(phrase='Здорова', name='Игорь')

# если не указать значение именованного параметра, используется дефолтное
hello('Вася')
```

```
Приветствую, Саша
Здорова, Игорь
Привет, Вася
```

Аналогично позиционным аргументам, если необходимо передать множество именованных параметров, используется синтаксис со звездочками. В данном случае все переданные именованные аргументы, если не определены явно, попадут в dict, указанный после **:

```
def congrats(today, everyone=False, **names):
    """
    Функция может поздравлять людей
    Args:
        today (str): сегодняшняя дата
        everyone(bool): флаг, нужно ли поздравить всех
        names (dict): отображение имя: дата
    """

    if everyone:
        print('Happy Birthday!')

    else:
        for name, date in names.items():
            if date[-5:] == today[-5:]:
                print(f'Happy Birthday, {name}!')  

congrats('2021-09-17', Paul='2001-03-08', Lena='1997-01-31', Mark='1997-09-17')
```

```
Happy Birthday, Mark
```

Что это за комментарий в кавычках внутри функции? Это один из общепринятых способов написания docstring - описания деталей работы функции.

11.6 Анонимные функции

Функции, определенные при помощи `def`, имеют название, по которому можно обратиться, но также существуют и анонимные или неименованные функции. Такие функции могут быть определены при помощи оператора `lambda`. Он позволяет задать входные и выходные данные. Вы можете попробовать самостоятельно почитать [документацию](#).

```
# синтаксис анонимных функций простой
lambda [арг0, ..., аргN]: выражение
```

В определении выражение собой то, что будет возвращено из анонимной функции:

```
lambda x: abs(x)

lambda num, div=2: 'нет' if num % div else 'да'

<function __main__.<lambda>(num, div=2)>
```

В примере выше `lambda`-функции были только созданы в моменте, но не вызваны. Можно сказать, что мы их определили (как с `def`), но не воспользовались и, по сути, сразу потеряли.

Для удобства можно присвоить переменной объект функции и по этому имени к ней обращаться:

```
# сохранили в check_div функцию
check_div = lambda num, div=2: 'нет' if num % div else 'да'

# вызов совершенно обычный
print(check_div(3), check_div(5, 5))

# если без сохранения в переменную:
print((lambda x: abs(x))(-120))
```

```
нет да
120
```

Данные функции можно применять, когда нужна одноразовая функция или лень писать лишнюю строку с `def`.

11.7 Декораторы

Полезный концепт Python - декоратор. Это функция, которая принимает на вход другую функцию и возвращает функцию. Звучит непонятно на первый взгляд. На деле декораторы помогают расширять возможности функций не меняя их кода.

Давайте сразу разберем пример:

```
import time

# на вход декоратор принимает параметр func - оборачиваемую функцию
def time_decorator(func):
```

(continues on next page)

(continued from previous page)

```

# внутри определена функция, которая "заменит" переданную
# wrapped будет засекать, за сколько выполняется переданная функция
def wrapped(*args):
    start = time.time()

    # почему func нет среди аргументов wrapped?
    # все аргументы, которые примет wrapped, тут передаются в func
    result = func(*args)

    end = time.time()
    print(f'Прошло {end - start} секунд')

    return result

# возвращает декоратор тоже функцию. обратите внимание, что
# возвращаемое значение именно объект function: после имени нет круглых
# скобок. если бы они были, возвращался бы результат выполнения wrapped,
# так как wrapped() - вызов функции
return wrapped

```

Выше определена функция many_var_sum(), давайте засекать, сколько она работает:

```

# сохраним обернутую функцию (помним, декоратор возвращает функцию) в
# переменную, обратите внимание, что на вход декоратору передается также
# объект самой функции, а не результат ее работы
many_var_sum = time_decorator(many_var_sum)

# поведение функции поменялось, а код - нет
summed = many_var_sum(10, 0, -120, 333)

```

Прошло 1.1920928955078125e-06 секунд

Отлично, но можно добавить так называемый **синтаксический сахар**: вместо присваивания значения над определением функции можно указать специальный символ @, после которого указывается название декоратора:

```

# теперь, при вызове, stupid_power сразу будет обернута!
@time_decorator
def stupid_power(x, power=5):
    result = 1
    for p in range(power):
        result *= x
    return result

powered = stupid_power(10)

```

Прошло 3.337860107421875e-06 секунд

Таким образом, использования синтаксического сахара с @ и указанием имени декоратора над функцией аналогично вызову stupid_power = time_decorator(stupid_power).

11.8 Что мы узнали из лекции

- Что такое функции и зачем их применять
- Как определить функцию (инструкция `def`)

```
def название_функции(параметры):
    тело функции
    <return возвращаемые_значения>
```

- Отличия позиционных параметров от именованных:
 - порядок указания - *сначала позиционные, потом именованные*
 - значения по умолчанию у именованных позволяют не указывать их при вызове
- Синтаксис со звездочками для получения заранее неизвестного числа позиционных (`*some_list`) и именованных (`**some_dict`) аргументов
 - в `*some_list` стоит **одна** `*`: эта конструкция для получения неограниченного количества позиционных/неименованных аргументов
 - в `**some_dict` **две** `*`: все именованные аргументы, явно не указанные среди параметров, попадут туда
- Что может возвращать функция при помощи `return`
 - `return` обозначает выход из функции и передачу “наружу” результата работы
 - в `return` можно перечислять несколько возвращаемых значений через запятую
- Что такое декораторы и как они работают, щепотку синтаксического сахара с `@`
 - использование декоратора эквивалентно сохранению результата вызова функции-декоратора с аргументом в виде оборачиваемой функции `stupid_power = time_decorator(stupid_power)`

ОБЪЕКТЫ, КЛАССЫ И МЕТОДЫ

12.1 Описание лекции

В этой лекции мы расскажем про:

- основы ООП: что это и зачем?
- инстанс класса
- на что указывает `self`
- методы классов в уже знакомых примерах
- атрибуты классов

12.2 ООП: Объектно-Ориентированное Программирование

Что же такое объектно-ориентированное программирование? Судя по названию, ключевую роль здесь играют объекты, на которые ориентируется дальнейший процесс программирования. Если мы взглянем на реальный мир, то для нас он предстанет в виде множества объектов, обладающих определенными свойствами, взаимодействующих между собой и вследствие этого изменяющимися. Эта привычная для взгляда человека картина мира была перенесена в программирование. Python - это объектно-ориентированный язык программирования, и **все в нем является объектами**.

Объект - это набор данных и инструкций в памяти компьютера, в том числе:

1. Тип объекта;
2. Данные, формирующие объект (контент внутри него);
3. Методы.

Как определить тип мы уже знаем. И с объектами на самом деле мы знакомы с самого начала - ведь даже целочисленная переменная 5, имеющая `type(5) = int`, представляет собой объект. Напомним, что говорилось в курсе ранее: **методы - это те же функции, только существующие не сами по себе, а являющиеся частью класса**.

```
example_list = list(["что-то", "с"])
print(f"Метод append имеет тип {example_list.append}")

# вот так можно вызвать метод, изменяя сам экземпляр класса
example_list.append("чем-то")

# а вот так можно обратиться к атрибуту конкретного объекта
```

(continues on next page)

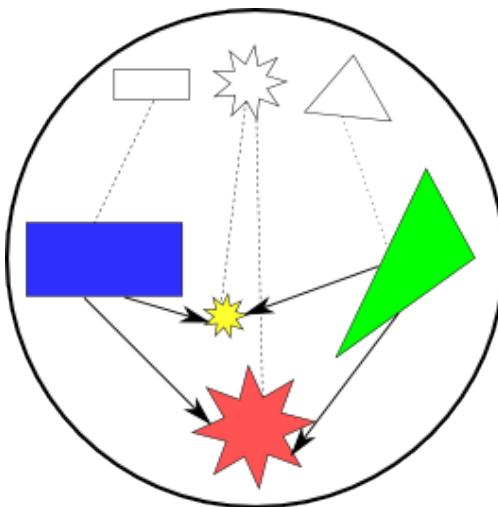


Fig. 12.1: Так можно проиллюстрировать соотношение “класс - объекты”. Сверху указаны общие прототипы (прямоугольник, звезда, треугольник), снизу - уникальные экземпляры, со своим цветом и деталями.

(continued from previous page)

```
attr_value = example_list.__doc__
print(f"Значение атрибута: {attr_value}")
```

Метод `append` имеет тип <built-in method `append` of `list` object at 0x7f2022e67640>
Значение атрибута: Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list.
The argument must be an iterable if specified.

`list` - это класс, и в первой строчке кода выше создается **объект, или экземпляр класса** `example_list`. Он хранит свой набор данных (контент). Метод `append` общий для всех списков, однако меняет только конкретный экземпляр. Обратите внимание: этот метод ничего не возвращает, он меняет сам объект и его контент. Не все методы работают схожим образом, и придется запоминать принцип работы самых распространенных методов. К примеру, сложение чисел или строк внутри Python также осуществляется с помощью методов, однако это не меняет саму переменную, как было рассказано в более ранних лекциях:

```
# создадим два объекта класса int
a = 3
b = int(4)

# неявно вызовем метод сложения
a + b
# проверим, что содержимое объектов не изменилось
print(f"{a = }, {b = }")
```

```
a = 3, b = 4
```

Конкретно в данном случае логика работы обусловлена свойствами `mutable/immutable` объектов, о чем также упоминалось в одной из предшествующих лекций. `int` неизменяем, и потому при вызове метода сложения возвращает новый объект - ведь самого себя изменить не получится. `list` же может меняться, и `append` меняет содержимое `inplace`.

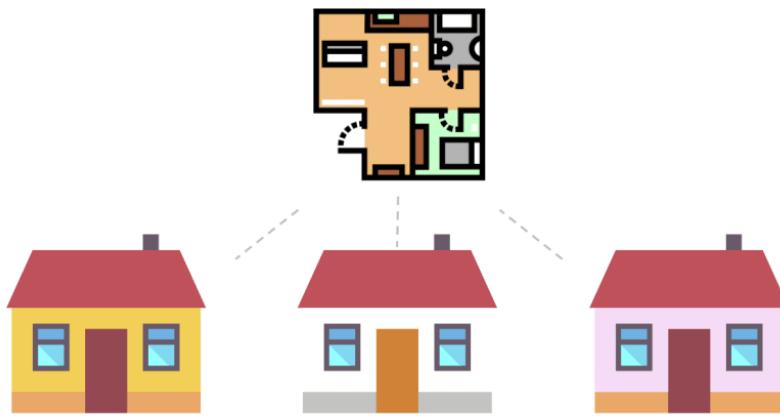


Fig. 12.2: Другой житейский пример - по одному чертежу (классу) можно построить разные (но похожие!) дома, в которых разные семьи, убранство, разные обои.

12.3 Определение классов

Давайте попробуем создать свой первый класс:

```
# определим классы
class List:
    def __init__(self, initial_content = None):
        # указывать в виде значения по умолчанию mutable-объекты неправильно
        if initial_content is None:
            initial_content = []
        self.content = initial_content

    def append(self, new_element):
        # изменение производится inplace, с заменой собственного контента
        self.content = self.content + [new_element]

class Int:
    def __init__(self, value):
        self.value = value

    def add(self, second_value):
        # изменение не производится, создается новый объект и он же возвращается
        return Int(self.value + second_value.value)

# воспроизведем примеры кода выше для List
example_list = List(["что-то", "с"])
print(f"{example_list.content = }")

example_list.append("чем-то")
print(f"{example_list.content = }")
print(f"type(example_list) = {type(example_list)}")

# и для Int
a = Int(3)
b = Int(4)
```

(continues on next page)

(continued from previous page)

```
a.add(b)
print(f"После простого сложения: {a.value = }, {b.value = }")

c = a.add(b)
print(f"После записи в переменную результата: {c.value = }")
```

```
example_list.content = ['что-то', 'с']
example_list.content = ['что-то', 'с', 'чем-то']
type(example_list) = <class '__main__.List'>
После простого сложения: a.value = 3, b.value = 4
После записи в переменную результата: c.value = 7
```

Что важно заметить и отметить в этом примере:

1. Как объявить класс? С помощью ключевого слова `class`, указания названия класса (таким образом создается новый тип данных) и перечисления методов:

```
class ИмяКласса:
    <код_тела_класса>
```

1. Обязательно определить метод `__init__`. Это метод, вызываемый при создании нового объекта класса и отвечающий за определение его контента. В нем через запятую, как и в обычной функции, перечислить необходимые переменные и, по возможности, их значения по умолчанию;
2. Каждый метод должен первым аргументом принимать **экземпляр объекта** (по соглашению он называется `self`). При этом при вызове метода Python автоматически подставит сам объект, делать это руками не нужно. По сути под капотом происходит следующее:

```
# вызываем метод класса - не объекта! - и передаем первым аргументом объект,
# который при обычных обстоятельствах будет записан в self
result = Int.add(Int(3), Int(4))
print(f"{result.value}")
```

7

1. **Атрибутом называется переменная, хранящаяся в экземпляре класса;**
2. В рамках методов атрибуты объекта доступны с использованием ключевого слова `self`. К примеру, `self.data`;
3. Вне методов атрибуты доступны у экземпляров через точку: `some_object.attr_name`;
4. Для создания объекта класса нужно указать имя класса (типа) и передать аргументы: `Int(3)` - как раз оно;
5. Методы как могут возвращать что-либо (в том числе новый объект того же класса), так и ничего не возвращать.

В заключение еще одна картинка, иллюстрирующая суть класса, экземпляра, атрибутов и методов:

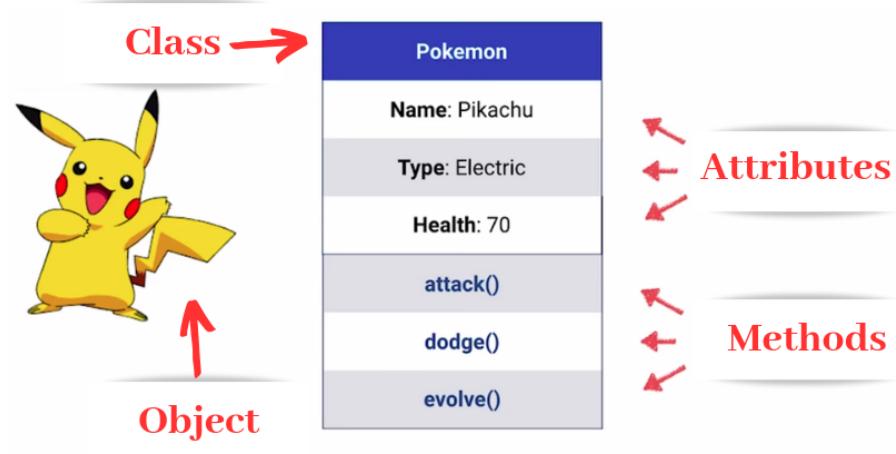


Fig. 12.3: Who's that pokemon?

12.4 Пример использования для хранения состояния

В курсе будет встречаться подобный паттерн, поэтому хочется остановиться поподробнее. Допустим, есть некоторая среда, заданная состоянием, и есть объекты, существующие в этой среде, как-то взаимодействующие между собой (и со средой). Есть набор функций, который осуществляет эти взаимодействия. Тогда удобно ввести некоторый объект `system` класса `System`, который хранит эту информацию, и передавать его как параметр в методы расчета чего-либо (никакой конкретной логики тут не заложено, не пытайтесь понять, за что отвечает `alpha` и прочие переменные - пример абстрактный):

```
# класс системы, с ее параметрами и состоянием
class System:
    def __init__(self, param_vector, alpha, gamma, multiplier = 2.0, energy = 0.0):
        self.param_vector = param_vector
        self.alpha = alpha
        self.gamma = gamma
        self.multiplier = multiplier
        self.energy = energy

    # начальное состояние
    self.system_state = [
        (alpha - gamma) * multiplier * param
        for param
        in self.param_vector
    ]

    def first_action(self, object):
        # взаимодействие с одним объектом: увеличение энергии, изменение состояния
        self.gamma -= object.energy
        self.energy += object.energy
        self.system_state = [param - self.alpha * self.energy for param in self.
        system_state]

    def second_action(self, list_of_object):
        # взаимодействие с несколькими объектами - увеличение их энергии
```

(continues on next page)

(continued from previous page)

```

for obj in list_of_object:
    obj.energy += self.gamma * self.multiplier

class Object:
    def __init__(self, energy = 10.0):
        # объект хранит только один атрибут
        self.energy = energy

# симуляция одного цикла в системе
def one_system_cycle(system: System, objects = None):
    if objects is not None:
        system.second_action(objects)
        # убираем один последний объект
        objects = objects[:-1]
    else:
        objects = [Object(val) for val in range(5)]
        for obj in objects:
            system.first_action(obj)
    return objects

params = [1.0, 3.0, 4.15, 0.0]
alpha = 5
gamma = 0.18
system = System(params, alpha, gamma, energy = 10)

# изначально объектов нет, они появятся после первого цикла
objects = None

for cycle_num in range(1, 4):
    print(f"Состояние системы на итерации {cycle_num}: {system.system_state}.", end="\r")
    objects = one_system_cycle(system, objects)
    print(f"Всего объектов: {len(objects)}", end="\r")

    total_object_energy = sum([obj.energy for obj in objects])
    print(f"Суммарная энергия объектов: {total_object_energy}")

```

Состояние системы на итерации 1: [9.64, 28.92, 40.00600000000001, 0.0].	Всего ↴
↪объектов: 5 Суммарная энергия объектов: 10	
Состояние системы на итерации 2: [-340.36, -321.08, -309.994, -350.0].	Всего ↴
↪объектов: 4 Суммарная энергия объектов: -72.56	
Состояние системы на итерации 3: [-340.36, -321.08, -309.994, -350.0].	Всего ↴
↪объектов: 3 Суммарная энергия объектов: -114.84	

Обратите внимание на то, что функция `one_system_cycle` не возвращает `system` - ведь мы прямо во время итераций меняем ее состояние, и в последующие моменты времени эти изменения сохраняются, то есть состояние остается. Таким образом, эту систему (среду) - а именно объект, экземпляр класса `System` - как параметр **можно передавать в десятки функций, и каждая из них будет видеть актуальное состояние**.

12.5 Что мы узнали из лекции

- Объекты одного класса отличаются между собой атрибутами, которые придают уникальности;
- Методы классов позволяют описывать логику взаимодействия, **обновлять состояние**;
- Для объявления класса нужно определить **как минимум один метод `__init__`**;
- Классы без методов можно использовать как **удобное хранилище разнородной информации** о схожих объектах (описать разных студентов и их оценки).

Это далеко не все, что можно написать по теме классов и объектов, однако это тот необходимый минимум, что потребуется для прохождения курса. Больше примеров и деталей можно найти, например, по ссылкам: [пример с несколькими взаимодействующими объектами](#), что такое наследование и зачем оно нужно.

ФИНАЛЬНАЯ ЛЕКЦИЯ ОБО ВСЕМ И НИ О ЧЕМ

13.1 Описание лекции

В этой лекции будет:

- рассказ о типовых ошибках при прохождении курса;
- разбор примеров кода из курса;
- дополнение для самых любознательных о Python, ML и вообще.

13.2 Заметки по прохождению курса

Хоть в пройденных лекциях по Python и содержится большое количество материала, проиллюстрированного примерами, с указанием на дополнительные темы для самостоятельного изучения – это все еще малая доля того, с чем вам придется столкнуться, если захочется продолжить развитие в теме ML/QML. Да, это грустно, но такова правда жизни. А еще вы столкнетесь с кучей проблем и ошибок во время прохождения самого курса. Если вы **попытались разобраться** с чем-либо, но у вас не получилось – приходите к нам в сообщество в Slack, задавайте вопросы. Не забывайте и о взаимопомощи другим студентам – если материал или тема дались вам легко, то кому-то они могут показаться адски сложными.

Помните, что абсолютное большинство возникающих ошибок в Python неуникально, и вы не первый, кто наступил на эти грабли. Не стесняйтесь пользоваться поисковиками для нахождения решений. Но перед этим обязательно прочтайте текст ошибки, попробуйте осознать, о чём Python хочет вам сказать - в этом языке программирования одни из самых понятных и доступных подсказок и указателей на проблемы. К примеру, `IndexError: list index out of range` явно говорит о том, что во время обращения к i-тому объекту некоторого списка (который, кстати, будет указан в так называемом трейбеке, от англ. traceback) возник выход за границы `list`. Возьмем для примера список `sample_list` чисел длины 5. Что произойдет, если обратиться к седьмому объекту `sample_list[7]`?

Обратите внимание на стрелочку слева, указывающую на строку номер 5 (кстати, в Jupyter Notebook для каждой отдельной ячейки можно включить нумерацию строк для удобства навигации с помощью клавиши 1). Это место возникновения ошибки, та строка, на которой Python не смог выполнить инструкцию. Как можно решить проблему? Для начала **методом пристального взгляда** поймите, написано ли в коде то, что вы хотели – очень часто код, к примеру, копируется (хотя нужно избегать этого с помощью написания функций), но переменные в нем не меняются в силу невнимательности. Если все переменные в строке те же, что и должны быть, и ошибка не ясна, то следует заняться **отладкой кода**.

Самый простой способ отладки в Jupyter – это расстановка `print`'ов до проблемной строки. Да-да, так кустарно и грубо. Есть более продвинутые технологии (например, **отладчики** в IDE, то есть средах разработки), однако они выходят за рамки данного курса. Очевидный совет - нужно размещать такой `print`, который помогает проверить какую-либо **гипотезу** о происхождении ошибки. Предложение таких гипотез – это отдельный навык,

```

Ввод [36]: sample_list = [1, 2, 3, 4, 5]

Ввод [39]: print('что-то')
print('И еще что-то')
example_str = 'а тут я ' + ' просто складываю строки'
i_umnozhau_chisla = 3*6
result = sample_list[7]
# а после этого еще оставляю пару строк
b = 'с какими то каракулями'

что-то
И еще что-то

-----
IndexError Traceback (most recent call last)
<ipython-input-39-567b10e20d31> in <module>
      3 example_str = 'а тут я ' + ' просто складываю строки'
      4 i_umnozhau_chisla = 3*6
----> 5 result = sample_list[7]
      6 # а после этого еще оставляю пару строк
      7 b = 'с какими то каракулями'

IndexError: list index out of range

```

Fig. 13.1: Получим ошибку

который приходит с опытом, так что не переживайте, что поначалу получается плохо “угадывать” и “понимать” ошибки.

В случае обращения по неверному индексу (`list index out of range`) логично предположить, что индекс оказался больше максимально возможного значения. Давайте выведем длину списка и индекс, по которому хотим обратиться:

И исходя из аргументов выше становится очевидно, в чем именно проблема. Конечно, рассмотренный случай крайне тривиален, однако он показывает базовые приемы поиска причин ошибок. Отметим также такой полезный прием как разбитие проблемной строки кода на подстроки, то есть разнесение кода на отдельные изолированные куски. В данном примере в отдельную переменную был записан индекс. Однако попробуйте угадать, в каком месте происходит ошибка в коде ниже:

Это уже более сложный пример, когда трейсбек состоит из двух блоков. Указатель на ошибку как бы “пропаливается” внутрь вызываемой функции и показывает, что не так. В данном случае ошибка `TypeError: unsupported operand type(s) for +: 'int' and 'str'` указывает на несовместимость типов операндов в операции сложения. Слева `int`, справа `str` – ну и как же их суммировать?

Однако исходная строчка, порождающая ошибку, крайне сложна. Мы вызываем функцию (`print`) от функции (`first_f`) от функции... Будь логика в коде чуть сложнее – легко можно было бы запутаться, что именно породило ошибку (главное обращайте внимание на заголовки трейсбека, указывающие на названия методов и функций). С разбиением все проще:

На этом остановимся, блок обязательно будет дополняться по мере возникновения типовых ошибок и способов их поимки. Главное помните, что можно распечатать и аргументы функции/метода, и атрибуты класса (в том числе скрытые/промежуточные состояния квантовой среды), и результаты расчетов.

```

Ввод [36]: sample_list = [1, 2, 3, 4, 5]

Ввод [40]: print('что-то')
print('И еще что-то')
example_str = 'а тут я ' + ' просто складываю строки'
i_umnozhau_chisla = 3*6
target_index = 7
print(f'ОТЛАДКА: {len(sample_list) = }, {target_index =}', )
result = sample_list[target_index]
# а после этого еще оставляю пару строк
b = 'с какими то каракулями'

что-то
И еще что-то
ОТЛАДКА: len(sample_list) = 5, target_index = 7

-----
IndexError Traceback (most recent call last)
<ipython-input-40-c3b40d282224> in <module>
      5 target_index = 7
      6 print(f'ОТЛАДКА: {len(sample_list) = }, {target_index =}', )
----> 7 result = sample_list[target_index]
      8 # а после этого еще оставляю пару строк
      9 b = 'с какими то каракулями'

IndexError: list index out of range

```

Fig. 13.2: Строчку с вашим отладочным выводом легко потерять - для удобства можно добавлять какие-то символы или текст.

```

Ввод [41]: def first_f(inp):
           return inp * 3

def second_f(inp):
    return inp - 2

def third_f(inp):
    return inp + 'Abracadabra'

print(first_f(third_f(second_f(12))))


-----
TypeError Traceback (most recent call last)
<ipython-input-41-16694fbb215b> in <module>
      8     return inp + 'Abracadabra'
      9
---> 10 print(first_f(third_f(second_f(12)))))

<ipython-input-41-16694fbb215b> in third_f(inp)
      6
      7 def third_f(inp):
----> 8     return inp + 'Abracadabra'
      9
     10 print(first_f(third_f(second_f(12))))


TypeError: unsupported operand type(s) for +: 'int' and 'str'

```

Fig. 13.3: Трейсбек ошибки предательски указывает на функцию, при вызове которой возникла ошибка - `third_f`

```
Ввод [42]: def first_f(inp):
    return inp * 3

def second_f(inp):
    return inp - 2

def third_f(inp):
    return inp + 'abracadabra'

tmp1 = second_f(12)
tmp2 = third_f(tmp1)
tmp3 = first_f(tmp2)

print(tmp3)

-----
TypeError                                     Traceback (most recent call last)
<ipython-input-42-5e2fbb8ada7f> in <module>
      9
     10 tmp1 = second_f(12)
--> 11 tmp2 = third_f(tmp1)
     12 tmp3 = first_f(tmp2)
     13

<ipython-input-42-5e2fbb8ada7f> in third_f(inp)
      6
      7 def third_f(inp):
----> 8     return inp + 'abracadabra'
      9
     10 tmp1 = second_f(12)

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Fig. 13.4: Нууу! Указатель в виде стрелочки прямо рапортует о том, что проблема в вычислении `third_f`. Дальше можно было бы добавить `print(tmp1)` для того, чтобы увидеть входные параметры и попытаться понять, в чем же проблема.

13.3 Примеры кода, не разобранные в лекциях

Ранее весь код, который упоминался в лекциях (за вычетом примера из первой) был написан на чистом Python без сторонних библиотек, которые мы так нахваливали. В курсе же, разумеется, они встретятся. Полезно поговорить о двух вещах - об **импортах** библиотек и об их алиасах.

Для того, чтобы использовать какой-либо объект из сторонней библиотеки, будь то функция, класс или даже словарь (например, с физическими константами), необходимо знать, в какой библиотеке и по какому пути он лежит. Все это описано в документации, а также в примерах кода в курсе. Не переживайте, если они плохо запоминаются - чаще всего их копируют, и со временем в голове откладывается паттерн. И все же для того, чтобы получить доступ к библиотеке, нужно сделать следующее – рассмотрим на примере `numpy`, библиотеки для алгебраических расчетов. Сначала следует ключевое слово `import`, затем через пробел название пакета. Опционально через `as` можно добавить алиас – название, под которым можно будет обращаться в коде для доступа к функциям. У многих пакетов есть общепринятые алиасы, с которыми все знакомы. Для `numpy` это `np`. Итак,

```
import numpy as np

np_example_array = np.array([1, 2, 3])
print(f"np_example_array.shape = {np_example_array.shape}")

abs_res = np.abs(-2)
print(f"abs_res = {abs_res}")
```

```
np_example_array.shape = (3,)
abs_res = 2
```

По сути обращение схоже с объявлением объекта класса или же каким-то методом – ведь можно импортировать не целый пакет, а его часть!

```
from numpy import zeros

zeros_matrix = zeros(shape=(1, 3))
print(f"zeros_matrix = {zeros_matrix}")
```

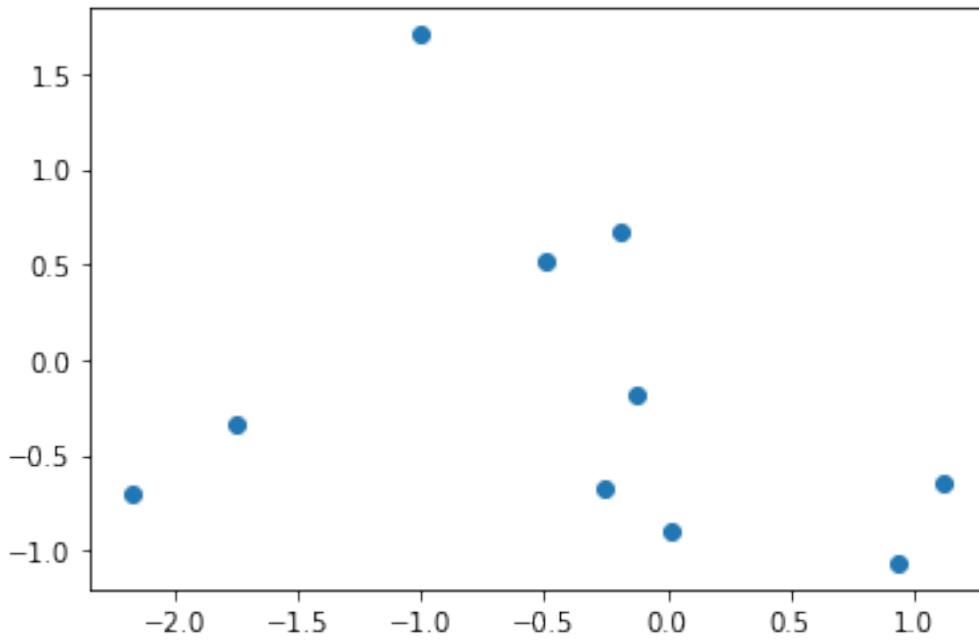
```
zeros_matrix = array([[0., 0., 0.]])
```

Здесь `zeros` – **функция**, формирующая матрицу из нулей заданного размера.

Или, к примеру, для отрисовки визуализаций часто используют следующую библиотеку (мы уже с ней сталкивались):

```
import matplotlib.pyplot as plt

plt.scatter(np.random.randn(10), np.random.randn(10));
plt.show();
```



Tip: В коде выше в последних двух строчках стоит ; . Это необязательный символ, означающий конец команды. В большинстве других языков программирования необходимо всегда в конце строки оставлять этот символ, однако в питоне сигналом о конце инструкции служит перенос строки. Почему же здесь они поставлены? Во-первых, чтобы показать, что они означают – чтобы вы не переживали, когда встретите подобное. Во-вторых, plt . show () возвращает объект после вызова метода. Jupyter же по умолчанию выводит последнее возвращенное значение. Попробуйте убрать точку с запятой и посмотреть в своем ноутбуке, что выйдет. Таким образом, в данном случае окончание команды ; говорит, что возвращаемая переменная никуда не будет записана и будет удалена - поэтому при ее наличии подпись объекта не возвращается.

В коде сначала происходит импорт, затем вызывается отрисовка точек (см. документацию тут), где координаты X и Y получаются случайно из нормального распределения от -1 до 1. Но сам по себе вызов scatter () лишь добавляет точки на график, но не отрисовывает его - за это отвечает вызов plt . show () . Стоит заметить, что plt хранит в себе “полотно”, на котором рисует, и результат добавления точек через scatter не нужно записывать в какую-либо переменную – все это хранится во внутреннем состоянии самой библиотеки. Помните урок про классы и их объекты, задающие состояние системы? Здесь ровно то же самое! Более того, аналогичный функционал будет у библиотеки для квантовых вычислений, с которой вы познакомитесь далее по курсу (под аliasом qml). Теперь, надеемся, картинка начинает складываться!

13.4 Блок дополнительных ссылок

Ниже представлен набор ссылок, которые предлагается просмотреть для расширения кругозора и лучшего понимания материала. Это бонусный материал, необязательный к изучению. Но если вас захватила идея получше познакомится со своим инструментом, то почему нет?

1. Курс по Python в виде примеров кода с очевидными и не очень механиками языка. Хорошо подойдет как дополнение к примерам в курсе;
2. realpython – сайт с большим количеством разнообразных уроков по Python на английском;
3. Огромный список ссылок на разные статьи по всем темам - от основ Python и до продвинутых приемов в разработке;

4. И другой – с набором ссылок на код разных программ. Игры, видеопроигрыватели и прочее - все тут;
5. Простой отладчик на случай, если начали путаться в своем коде, а продвинутые инструменты кажутся очень сложными и громоздкими.

Tip: Как ставить новые библиотеки в Python – вот хотя бы тот же отладчик из пункта 5? Вспоминайте совет – гугл ваш лучший друг. “how to install python library” (да, прямо дословно library = библиотека) – и вы на коне!

13.5 Вместо заключения

Что ж, вот и подошел к концу первый блок курса. Его можно назвать самым сложным – нет, не с точки зрения материала и его научности, но по количеству усилий, которые необходимо приложить, чтобы не сбиться с пути и продолжить обучение несмотря на трудности. Именно тут большая часть людей машет рукой и закрывает курс. Если ты, дорогой студент, читаешь это, то мы безумно рады за твои достижения и подвиги, и желаем дальнейших успехов в прохождении курса!



О БЛОКЕ “ЛИНЕЙНАЯ АЛГЕБРА И NUMPY”

Блок задуман для обсуждения основных понятий и идей линейной алгебры и представляет собой краткое пособие. Данный блок **не является** полноценным учебником по линейной алгебре. Для более детального погружения в тему рекомендуем учебник Тыртышникова Е. Е. «Матричный анализ и линейная алгебра».

В рамках блока мы:

- обсудим основные понятия и идеи, такие как **векторы и матрицы**;
- познакомимся с инструментами для работы с векторами и матрицами;
- попробуем что-нибудь посчитать!

Обозначения:

- a – число (скаляр);
- \mathbf{a} – вектор;
- \mathbf{A} – матрица.

Полезные материалы:

- MathProfi: [Аналитическая геометрия](#).
- 3Blue1Brown: [Essence of linear algebra](#).

ВЕКТОРЫ

15.1 Определение

Вектор – это термин, который имеет несколько различных интерпретаций: математическую, геометрическую, физическую и др. Точный смысл данного термина зависит от контекста.

Формально вектор определяется как элемент векторного пространства – множества, на котором определены операции **сложения и умножения вектора на число (скаляр)**, которые должны удовлетворять **8 аксиомам**.

Для простоты понимания рассмотрим знакомую нам со школьных времен прямоугольную (Декартову) систему координат на плоскости – две перпендикулярные друг другу оси x и y , выбранные на них единичные векторы (орты) \mathbf{e}_1 , \mathbf{e}_2 и начало координат.

Вектор \mathbf{a} в такой системе координат можно записать следующим образом: $\mathbf{a} = a_1\mathbf{e}_1 + a_2\mathbf{e}_2 = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$

Замечание 1: Координаты вектора не определяют однозначно его положение на плоскости, а лишь положение конца вектора относительно его начала. Например, вектор $\begin{pmatrix} 3 \\ 4 \end{pmatrix}$ может быть отложен как из точки начала координат $(0, 0)$ с концом в точке $(3, 4)$, так и из произвольной точки, например, $(1, 1)$ с концом в точке $(4, 5)$. Оба этих вектора соответствуют 3 ортам на оси x и 4 на оси y . Обычно, если не сказано иное, предполагается, что вектор отложен из начала координат.

Замечание 2: Вектор можно представить либо как вектор-столбец $\begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$, либо как вектор-строку $(a_1 \ a_2)$. Здесь и далее под вектором будем подразумевать вектор-столбец, если не сказано иного.

Понятие вектора на плоскости можно обобщить на 3-мерное пространство, и, в общем случае, на n -мерное пространство (которое уже не получится визуализировать):

$$\mathbf{a} = a_1\mathbf{e}_1 + a_2\mathbf{e}_2 + \dots + a_n\mathbf{e}_n = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}$$

15.2 Операции с векторами

Как было сказано ранее, в формальном определении существуют две основные операции над векторами:

- сложение:

$$\begin{aligned}\mathbf{a} + \mathbf{b} &= (a_1 + b_1)\mathbf{e}_1 + (a_2 + b_2)\mathbf{e}_2 + \dots + (a_n + b_n)\mathbf{e}_n = \\ &= \begin{pmatrix} a_1 \\ a_2 \\ \dots \\ a_n \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \dots \\ a_n + b_n \end{pmatrix}\end{aligned}$$

- умножения вектора на число (скаляр):

$$\lambda\mathbf{a} = \lambda a_1\mathbf{e}_1 + \lambda a_2\mathbf{e}_2 + \dots + \lambda a_n\mathbf{e}_n = \lambda \begin{pmatrix} a_1 \\ a_2 \\ \dots \\ a_n \end{pmatrix} = \begin{pmatrix} \lambda a_1 \\ \lambda a_2 \\ \dots \\ \lambda a_n \end{pmatrix}$$

Операцию умножения вектора на число можно интерпретировать геометрически, как сжатие / растяжение вектора.

Используя эти две операции, мы можем считать **линейные комбинации векторов**: $\alpha_1\mathbf{a}_1 + \alpha_2\mathbf{a}_2 + \dots + \alpha_n\mathbf{a}_n$, где $\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$ – векторы, а $\alpha_1, \alpha_2, \dots, \alpha_n$ – числа.

15.3 Норма (длина) вектора

В линейной алгебре для обобщения понятия длины вектора используется термин **норма**. Можно сказать, что понятия длины и нормы эквивалентны.

Формально норма определяется как функционал в векторном пространстве, удовлетворяющий [3 аксиомам](#) и отображающий элементы этого пространства (векторы) в множество неотрицательных вещественных чисел.

Данному определению нормы удовлетворяет множество функционалов, но мы будем рассматривать наиболее часто используемый – **Евклидову норму**.

Для простоты понимания рассмотрим вектор на плоскости. С геометрической точки зрения он представляет собой направленный отрезок. Направленность вектора никак не влияет на его длину, поэтому, при расчете длины, мы можем работать с ним как с отрезком на плоскости и посчитать длину по школьным формулам. Заметим тот факт, что координаты вектора соответствуют числам, умноженным на орты координатных осей, поэтому формула длины вектора выглядит следующим образом: $\|\mathbf{a}\| = \sqrt{a_1^2 + a_2^2}$. Соответственно, в общем случае формула выглядит следующим образом: $\|\mathbf{a}\| = \sqrt{a_1^2 + a_2^2 + \dots + a_n^2}$

15.4 Скалярное произведение

Одной из самых распространенных операций над двумя векторами является так называемое **скалярное произведение**, результатом которого является число (скаляр) – отсюда и название операции.

Замечание: Кроме скалярного произведения существует также **векторное произведение** над парой векторов, результатом которого являются вектор. Также существует **смешанное произведение** над тройкой векторов, результатом которого является число. Данные операции в рамках курса рассматриваться не будут.

Скалярное произведение используется в определении длины векторов и угла между ними. Данная операция имеет два определения:

- алгебраическое: $\mathbf{a} \cdot \mathbf{b} = a_1b_1 + a_2b_2 + \dots + a_nb_n$

- геометрическое: $\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta$, где θ – угол между векторами \mathbf{a} и \mathbf{b} .

Используя оба эти определения можно вывести формулу для расчета косинуса угла между векторами:

$$\cos \theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} = \frac{a_1 b_1 + a_2 b_2 + \dots + a_n b_n}{\sqrt{a_1^2 + a_2^2 + \dots + a_n^2} \sqrt{b_1^2 + b_2^2 + \dots + b_n^2}}$$

С помощью данной формулы можно прийти к одному из главных свойств скалярного произведения, которое заключается в том, что **два вектора перпендикулярны тогда и только тогда, как их скалярное произведение равно 0**: $\mathbf{a} \perp \mathbf{b} \Leftrightarrow \cos \theta = 0 \Leftrightarrow \mathbf{a} \cdot \mathbf{b} = 0$

Скалярное произведение можно использовать для расчета нормы вектора следующим образом: $\|\mathbf{a}\| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$

15.5 Линейная независимость

Одним из основополагающих понятий линейной алгебры является **линейная зависимость/независимость**.

Для определения данного понятия рассмотрим набор из нескольких векторов. Набор векторов является **линейно зависимым**, если существует такая ненулевая линейная комбинация векторов данного набора (как минимум один элемент данной комбинации не равен 0) равная нулевому вектору (вектор, состоящий только из 0):

$$\lambda_1 \mathbf{a}_1 + \lambda_2 \mathbf{a}_2 + \dots + \lambda_m \mathbf{a}_m = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

$$\exists \lambda_1 \neq 0 \vee \lambda_2 \neq 0 \vee \dots \vee \lambda_m \neq 0$$

Если набор векторов не является линейно зависимым, то есть не существует ненулевой линейной комбинации векторов данного набора равной нулевому вектору, то такой набор векторов называется **линейно независимым**.

Пример:

- линейно независимый набор векторов (только нулевые коэффициенты линейно комбинации приводят к нулевому вектору):

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \rightarrow 0 \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + 0 \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + 0 \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

- линейно зависимый набор векторов (существуют ненулевые коэффициенты линейно комбинации, которые приводят к нулевому вектору):

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 0 \end{pmatrix} \rightarrow 2 \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + 3 \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} - 1 \cdot \begin{pmatrix} 2 \\ 3 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Из определения линейной зависимости можно вывести следующее свойство: Набор векторов линейно зависим тогда и только тогда, когда один из элементов этого набора может быть выражен через оставшиеся.

Замечание: Если векторы рассматривать как какие-то характеристики объектов, то линейную зависимость можно интерпретировать как избыточность данных.

С помощью понятия линейной независимости вводится понятие **размерности** векторного пространства – это максимальное число линейно независимых векторов в нем.

15.6 Базис

В формальном определении вектора не присутствуют никакие его количественные интерпретации, а лишь две операции над векторами и 8 аксиом.

Откуда же появляются количественные измерения? Для того, чтобы это стало понятно, нужно ввести понятия **базиса**.

Базис – это конечный набор векторов в векторном пространстве, такой, что любой вектор этого пространства может быть единственным образом представлен в виде линейной комбинации векторов этого набора.

Вспомним один из примеров выше, где мы представляли прямоугольную систему координат на плоскости и единичные векторы (орты) $\mathbf{e}_1, \mathbf{e}_2$. $\mathbf{e}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \mathbf{e}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ В данном примере мы разложили произвольный вектор \mathbf{a} следующим образом:

$$\mathbf{a} = a_1 \mathbf{e}_1 + a_2 \mathbf{e}_2 = a_1 \begin{pmatrix} 1 \\ 0 \end{pmatrix} + a_2 \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} a_1 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ a_2 \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$$

Получается, что орты $\mathbf{e}_1, \mathbf{e}_2$ являются **базисом двумерного векторного пространства**, и с помощью линейной комбинации этих векторов мы можем единственным образом представить любой вектор этого пространства.

Возникает вопрос, данный базис является единственным в двумерном пространстве, или нет?

Ответ на этот вопрос – нет. На самом деле можно взять два любых вектора (**почти любых**), и они тоже будут являться базисом, при условии, что любой вектор можно разложить с помощью их линейной комбинации.

Пример.

Пусть у нас есть вектор $\begin{pmatrix} 2 \\ 3 \end{pmatrix}$ в базисе единичных орт, и мы хотим разложить его по другому базису $\begin{pmatrix} -2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ -3 \end{pmatrix}$:

$$\alpha_1 \begin{pmatrix} -2 \\ 0 \end{pmatrix} + \alpha_2 \begin{pmatrix} 0 \\ -3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

Откуда мы можем найти, что $\alpha_1 = -1, \alpha_2 = -1$:

$$-1 \begin{pmatrix} -2 \\ 0 \end{pmatrix} + -1 \begin{pmatrix} 0 \\ -3 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$$

Таким образом, вектор $\begin{pmatrix} 2 \\ 3 \end{pmatrix}$ в базисе единичных орт представляется как $\begin{pmatrix} -1 \\ -1 \end{pmatrix}$ в базисе $\begin{pmatrix} -2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ -3 \end{pmatrix}$.

Но для базиса, как было сказано ранее, подойдет **не любой набор векторов**.

Например через базис $\begin{pmatrix} -2 \\ 0 \end{pmatrix}, \begin{pmatrix} -3 \\ 0 \end{pmatrix}$ нельзя разложить вектор $\begin{pmatrix} 2 \\ 3 \end{pmatrix}$, поэтому данный набор векторов не является базисом.

В чем же принципиальная разница между этими базисами, и может ли базис двумерного пространства состоять, например, из большего или меньшего числа векторов, чем 2?

У линейной алгебры есть на это ответ: **Любые n линейно независимых векторов n -мерного векторного пространства образуют базис этого пространства**.

Именно из-за линейной зависимости векторов $\begin{pmatrix} -2 \\ 0 \end{pmatrix}, \begin{pmatrix} -3 \\ 0 \end{pmatrix}$ они не могут являться базисом двумерного пространства.

15.7 Что мы узнали?

- Определение вектора
- Операции с векторами
- Норма (длина) вектора
- Скалярное произведение
- Линейная независимость
- Базис

СКОБКИ

Дисклеймер: нам нужен инструментарий для работы с более абстрактными пространствами!

Символ Кронекера и остальные специальные тензоры – это удобные примитивы для низкоуровневой работы с матрицами и векторами. Физики смогли придумать поверх примитивов линейной алгебры отличный синтаксический сахар, т.н. нотацию Дирака. Но давайте начнем с определения гильбертова пространства.

16.1 Гильбертово пространство

Гильбертово пространство – это полное линейное векторное пространство, определяемое:

1. скалярным произведением (u, v) , в простейшем частном случае - $(u, v) = u^\dagger v$;
2. зафиксированной нормой вида $\|v\| = \sqrt{(v, v)}$;
3. метрикой $d(u, v) = \|u - v\| = \sqrt{(u - v, u - v)}$.

16.2 Полное пространство

При нахождении предела некоторой последовательности полное метрическое пространство дает возможность не думать о принадлежности этого предела данному пространству, то есть по своей сути разрешает предельный переход и позволяет доказать множество красивых теорем. Подробнее строгое определение можно посмотреть [тут](#).

16.3 Нотация Дирака, или Bra-Ket нотация

Напомним, что в гильбертовом пространстве для пространства H определено сопряженное пространство H^\dagger . Тогда можно рассмотреть обозначение Ket для элемента v из H как вектор-столбец:

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix}$$

и обозначение Bra для элемента u из сопряженного пространства H^* как сопряженную вектор-строку

$$u = [u_1^* \quad u_2^* \quad \cdots \quad u_N^*]$$

Bra-ket u и v просто задает скалярное произведение между этими элементами:

$$uv = u_1^*v_1 + u_2^*v_2 + \dots + u_N^*v_N = [u_1^* \ u_2^* \ \dots \ u_N^*] \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix}$$

Часто, например, требуется показать два граничных состояния кубита 0, 1, тогда мы можем записать их просто как вектор-столбцы: $0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ и $1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. В следующей лекции этот вопрос будет рассмотрен детальнее.

16.4 Внешнее произведение (outer-product)

Также имеет смысл и переставленная запись Ket-Bra, которая называется *внешним произведением*:

$$uv = [u_1^* \ u_2^* \ \dots \ u_N^*] \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_N \end{bmatrix} = \begin{bmatrix} u_1^*v_1 & u_1^*v_2 & \dots & u_1^*v_N \\ u_2^*v_1 & u_2^*v_2 & \dots & u_2^*v_N \\ \vdots & \dots & \ddots & \vdots \\ u_N^*v_1 & \dots & \dots & u_N^*v_N \end{bmatrix}$$

По сути это матрица $N \times N$, то есть новый оператор в гильбертовом пространстве. Не все перестановки имеют смысл, например, нельзя записать vv или uv .

Мы вернемся к Ket-Bra чуть ниже, когда будем говорить про операторы проекции.

16.5 Эрмитов оператор

Оператор U называется *эрмитовым*, если он удовлетворяет равенству $(Uv, u) = (v, Uu)$ для всех u, v из H или в матричном виде:

$$U = U^\dagger$$

16.6 Унитарный оператор

Унитарный оператор $\hat{U} : H \rightarrow H$ на гильбертовом пространстве H – это линейный оператор, который удовлетворяет следующему равенству $(Uv, Uu) = (v, u)$ для любых $v \in H, u \in H$. Или в матричной форме:

$$\hat{U}^\dagger \hat{U} = \hat{U} \hat{U}^\dagger = I$$

Напомним, что операция \hat{U}^\dagger (другое частое обозначение в работах – звездочка или $H\hat{U}^* = \hat{U}^H$) в матричных терминах является последовательным применением операции транспонирования и последующего комплексного сопряжения элементов этой матрицы $\hat{U}^\dagger = \overline{\hat{U}}^T$ (порядок этих операций, естественно, не влияет на результат).

```
import numpy as np
import scipy as sp
from scipy import linalg

U_hat = np.array([
    [1 + 0j, 0 + 0j],
```

(continues on next page)

(continued from previous page)

```
[0 + 0j, 1 + 0j]
])

U_hat_star_byhands = U_hat.conj().T
U_hat_star_long = np.conjugate(np.transpose(U_hat))

print(np.allclose(U_hat_star_long, U_hat_star_byhands))
```

True

Также в Python для многих операций есть соответствующие методы вместо функций и их сокращения, например `U.transpose()` – то же самое, что `U.T`, а `U.conjugate()` – `U.conj()`.

Важное свойство, что любой эрмитов оператор U можно привести к унитарному оператору с помощью взятия матричной экспоненты от матрицы оператора, умноженного на мнимую единицу:

$$\hat{U} = e^{iU}$$

Давайте докажем быстро этот факт: для эрмитовых квадратных матриц можно определить матричные функции через спектральное разложение матрицы, которое в силу свойств эрмитовых матриц имеет вид: $U = S\Lambda S^\dagger$, где S – унитарная матрица перехода к тому базису, где изначальный оператор выражается диагональной матрицей. Тогда матричная функция $f(U) = Sf(\Lambda)S^\dagger$, т.е. мы диагонализируем матрицу и применяем функцию ко всем диагональным элементам, а потом возвращаемся в исходный базис унитарным преобразованием. Соответственно, для экспоненты:

$$e^V = Se^\Lambda S^\dagger.$$

Note: В качестве упражнения для самопроверки можете показать, что такое определение эквивалентно определению через степенной ряд: $e^U = \sum_{k=0}^{\infty} \frac{1}{k!} U^k$

Доказательство: Проверим определение унитарной матрицы: $\hat{U}^\dagger \hat{U} = (S \exp(i*\Lambda)S^\dagger)^\dagger S \exp(i*\Lambda)S^\dagger$. Пользуясь тем, что эрмитово сопряжение переставляет матрицы и является обратной операцией к себе же, а также тем, что S – унитарная матрица (то есть $S \cdot S^\dagger = I$), получаем:

$$(S \exp(i*\Lambda)S^\dagger)^\dagger S \exp(i*\Lambda)S^\dagger = \exp(-i*\Lambda)S^\dagger S \exp(i*\Lambda)S^\dagger$$

Заметим, что теперь у нас уже скалярная экспонента, которая применяется к элементам диагональной матрицы, и мы можем воспользоваться тем, что произведение экспонент превращается в экспоненту от суммы степеней:

$$S \exp(-i\Lambda + i\Lambda)S^\dagger = S \exp(O)S^\dagger = SIS^\dagger = I$$

В конце мы еще раз воспользовались тем, что S – унитарная. Абсолютно так же доказывается, что $\hat{U}\hat{U}^\dagger = I$.

Note: Кстати, любая матрица вида HH^\dagger является эрмитовой.

Давайте продемонстрируем доказанный факт на примере матрицы дискретного преобразования Фурье без нормировочного коэффициента $\frac{1}{N}$, $N = 3$, преобразованной к DD^\dagger :

```
N = 3
w = np.sqrt(np.exp(-1j * 2 * np.pi / N))
```

(continues on next page)

(continued from previous page)

```
D = np.array([
    [1, 1, 1],
    [1, w, w ** 2],
    [1, w ** 2, w ** 4]
])

U = D @ D.conj().T
print(f"U = {U}")

print(f"D = {D}")
print(f"np.allclose(U @ U.T.conj(), np.eye(N)) = {np.allclose(U @ U.T.conj(), np.eye(N))} # no")

U_hat = linalg.expm(1j * U)
print(f"\nnp.allclose(U_hat @ U_hat.conj().T, np.eye(N)) = {np.allclose(U_hat @ U_hat.conj().T, np.eye(N))}")
print(f"\nnp.allclose(U_hat.conj().T @ U_hat, np.eye(N)) = {np.allclose(U_hat.conj().T @ U_hat, np.eye(N))}")
```

```
U = array([[ 3.00000000e+00+0.00000000e+00j,  1.00000000e+00+1.73205081e+00j,
           -1.11022302e-16+7.77156117e-16j],
           [ 1.00000000e+00-1.73205081e+00j,  3.00000000e+00+0.00000000e+00j,
            1.00000000e+00+1.73205081e+00j],
           [-1.11022302e-16-7.77156117e-16j,  1.00000000e+00-1.73205081e+00j,
            3.00000000e+00+0.00000000e+00j]])
D = array([[ 1. +0.j      ,  1. +0.j      ,  1. +0.j      ],
           [ 1. +0.j      ,  0.5-0.8660254j, -0.5-0.8660254j],
           [ 1. +0.j      , -0.5-0.8660254j, -0.5+0.8660254j]])
np.allclose(U @ U.T.conj(), np.eye(N)) = False

np.allclose(U_hat @ U_hat.conj().T, np.eye(N)) = True
np.allclose(U_hat.conj().T @ U_hat, np.eye(N)) = True
```

16.6.1 Пример: оператор-проектор

Оператором проекции является оператор P со свойством $P^2 = P$.

Покажем, что Ket-Bra вида $\Psi\Psi$ обладает этим свойством.

$$(\Psi\Psi)^2 = \Psi\Psi\Psi\Psi = \Psi(\Psi\Psi)\Psi = \Psi\Psi,$$

поскольку вектор состояния – нормированный: $\Psi\Psi = 1$.

Оператор Ket-Bra с вектором состояния Ψ , то есть $\Psi\Psi$ рассматривается во *вводной лекции про кубиты*, а также пригодится позже, когда речь зайдет о *смешанных состояниях*.

16.6.2 Пример: оператор поворота

Оператором поворота по оси вращения $v = (x, y, z)$ на угол θ является

$$M(v, \theta) = \begin{bmatrix} \cos \theta + (1 - \cos \theta)x^2 & (1 - \cos \theta)xy - (\sin \theta)z & (1 - \cos \theta)xz + (\sin \theta)y \\ (1 - \cos \theta)yx + (\sin \theta)z & \cos \theta + (1 - \cos \theta)y^2 & (1 - \cos \theta)yz - (\sin \theta)x \\ (1 - \cos \theta)zx - (\sin \theta)y & (1 - \cos \theta)zy + (\sin \theta)x & \cos \theta + (1 - \cos \theta)z^2 \end{bmatrix}$$

Например, матрица поворота относительно оси x на 90° : $x = 1, y = 0, z = 0, \theta = \frac{\pi}{2}$, будет иметь вид:

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \frac{\pi}{2} & -\sin \frac{\pi}{2} \\ 0 & \sin \frac{\pi}{2} & 1 \end{bmatrix}$$

Если у нас больше измерений, то по тем измерениям, которые не затрагиваются этим поворотом, у нас в строке и столбце стоят нули, кроме пересечения строки и столбца, отвечающих за это измерение - там стоит единица.

Операторы поворота очень важны в квантовых вычислениях. Они рассматриваются отдельно в [лекции про квантовые гейты](#) и далее используются в [вариационных квантовых схемах](#) для кодирования классических данных в квантовые операторы.

16.6.3 Пример: оператор дифференцирования

В пространстве многочленов P с базисом $\{1, t, t^2, \dots, t^n\}$ можно задать оператор дифференцирования $\mathcal{D}: P_n \rightarrow P_{n-1}$ в виде матрицы:

$$D = \begin{bmatrix} 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 2 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 3 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & 0 & \cdots & n \end{bmatrix}$$

Тогда производная многочлена $p = a_0 + a_1t + \dots + a_nt^n = [a_0 \ a_1 \ \dots \ a_n] \cdot \begin{bmatrix} 1 \\ t \\ \vdots \\ t^n \end{bmatrix}$

$$\mathcal{D}(p) = [a_0 \ a_1 \ \dots \ a_n] D \begin{bmatrix} 1 \\ t \\ \vdots \\ t^{n-1} \end{bmatrix} = a_1 + 2a_2t + \dots + na_nt^{n-1}.$$

Про разницу между оператором и матрицей преобразования

Мы с вами рассматривали операторы через конечные матрицы, но на самом деле гильбертово пространство было придумано как раз, чтобы можно было работать с бесконечномерными векторами и применять непрерывные операторы. Можно считать, что есть некоторый дуализм между непрерывным оператором и пределом бесконечной матрицы. Иногда удобнее работать с матрицей, а иногда – с абстрактным оператором.

16.7 Произведение Кронекера

Давайте рассмотрим еще одну интересную операцию, которая называется матричным тензорным произведением (является тензорным произведением для линейных операторов) или произведением Кронекера.

Проще всего его необходимость можно продемонстрировать на примере двух игр: Орел/Решка и бросок кубика. Мы можем записать состояния этих игр через вероятности событий и давайте возьмем монетку со смешенным центром тяжести и такой же кубик:

$$\text{coin} = \begin{bmatrix} \frac{1}{3} & \frac{2}{3} \end{bmatrix} \text{ для нашей монетки и dice} = \begin{bmatrix} \frac{1}{4791} & \frac{1}{5} & \frac{1}{7} & \frac{1}{11} & \frac{1}{13} \end{bmatrix}_{2020} \text{ для нашей игральной кости.}$$

Тогда если мы захотим сыграть в игру, когда сначала подкидывается монетка, а потом – игральный кубик, нам

будет удобно записать это в виде либо очень длинного вектора:

$$\text{game}_{\text{vec}} = \begin{bmatrix} \frac{1}{3} \times \frac{1}{4} \\ \frac{1}{3} \times \frac{4791}{20020} \\ \frac{2}{3} \times \frac{1}{13} \\ \frac{1}{3} \times \frac{1}{5} \\ \frac{2}{3} \times \frac{4791}{20020} \\ \frac{2}{3} \times \frac{1}{7} \\ \frac{1}{3} \times \frac{1}{11} \\ \frac{2}{3} \times \frac{1}{11} \\ \frac{1}{3} \times \frac{1}{13} \end{bmatrix}$$

Либо в виде матрицы, где по строкам будут события монетки, а по столбцам – кубика:

$$\text{game}_{\text{matrix}} = \begin{bmatrix} \frac{1}{3} \times \frac{1}{4} & \frac{1}{3} \times \frac{1}{5} & \frac{1}{3} \times \frac{1}{7} & \frac{1}{3} \times \frac{1}{11} & \frac{1}{3} \times \frac{1}{13} \\ \frac{1}{3} \times \frac{4791}{20020} & \frac{2}{3} \times \frac{1}{5} & \frac{2}{3} \times \frac{1}{7} & \frac{2}{3} \times \frac{1}{11} & \frac{2}{3} \times \frac{1}{13} \\ \frac{2}{3} \times \frac{4791}{20020} & \frac{2}{3} \times \frac{1}{7} & \frac{2}{3} \times \frac{1}{11} & \frac{2}{3} \times \frac{1}{13} & \frac{2}{3} \times \frac{1}{13} \end{bmatrix}$$

С помощью произведения Кронекера (или, повторимся, – матричного тензорного произведения) похожие огромные вектора и матрицы можно очень компактно записать:

$$\begin{aligned} \text{game}_{\text{vec}} &= \text{coin} \otimes \text{dice} \\ \text{game}_{\text{matrix}} &= \text{coin}^T \otimes \text{dice} \end{aligned}$$

В общем случае,

$$A \otimes B = \begin{bmatrix} a_{11}B & \dots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \dots & a_{mn}B \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \dots & a_{11}b_{1q} & \dots & \dots & a_{1n}b_{11} & a_{1n}b_{12} & \dots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \dots & a_{11}b_{2q} & \dots & \dots & a_{1n}b_{21} & a_{1n}b_{22} & \dots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & a_{11}b_{p2} & \dots & a_{11}b_{pq} & \dots & \dots & a_{1n}b_{p1} & a_{1n}b_{p2} & \dots & a_{1n}b_{pq} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \dots & a_{m1}b_{1q} & \dots & \dots & a_{mn}b_{11} & a_{mn}b_{12} & \dots & a_{mn}b_{1q} \\ a_{m1}b_{21} & a_{m1}b_{22} & \dots & a_{m1}b_{2q} & \dots & \dots & a_{mn}b_{21} & a_{mn}b_{22} & \dots & a_{mn}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \dots & a_{m1}b_{pq} & \dots & \dots & a_{mn}b_{p1} & a_{mn}b_{p2} & \dots & a_{mn}b_{pq} \end{bmatrix}$$

Основные его свойства вы можете прочитать в статье: [Произведение Кронекера](#)

Есть и другие нужные тензорные операции, например, чуть больший список вы можете найти в [этой статье](#) или в рекомендованной литературе по квантовой механике.

16.8 Рекомендованная литература

- Quantum Mechanics: The Theoretical Minimum Illustrated Edition by Leonard Susskind, Art Friedman [[SF14](#)] или его русский перевод [[15b](#)]. Книга сочетает в себе довольно строгий формализм вместе с интуицией квантовой механики. Главы построены в виде конкретных примеров или задач, которые понимают, зачем та или иная теория необходима.
- Mathematics for machine learning by Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong [[DFO20](#)] - отличная книжка чтобы освежить воспоминания о базовых алгоритмах в машинном обучении и о математике, которая применяется в них. [Доступна бесплатная электронная версия](#).
- Deep Learning by Ian Goodfellow, Yoshua Bengio, Aaron Courville [[GBC16](#)] - отличная книжка, чтобы разобраться в основах глубокого обучения, чем-то может заменить предыдущую книгу, отличный старт, если вы хотите разобраться в автоэнкодерах или в других нейросетевых моделях. [Доступна бесплатная электронная версия](#).

16.9 Что мы узнали

- произвол со скобками
- Гильбертовы пространства
- Эрмитовый оператор
- Унитарный оператор
- Примеры различных операторов

МАТРИЦЫ

17.1 Определение

Матрица – это математический объект, представляющий собой прямоугольную таблицу чисел.

У каждой матрицы есть размер, который характеризуется двумя числами – количеством строк и количеством столбцов. Матрицу, состоящую из n строк m столбцов будем называть матрицей размера $n \times m$. Пространство вещественных матриц $n \times m$ обозначается $\mathbf{R}^{n \times m}$. Мы будем рассматривать матрицу либо как набор n вектор-строк, либо как набор m вектор-столбцов.

Замечание: Вектор-строку из n элементов можно рассматривать как матрицу размера $1 \times n$, а аналогичного размера вектор-столбец – как матрицу размера $n \times 1$.

Рассмотрим произвольную матрицу \mathbf{A} :

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix}$$

Элемент матрицы \mathbf{A} , стоящий на пересечение строки i и столбца j будем обозначать a_{ij} .

В машинном обучении часто приходится иметь дело с матрицей объекты-признаки. Объекты, как и признаки – это набор числовых характеристик (точки в многомерном пространстве), которые обычно рассматриваются как векторы.

17.2 Операции с матрицами

Аналогично векторным операциям, для матриц определены операции **сложения и умножения на число**:

- сложение:

$$\mathbf{A} + \mathbf{B} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{pmatrix} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \dots & a_{1m} + b_{1m} \\ a_{21} + b_{21} & a_{22} + b_{22} & \dots & a_{2m} + b_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} + b_{n1} & a_{n2} + b_{n2} & \dots & a_{nm} + b_{nm} \end{pmatrix}$$

- умножения матрицы на число (скаляр):

$$\lambda \cdot \mathbf{A} = \lambda \cdot \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} = \begin{pmatrix} \lambda \cdot a_{11} & \lambda \cdot a_{12} & \dots & \lambda \cdot a_{1m} \\ \lambda \cdot a_{21} & \lambda \cdot a_{22} & \dots & \lambda \cdot a_{2m} \\ \dots & \dots & \dots & \dots \\ \lambda \cdot a_{n1} & \lambda \cdot a_{n2} & \dots & \lambda \cdot a_{nm} \end{pmatrix}$$

- транспонирование (строки и столбцы меняются местами):

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} \rightarrow \mathbf{A}^T = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \dots & \dots & \dots & \dots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{pmatrix}$$

Но наибольший интерес представляет операция умножения матриц, которая определяется не самым интуитивным способом.

Рассмотрим операцию скалярного произведения как умножение матрицы размера $1 \times n$ (вектор-строка) на матрицу размера $n \times 1$ (вектор-столбец). Результатом данной операции получается матрица размера 1×1 (скаляр).

Таким образом для получения значения c_{11} результирующей матрицы \mathbf{C} мы скалярно умножили 1-ю строку \mathbf{a}_1 матрицы \mathbf{A} (вектор) на 1-й столбец $\mathbf{b}_{:1}$ матрицы \mathbf{B} (вектор).

Из примере выше мы можем прийти к следующим заключениям:

- для умножения матриц необходимо, чтобы количество столбцов первой матрицы было равно количеству строк второй, иначе нельзя посчитать скалярное произведение
- операция умножения матриц не является коммутативной, то есть $\mathbf{AB} \neq \mathbf{BA}$

Развивая логику примера выше мы можем перейти к определению операции умножения матриц. Рассмотрим матрицы $\mathbf{A} \in \mathbb{R}^{n \times p}$ и $\mathbf{B} \in \mathbb{R}^{p \times m}$. Результатом произведения этих матриц будем матрица $\mathbf{AB} = \mathbf{C} \in \mathbb{R}^{n \times m}$, где значение c_{ij} получится путем скалярного произведения i -й строки \mathbf{a}_i матрицы \mathbf{A} (вектор) на j -й столбец $\mathbf{b}_{:j}$ матрицы \mathbf{B} (вектор):

$$c_{ij} = \mathbf{a}_i \cdot \mathbf{b}_{:j} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{ip}b_{pj}$$

Замечание 1: В результате умножения квадратных матриц размера $n \times n$ получается квадратная матрица того же размера.

Замечание 2: Частными случаями матричного умножения являются операции умножения вектор-строки на матрицу и матрицы на вектор-столбец, в результате которых получаются вектор-строка и вектор-столбец соответственно.

Рассмотрим пример, почему удобно представлять операцию умножения матриц именно таким образом. Пусть имеется система линейных уравнений:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1m}x_m = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2m}x_m = b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nm}x_m = b_n \end{cases}$$

Представим данную систему в матричном виде:

$$\mathbf{Ax} = \mathbf{b} \rightarrow \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_m \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{pmatrix}$$

17.3 Норма матрицы

Понятие матричной нормы несколько сложнее, чем векторной. Обычно принято рассматривать так называемые **операторные нормы**. Такая норма показывает насколько максимально растягивается произвольный вектор \mathbf{x} при отображении $y = \mathbf{A}\mathbf{x}$. Формально операторная норма определяется следующим образом:

$$\|\mathbf{A}\|_p = \sup_{\mathbf{x} \neq 0} \frac{\|\mathbf{A}\mathbf{x}\|_p}{\|\mathbf{x}\|_p}$$

Существует множество матричных норм, но наиболее полезной является **спектральная норма**. Опуская математические выкладки, можно показать, что спектральная норма равна максимальному собственному числу матрицы $\mathbf{A}^T \mathbf{A}$:

$$\|\mathbf{A}\|_2 = \sqrt{\lambda_{max}(\mathbf{A}^T \mathbf{A})}$$

Про собственные числа поговорим позже, а пока вернемся к нормам. Иногда также рассматривают **поэлементные нормы** матриц. Такая норма определяется тем же образом, что и для векторов. Наиболее распространенной из данного вида является **Евклидова норма**, она же **норма Фробениуса**:

$$\|\mathbf{A}\|_F = \sqrt{\sum_i \sum_j a_{ij}^2}$$

17.4 Ранг матрицы

Вспомним, что матрицу можно рассматривать либо как набор вектор-строк, либо как набор вектор-столбцов, а для набора векторов определено понятие линейной независимости.

Таким образом мы приходим двум следующим понятиям:

- **строчный ранг** матрицы – это максимальное число линейно независимых строк (вектор-строк)
- **столбцовый ранг** матрицы – это максимальное число линейно независимых столбцов (вектор-столбцов)

Тут мы можем задаться вопросом о том, равны ли эти ранги между собой, и если равны, то при каких условиях, и на помощь нам приходит **фундаментальная теорема «о ранге матрицы»**, которая говорит, что ранг, посчитанный по строкам всегда равен рангу, посчитанному по столбцам. Поэтому обычно говорят просто о ранге матрицы.

17.5 Линейные преобразования

В рамках школьной программы все знакомы с понятием функции. Функция – это соответствием между элементами двух множеств.

Например, линейная функция вида $y = f(x) = kx + b$ отображает элементы множества вещественных чисел $x \in \mathbf{R}$ в себя $y \in \mathbf{R}$:

$$f : \mathbf{R} \rightarrow \mathbf{R}$$

Парабола $y = f(x) = x^2$ отображает элементы множества вещественных чисел $x \in \mathbf{R}$ во множество неотрицательных вещественных чисел $y \in \mathbf{R}^+ \cup \{0\}$:

$$f : \mathbf{R} \rightarrow \mathbf{R}^+ \cup \{0\}$$

Среди всего множества функций есть определенный класс функций, называемых линейными. Этот класс является самым простым среди множества всех функций. Основным свойством линейных функций является пропорциональность x и y : изменение значения функции y пропорционально изменению значения аргумента x . Графиком линейной функции является прямая.

Обобщая идею линейных функций на матрицы и вектора мы можем прийти к понятиям **линейных отображений и линейных преобразований**.

Линейное отображение – это обобщение линейной функции, которое принимает на вход вектор и возвращает вектор. Вектор может быть как той же размерности, так и другой.

Формально, линейное отображение – это отображение множества элементов одного векторного пространства V в другое W

$$f : V \rightarrow W,$$

удовлетворяющее так называемым **условиям линейности**.

Если V и W — это одно и то же векторное пространство, то f называют не просто линейным отображением, а **линейным преобразованием**.

Отображения и преобразования удобно рассматривать как движение в векторном пространстве.

Ранее упоминалось, что при умножении вектора на матрицу мы получаем вектор, поэтому матрица задает преобразование, которое, если проверить условия линейности, является линейным. Соответственно, **матрица задает линейное отображение**.

Например, матрица $A \in \mathbb{R}^{n \times m}$ задает преобразование из векторного пространства размерности n в векторное пространство размерности m .

Замечание 1: Для того, чтобы матрица задавала линейное преобразование, она должна быть квадратной, т.к. в результате умножение вектора на квадратную матрицу его размерность не меняется. Таким образом вектор, как элемент векторного пространства, отображается в то же векторное пространство.

Замечание 2: Пусть имеются две матрицы $A \in \mathbb{R}^{n \times p}$ и $B \in \mathbb{R}^{p \times m}$. Умножив матрицу A на матрицу B мы получим матрицу $C \in \mathbb{R}^{n \times m}$. Матрица C задает линейное отображение, равное последовательному применению отображений, задаваемых матрицами A и B . Таким образом операция умножения матриц имеет простую интерпретацию - последовательное применение линейных отображений.

Рассмотрим некоторые примеры линейных преобразований на плоскости, задаваемых матрицами размера 2×2 :

- Вращение / поворот:
 - на 90 градусов против часовой стрелки: $A = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$
 - на угол θ против часовой стрелки: $A = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}$
- Отражение:
 - относительно оси x : $A = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
 - относительно оси y : $A = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$
 - относительно линии, образующей угол θ с началом координат: $A = \begin{pmatrix} \cos 2\theta & \sin 2\theta \\ \sin 2\theta & -\cos 2\theta \end{pmatrix}$
- Масштабирование:
 - в 2 раза по всем направлениям: $A = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$

- Проекция:

– на ось y : $\mathbf{A} = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$

17.6 Обратная матрица

Рассмотрим особую матрицу линейного преобразования, которая переводит каждый вектор сам в себя, т.е. никак его не изменяет. Данное преобразование задается так называемой **единичной матрицей**:

$$\mathbf{E} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Произведение любой матрицы / вектора и единичной матрицы подходящего размера равно самой матрице / вектору (аналог единицы для операции умножения чисел):

$$\mathbf{AE} = \mathbf{EA} = \mathbf{A}$$

Появление такого элемента в пространстве вещественных матриц наводит на мысль о том, что для каждой матрицы должен существовать обратный элемент:

$$\mathbf{AA}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{E}$$

Замечание: Аналогично операции умножения чисел, если $a \cdot 1 = 1 \cdot a = a$, то существует a^{-1} , такой, что $a \cdot a^{-1} = a^{-1} \cdot a = 1$.

Теория линейной алгебры говорит о том, что обратная матрица существует только для **невырожденных квадратных матриц**, строки и столбцы которых линейно независимы.

Те линейные преобразования, которые были приведены в примерах выше задаются невырожденными квадратными матрицами, поэтому для них существуют обратные преобразования, которые тоже будут линейными.

Например, обратными к преобразованиям вращения / поворота, отражения и масштабирования являются эти же преобразования: вращение / поворот, отражение и масштабирование соответственно.

Замечание: Существуют разные алгоритмы нахождения обратной матрицы, которые не будут нами рассматриваться, но с ними можно ознакомиться [здесь](#).

Будем называть линейное преобразование **вырожденным**, если оно задается вырожденной матрицей.

Соответственно, возникает вопрос, чем же так особенны вырожденные линейные преобразования, если для них не существует обратного преобразования?

Невырожденное линейное преобразование устанавливает взаимно-однозначное соответствие между входными и выходными векторами. Вырожденное преобразование этим свойством не обладает.

Это связано с тем, вырожденные линейные преобразования переводят некоторые ненулевые вектора в нулевые (для невырожденных линейных преобразований только нулевой вектор переходит в нулевой):

$$\mathbf{Ax} = \mathbf{0}, \mathbf{x} \neq \begin{pmatrix} 0 \\ 0 \\ \dots \\ 0 \end{pmatrix}$$

В связи с этим одному выходному вектору (нулевому) соответствует несколько входных векторов, поэтому не существует обратного преобразования.

17.7 Определитель

Введем еще одно из ключевых понятий линейной алгебры – **определитель**.

Определитель – это скалярный показатель (число) **квадратной матрицы** $f : \mathbf{R}^{n \times n} \rightarrow \mathbf{R}$, который позволяет охарактеризовать некоторые свойства этой матрицы и линейного преобразования, заданного этой матрицей.

Определитель матрицы \mathbf{x} обозначается как: $\det(\mathbf{A}) = |\mathbf{A}| = \Delta(\mathbf{A})$

Формально понятие определителя может быть введено несколькими способами:

- через перестановки
- через свойства определителя
- геометрически

Замечание: В данной лекции не будет рассказано, как посчитать определитель. Ознакомиться с алгоритмом расчета и формальным выводом определителя можно по [ссылке](#).

Определитель обладает следующими важными свойствами – он **равен нулю** тогда и только тогда, когда **строки (столбцы) матрицы линейно зависимы**.

Из этого можно сделать следующие выводы:

- матрица вырожденная тогда и только тогда, когда ее определитель равен 0
- линейное преобразование вырожденное тогда и только тогда, когда определитель матрицы, задающей данное преобразование, равен 0
- если определитель матрицы равен 0, то для данной матрицы не существует обратной

Существуют эффективные алгоритмы вычисления определителя квадратной матрицы, которые позволяют проверять, является матрица вырожденной.

17.8 Собственные числа и векторы

Для некоторых линейных преобразований существуют такие векторы, которые после данного преобразования либо растягиваются, либо сжимаются (тот же вектор, умноженный на некоторое число). Такие векторы называются **собственными векторами** линейного преобразования, а числа, на которые данные векторы умножаются – **собственными значениями**.

Рассмотрим произвольную матрицу линейного преобразования \mathbf{A} . Формально понятия собственных векторов и собственных значений вводятся через следующее равенство: $\mathbf{Ax} = \lambda \mathbf{x}$

Векторы \mathbf{x} , удовлетворяющие данному равенству являются собственными векторами линейного преобразования \mathbf{A} , а соответствующие данным векторам значения λ – собственными числами.

Линейное преобразование может как не иметь собственных векторов (например, поворот в двумерном пространстве), или иметь n собственных векторов с различными собственными значениями.

Собственные числа и собственные значения играют большую роль в линейной алгебре, так как многие соотношения, связанные с линейными преобразованиями, существенно упрощаются в системе координат, построенной на базисе из собственных векторов линейного преобразованиями.

Множество собственных значений линейного преобразованиями (**спектр оператора**) характеризует важные свойства оператора без привязки к какой-либо конкретной системе координат.

Также собственные векторы используются в Методе Главных Компонент, который предназначен для уменьшения размерности данных с потерей наименьшего количества информации.

17.9 Что мы узнали?

- Определение матрицы
- Операции с матрицами
- Норма матрицы
- Ранг матрицы
- Линейные преобразования
- Обратная матрица
- Определитель матрицы
- Собственные числа и векторы матрицы

CHAPTER
EIGHTEEN

NUMPY

Numpy – это широко используемая библиотека для вычислений с многомерными массивами. API большей частью вдохновлен MATLAB (великая и ужасная среда, язык и IDE для матричных вычислений), а теперь сам является примером для подражания API различных вычислительных пакетов. Более последовательный гайд стоит посмотреть на [сайте библиотеки](#).

18.1 Массивы

```
import numpy as np

a = np.array([1, 2, 3]) # создадим вектор
print(f"{a = }")

b = np.zeros((2, 2))
print(f"{b = }")

c = np.eye(3)
print(f"{c = }")

q = np.random.random((1, 20))
print(f"{q = }")
```

```
a = array([1, 2, 3])
b = array([[0., 0.],
           [0., 0.]])
c = array([[1., 0., 0.],
           [0., 1., 0.],
           [0., 0., 1.]])
q = array([[0.34979354, 0.32469037, 0.07987546, 0.76317591, 0.28396415,
           0.29350859, 0.06632662, 0.15404105, 0.60851127, 0.66636035,
           0.5680136 , 0.23226561, 0.28062998, 0.03541475, 0.26637316,
           0.31127287, 0.6340288 , 0.57927801, 0.70243011, 0.76263531]])
```

18.2 Арифметические операции

Для удобства использования np.ndarray арифметические операторы определены так, чтобы соответствовать ожиданиям:

```
a = np.array([1, 2, 3])
b = np.array([-1, 3, 4])

diff = a - b
print(f"{diff = }")

mult = a * b
print(f"{mult = }")

scalar_mult = a @ b
print(f"{scalar_mult = }")
```

```
diff = array([ 2, -1, -1])
mult = array([-1, 6, 12])
scalar_mult = 17
```

18.3 Indexing, slicing and sugar

Numpy поддерживает, кажется, все разумные варианты индексации:

```
a = np.arange(16).reshape(4, 4)
print(f"{a = }")

# просто по индексам
print(f"\n{a[0, 1] = }")
print(f"{a[0][1] = }")

# по слайсам
print(f"\n{a[0, 1:3] = }")
print(f"{a[2] = }")
print(f"{a[2, :] = }")
print(f"{a[2, ...] = }")

# по маске
mask = (a % 3 == 0)
print(f"\n{mask = }")
print(f"{a[mask] = }")

first_rows = np.array([True, True, False, False])
print(f"\n{a[first_rows] = }")
```

```
a = array([[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11],
           [12, 13, 14, 15]])

a[0, 1] = 1
a[0][1] = 1
```

(continues on next page)

(continued from previous page)

```
a[0, 1:3] = array([1, 2])
a[2] = array([ 8,  9, 10, 11])
a[2, :] = array([ 8,  9, 10, 11])
a[2, ...] = array([ 8,  9, 10, 11])

mask = array([[ True, False, False,  True],
              [False, False,  True, False],
              [False,  True, False, False],
              [ True, False, False,  True]])
a[mask] = array([ 0,  3,  6,  9, 12, 15])

a[first_rows] = array([[0, 1, 2, 3],
                       [4, 5, 6, 7]])
```

Для работы с размерностями часто используются еще три конструкции: `None`, `...` (ellipsis, многоточие) и `:` (двоеточие).

```
a = np.arange(16).reshape(4, 4)
print(f" {a = }")

# None добавляет ось размерности 1
print(f"\n{a[None].shape = }")
print(f"{a[:, :, None].shape = }")

# : превращается в slice (None), берет все элементы вдоль размерности
print(f"\n{a[2, :] = }")
print(f"{a[2, 0:None] = }")

# ... ellipsis, превращается в необходимое число двоеточий ::, ::, :
print(f"\n{a[...] = }")

# также ... удобен когда мы не знаем настоящий шейп массива или нужно не трогать
# несколько подряд идущих размерностей
z = np.arange(27).reshape(3, 3, 3)
print(f"\n{z[0, ..., 1] = }")
print(f"{z[0, :, 1] = }")
```

```
a = array([[ 0,  1,  2,  3],
           [ 4,  5,  6,  7],
           [ 8,  9, 10, 11],
           [12, 13, 14, 15]])

a[None].shape = (1, 4, 4)
a[:, :, None].shape = (4, 4, 1)

a[2, :] = array([ 8,  9, 10, 11])
a[2, 0:None] = array([ 8,  9, 10, 11])

a[...] = array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])

z[0, ..., 1] = array([1, 4, 7])
z[0, :, 1] = array([1, 4, 7])
```

В целом в NumPy очень здорово реализованы методы `__getitem__`/`__setitem__`.

```
a = np.array([1, 2, 3])
element = a[2]
print(f"element = {element}")

a[2] = 5
print(f"element = {a[2]}")
```

```
element = 3
a = array([1, 2, 5])
```

Кроме того, мы можем делать индексацию по заданному условию с помощью `np.where`.

```
# создадим вектор
a = np.array([2, 4, 6, 8])

selection = np.where(a < 5)
print(f"selection = {selection}")

# дополнительно можем передать два значения или вектора, при выполнении условия
# выбираются элементы из первого значения/вектора, при невыполнении -- из второго
a2 = np.where(a < 5, 2, a * 2)
print(f"a2 = {a2}")

# np.where работает и с многомерными массивами
b = np.array([[8, 8, 2, 6], [0, 5, 3, 4]])
b_mult = np.where(b < 4, b, 1)
print(f"b_mult = {b_mult}")
```

```
selection = (array([0, 1]),)
a2 = array([ 2,  2, 12, 16])
b_mult = array([[1, 1, 2, 1],
               [0, 1, 3, 1]])
```

18.4 Broadcasting

Что происходит, если мы хотим производить арифметические операции с массивами разных размеров?

```
a = np.array([1, 2, 3])
k = 2
broad = a * k
print(f"broad = {broad}")
```

```
broad = array([2, 4, 6])
```

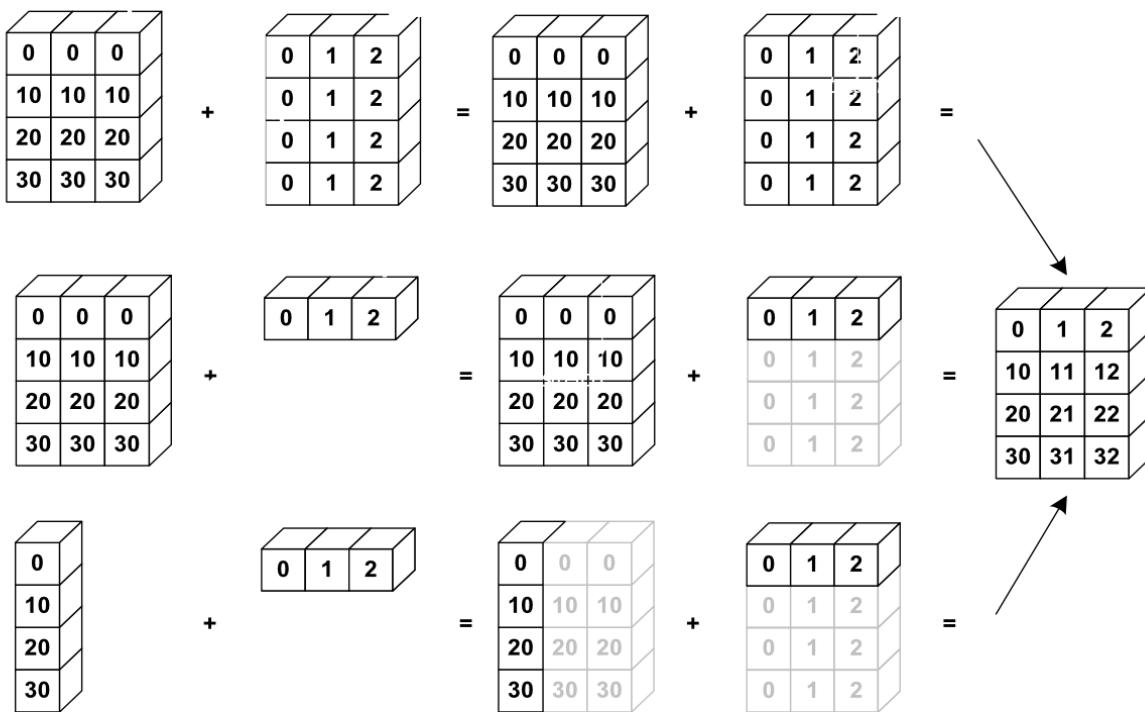
С точки зрения математики, ничего интересного тут не происходит: мы подразумевали умножение всего вектора на скаляр. Однако матричные операции в numpy справляются и с менее очевидными случаями, например, при сложении или вычитании вектора и скаляра:

```
a = np.array([1, 2, 3])
k = 2
broad = a - k
print(broad)
```

[-1	0	1]
-----	---	----

В numpy приняты следующие правила работы с массивами разного размера:

1. размерности сравниваются справа налево;
2. два массива совместимы в размерности, если она одинаковая, либо у одного из массивов единичная;
3. вдоль отсутствующих размерностей происходит расширение повторением (np.repeat).



Attention: Be aware Автоматический броадкастинг легко приводит к ошибкам, так что лучше делать его самостоятельно в явной форме.

18.5 Операции с плавающей точкой

Отдельно стоит поговорить про числа с плавающей точкой. Число с плавающей точкой (или число с плавающей запятой) – экспоненциальная форма представления вещественных (действительных) чисел, в которой число хранится в виде мантиссы и порядка (показателя степени). При этом число с плавающей точкой имеет фиксированную относительную точность и изменяющуюся абсолютную. В результате одно и то же значение может выглядеть по-разному, если хранить его с разной точностью.

```
f16 = np.float16("0.1")
f32 = np.float32(f16)
f64 = np.float64(f32)
print(f" {f16} , {f32} , {f64} ")
print(f" {f16} == {f32} == {f64} ")

f16 = np.float16("0.1")
f32 = np.float32("0.1")
f64 = np.float64("0.1")
print(f" {f16} , {f32} , {f64} ")
print(f" {f16} == {f32} == {f64} ")
```

```
f16 = 0.1, f32 = 0.099975586, f64 = 0.0999755859375
f16 == f32 == f64 = True
f16 = 0.1, f32 = 0.1, f64 = 0.1
f16 == f32 == f64 = False
```

Из-за этого для сравнения массивов с типом float используют `np.allclose`.

```
print(f" {np.allclose([1e10,1e-7], [1.00001e10,1e-8])} ")
print(f" {np.allclose([1e10,1e-8], [1.00001e10,1e-9])} ")
```

```
np.allclose([1e10,1e-7], [1.00001e10,1e-8]) = False
np.allclose([1e10,1e-8], [1.00001e10,1e-9]) = True
```

18.6 NumPy и линейная алгебра

В NumPy много удобных функций, которые позволяют упростить код. Приведем несколько примеров:

```
# матрица с единицами по диагонали и с нулями в остальных ячейках
print(f" {np.eye(2, dtype=int)} ")
# есть возможность указать индекс диагонали
print(f" {np.eye(3, k=-1, dtype=int)} ")

# в NumPy есть свой генератор случайных чисел и векторов
print(f" {np.random.beta(1, 2)} ")
print(f" {np.random.randint(1, 5, (2, 3))} ")

# NumPy позволяет заменить значений основной диагонали матрицы.
# внимание, эта функция работает in-place
a = np.random.randint(1, 5, (3, 3))
print(f" {a} ")
np.fill_diagonal(a, 4)
print(f" {a} ")

# Можно сделать и наоборот -- получить вектор значений диагонали матрица
print(f" {np.diag(a)} ")
```

```
np.eye(2, dtype=int) = array([[1, 0],
                             [0, 1]])
np.eye(3, k=-1, dtype=int) = array([[0, 0, 0],
                                    [1, 0, 0],
                                    [0, 1, 0]])
```

(continues on next page)

(continued from previous page)

```
np.random.beta(1, 2) = 0.15792174610491572
np.random.randint(1, 5, (2, 3)) = array([[4, 3, 4],
                                         [4, 3, 4]])
a = array([[1, 2, 1],
           [2, 2, 4],
           [2, 2, 4]])
a = array([[4, 2, 1],
           [2, 4, 4],
           [2, 2, 4]])
np.diag(a) = array([4, 4, 4])
```

18.6.1 Решение систем линейных уравнений

NumPy позволяет решить систему линейных уравнений.

```
a = np.array([[7, 4], [9, 8]])
b = np.array([5, 3])
solution = np.linalg.solve(a, b)
print(solution)
```

```
[ 1.4 -1.2]
```

18.6.2 Обращение матриц

NumPy дает возможность выполнить операцию обращения матриц.

```
a = np.array([[1., 2.], [3., 4.]])
inv = np.linalg.inv(a)
print(inv)
```

```
[[ -2.   1. ]
 [ 1.5 -0.5]]
```

18.6.3 Собственные вектора и числа

Вычисление собственных векторов и чисел.

```
print(np.linalg.eig(np.diag((1, 2, 3))))
```

```
(array([1., 2., 3.]), array([[1., 0., 0.],
                             [0., 1., 0.],
                             [0., 0., 1.]]))
```

18.7 Мы вкратце рассмотрели

- основы работы с NumPy;
- индексацию в массивах;
- broadcasting массивов NumPy;
- операции с плавающей точкой;
- NumPy и примитивы линейной алгебры.

CHAPTER
NINETEEN

О БЛОКЕ “МАШИННОЕ ОБУЧЕНИЕ”

Этот блок рассказывает о том, что общего у разных задач машинного обучения и как из основных компонентов, подобно паззлу, складываются различные применения моделей машинного обучения. Основная наша задача – чтобы у читателя даже не знакомого с машинным обучением возникло понимание того, как это все работает и как “сложить паззл” в новой задаче, будь то кластеризация новостей, детекция лиц на фотографиях или различные сложные применения вариационных квантовых схем, о которых пойдет речь далее в курсе.

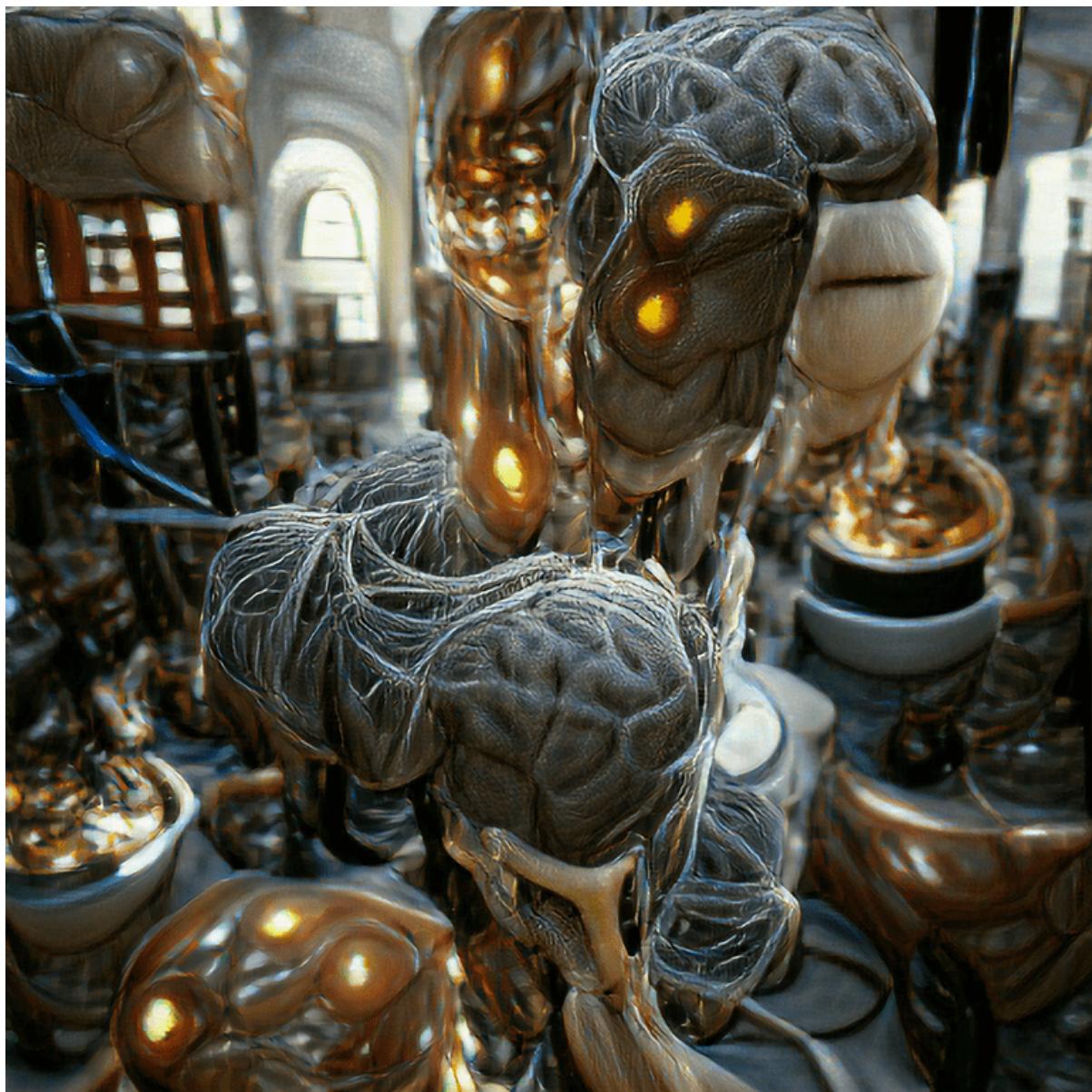


Fig. 19.1: Нейронные сети “глазами” других нейросетей – VQGAN и CLIP, сгенерировано в Google Colab

МАШИННОЕ ОБУЧЕНИЕ КАК ПАЗЛ

20.1 Введение

Мы продолжаем вводную часть нашего курса и переходим к машинному обучению. Если тема для вас хорошо знакома, можете пропустить этот блок, при желании можно пройти [этот тест](#), для определения достаточности уровня знаний (там 5 вопросов по Python, 5 – по машинному обучению и столько же – по линейной алгебре). Тест можно проигнорировать, если вы не знакомы с машинным обучением и для вас выглядят магией такие вещи как автоматическая детекция лиц на фото или определение тональности отзыва на товар.

Про машинное обучение, конечно, уже много всего написано, есть и немало неплохих курсов, сочетающих как теорию, так и практику. Но все же теория в этой области еще не догоняет практику, мы пока не понимаем, “почему оно работает”, а гарантии обобщающей способности алгоритмов (т.е. гарантии того, что модель машинного обучения будет работать на новых данных) в теории даются только для очень простых моделей. Таким образом, работа со сложными моделями остается своего рода искусством с примесью математики, инженерии и просто следования хорошим практикам, выработанным, как правило, в корпорациях или академическом сообществе.

В этой лекции мы примем сторону практики и расскажем про задачи машинного обучения как некоторый пазл (или легко, кому что ближе) – меняя разные кусочки, мы будем получать разные прикладные задачи/сценарии/модели применения машинного обучения. Для иллюстрации такое описание мы сопроводим 3-мя примерами:

- задача рекомендации контента и градиентный бустинг
- автоматическая оценка читаемости научной статьи и BERT
- детекция симптомов COVID-19 на рентгенограммах и YOLO

В этой лекции мы не опишем подробно, что это за модели машинного обучения (градиентный бустинг, BERT, YOLO), но зато покажем, что сценарии их применения в разных задачах (анализ табличных данных, текстов, изображений) похожи.

Note: Здесь мы почти не будем говорить о математике. Изложенный взгляд на машинное обучение как ремесло, вполне вероятно, вызовет критику со стороны специалистов в области статистики, эконометрики и теории машинного/статического обучения. Мы осознаем эти риски и тем не менее рассказываем о машинном обучении именно как о ремесле. Акцент в курсе делается на квантовые вычисления и квантовое машинное обучение, и в этой лекции мы опишем задачи “классического” машинного обучения на том уровне, чтобы просто было понятно, как это переносится на квантово-классические вариационные схемы и прочие алгоритмы, о которых пойдет речь далее в курсе. При этом строгость изложения материала тоже может немного пострадать.

20.2 Составляющие части задачи машинного обучения

Выделим следующие компоненты (“пазлы”), которые просматриваются во многих разных задачах машинного обучения:

- Целевой признак
- Модель
- Данные
- Функция потерь
- Решатель
- Схема валидации и метрика качества

По ходу изложения будем обсуждать упомянутые примеры задач машинного обучения.

20.2.1 Целевой признак

Есть задачи, в которых машинное обучение не нужно, а достаточно экспертных знаний. По закону Ома, известно что напряжение пропорционально силе тока и электрическому сопротивлению, и вряд ли захочется предсказывать напряжение в сети каким-то другим образом, кроме как применением закона Ома. То же самое можно сказать про многие другие физические явления.

Однако, для очень многих явлений вокруг нет хорошего теоретического объяснения или достаточных экспертных знаний. У нас нет “формулы”, которая описала бы, как поставленный лайк к посту в соцсети, возврат кредита, клик по рекомендации товара или локализация заразы в конкретной части легких зависят от прочих факторов. В таких случаях мы можем приблизить такую неизвестную нам формулу с помощью машинного обучения.

В идеале с помощью машинного обучения мы хотели бы предсказывать какое-то событие, явление или процесс так, чтобы от этого была польза: прибыль компании/клиентов, если речь о бизнес-проекте, или новые знания, если это исследовательский проект. При этом напрямую это сделать вряд ли получится, и надо определить целевой признак, который, как мы считаем, будет связан с целевым событием/явлением. Звучит абстрактно, и дать строгое определение таких событий, явлений или процессов вряд ли получится. Поэтому сразу перейдем к примерам.

Note: Перед этим только небольшое замечание, что потребовав наличие целевого признака, мы ограничились рассмотрением задач обучения с учителем (*supervised learning*). Это все еще включает очень большой перечень типов задач машинного обучения, но не все.

Пример 1. Рекомендация новостного контента

Новостному порталу хочется понять, какой контент нравится пользователям и по каким ссылкам они будут кликать. Здесь *событием* будет то, что пользователю нравится рекомендуемый контент.

Понятно, что нет возможности установить строгую зависимость такого события от прочих факторов. Поэтому мы определяем целевой признак: факт клика пользователя по показанной рекомендованной ссылке. Мы верим, что клик по ссылке связан с *событием*: если пользователю нравится рекомендуемый контент, он/она перейдет по ссылке.

Пример 2. Автоматическая оценка читаемости научной статьи

Допустим, научному журналу хочется автоматически оценивать читаемость текста, чтобы знать, какие статьи можно сразу подавать на рецензию, а какие лучше предварительно направить в сервис proofreading, где статья будет вычитана и поправлена носителем языка.

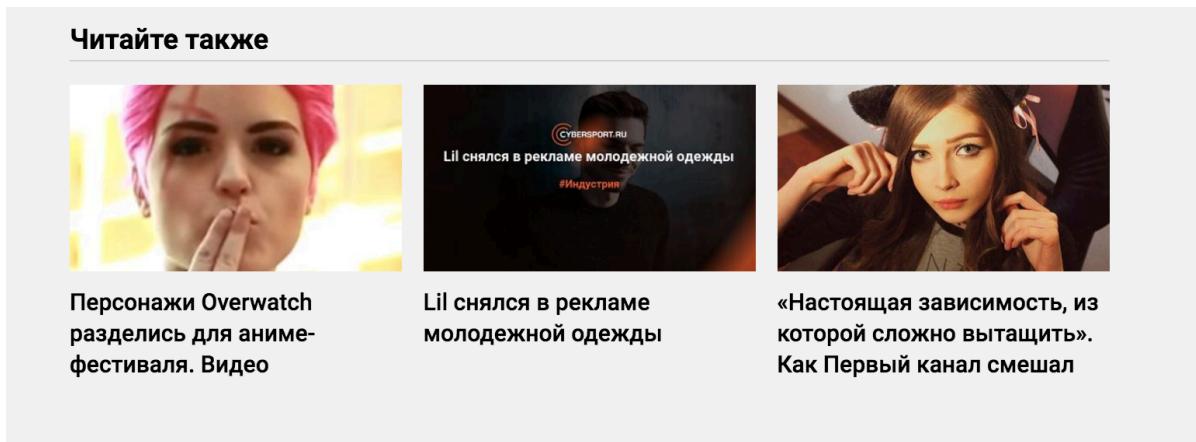


Fig. 20.1: Блок “Читайте также” на новостном портале

В идеале мы бы хотели предсказывать, “хорошо” ли написана статья или “плохо”. Но это очень сложно определить формально, и потому есть много метрик читаемости текста, таких как [Automated readability index](#) или [Flesch reading ease](#), которые являются эвристиками и “приближают” то что мы имеем в виду под “хорошо” или “плохо” написанным текстом. Кстати, на момент написания это лекции на платформе Kaggle проходит соревнование по этой теме.

Небольшое лирическое отступление: во многом опыт специалиста по машинному обучению оказывается в способности понять, когда это машинное обучение **не** требуется. Описанную задачу можно решить и без всякого машинного обучения. Можно замерить 5-10 метрик читаемости текста, разметить 100-200 статей вручную (желательно, чтобы это делали эксперты уровня редактора журнала, а еще лучше иметь по 3 оценки на статью) и заключить, хорошо ли метрики читаемости коррелируют с оценками экспертов. Другой вариант – попытаться малой ценой использовать готовые решения, например, Grammarly.

Но если этого окажется недостаточно, придется подумать. Пока остановимся тут и еще раз подчеркнем, что определить читаемость текста напрямую – невозможно, и мы это заменим на другой *Целевой признак*, например, на агрегированную метрику читаемости текста.

Пример 3. Детекция симптомов COVID-19 на рентгенограммах

Последние пару лет мы видели бурное развитие методов глубокого обучения в приложениях к анализу медицинских данных, а в особенности это стало актуальным в симптомах COVID-эпохи.

Допустим, стоит задача определения аномалий на рентгенограммах грудной клетки. В идеале мы хотели мы сразу по таким изображениям обнаруживать симптомы симптомов COVID-19 у пациента. Но заголовок этого примера выдает желаемое за действительное, и, конечно, сразу по снимкам диагностировать не получится.

Поэтому в такой задаче надо аккуратно определить *Целевой признак*. В данном случае их несколько. Согласно [описанию источника данных](#) соревнования, комитет врачей-радиологов проанализировал 6334 рентгенограмм и пометил их метками: ‘Negative for Pneumonia’ (нет пневмонии), ‘Typical Appearance’ (нормально), ‘Indeterminate Appearance’ (неразборчиво) и ‘Atypical Appearance’ (ненормально). Надо четко понимать, что возможности обученной модели будут ограничены имеющейся разметкой и поэтому заголовок “детекция симптомов COVID-19” слегка “желтоват”, в реальности модель детекции сможет выделять участок изображения (bounding box) и помечать это вектором из 4-х значений, соответствующих описанным целевым признакам в обучающей выборке.

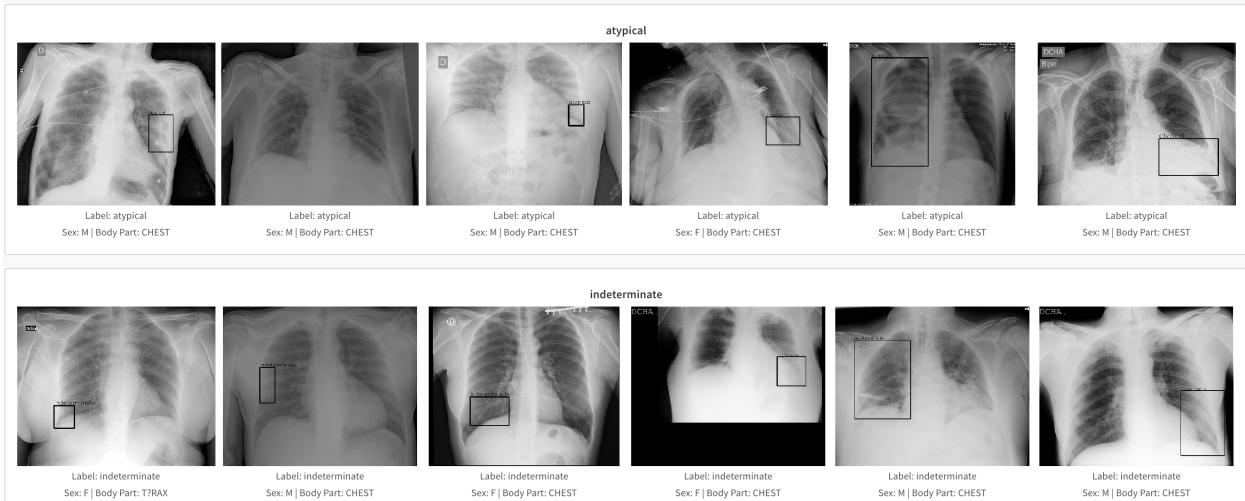


Fig. 20.2: Пример данных соревнования по определению аномалий на рентгенограммах грудной клетки. Источник

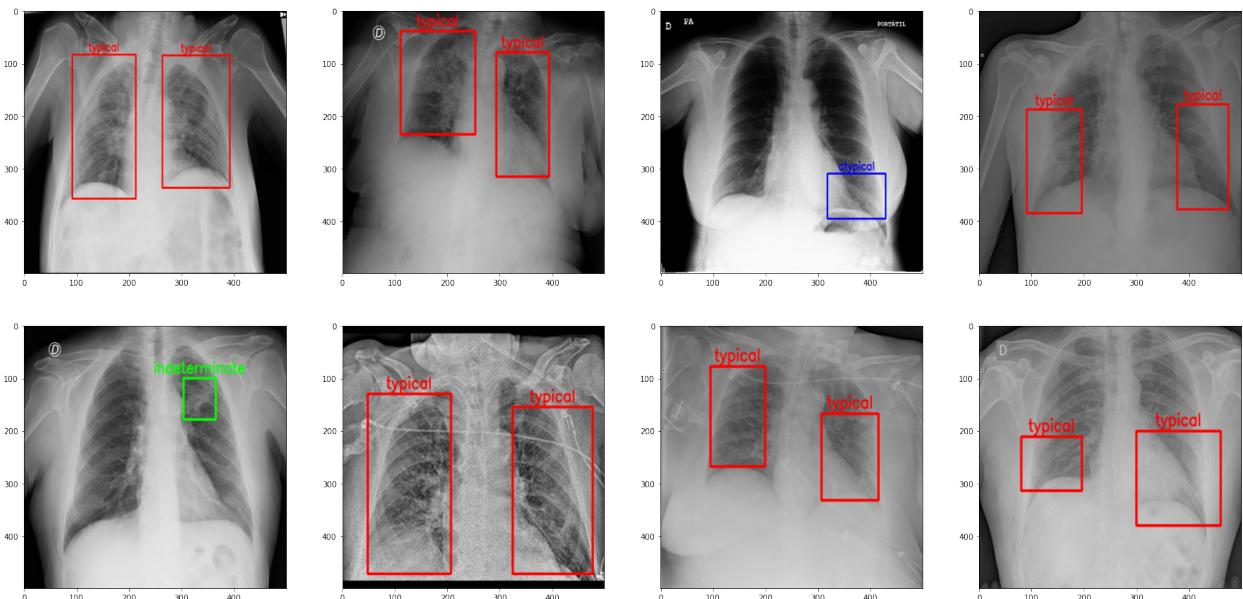


Fig. 20.3: Прогноз модели детекции, обученной на данных соревнования по определению аномалий на рентгенограммах грудной клетки. Источник

20.2.2 Данные

Определение *Целевого признака* неразрывно связано с имеющимися данными. Нет смысла определять целевой признак, который мы не можем измерить или по которому мы не можем собрать данные. Например, если новостной портал не логирует клики пользователей, нет смысла задавать вопрос о том, нравятся ли пользователям показанные рекомендации – для начала надо настроить сервисы для хранения кликов. Другой пример: вряд ли стоит пытаться предсказывать движения денежных активов в микросекундном диапазоне, если нет дорогостоящей инфраструктуры для сбора и обработки таких данных.

Но *Данные* – это, конечно, не только *Целевой признак*. Но и просто *признаки*, также в эконометрике называемые предикторами или независимыми переменными. С помощью *признаков* как раз и получится предсказать *Целевой признак*, а хорошо или плохо – об этом чуть далее.

Продумывание, сбор, и обработка *признаков* – один из наиболее творческих аспектов работы специалистов по машинному обучению. Эта работа может включать в себя общение с представителями предметной области или бизнеса, клиентами, чтение научных статей, применение и более технических приемов, таких как порождение признаков из других признаков (feature engineering) или использование моделей для получения признаков (например, признаки изображения, полученные предобученной искусственной нейросетью).

Пример 1. Рекомендация новостного контента

В этой задаче, как правило, хорошо работают данные о поведении пользователей, попросту – “клики”. Обычно алгоритмы рекомендации хорошо работают при обучении на больших объемах данных, и поэтому большая часть проекта посвящена настройка хранилища данных и обработке потоковых событий (показ рекомендации, клик по рекомендации).

Если есть возможность собрать какие-то признаки пользователей (пол, возраст, указанные интересы) или рекомендуемого контента (темы новостей, представление текста новости “эмбеддингами”), эту информацию можно добавить в модель.

При достаточно хорошей подготовке данных задачу можно решить и без машинного обучения. Довольно сильным прототипом (baseline-решением), опять же, при достаточных объемах данных, может быть просто сортировка контента по Click-through-Rate (CTR). Для каждой новости мы просто собираем статистику, сколько раз она была показана пользователю в качестве рекомендации и сколько раз по ней в итоге кликнули. Отношение кликов к показам и есть Click-through-Rate. Есть, конечно, детали – что делать с “холодными” новостями, без накопленной статистики для расчета CTR, что делать с clickbait-новостями, как фильтровать то, что нельзя показывать в качестве рекомендации. Но после решения этих проблем может оказаться, что просто сортировка по CTR – уже неплохое решение.

Пример 2. Автоматическая оценка читаемости научной статьи

Допустим, научному журналу удалось договориться с сервисом proofreading и получить данные о том, какие статьи хорошо написаны и не требуют множества правок, а какие пришлось переписывать почти что с нуля. Эту информацию можно пытаться использовать для обучения модели, которая для заданного куска текста будет предсказывать, как сильно его нужно поправить.

Тут мы будем иметь дело с задачей из области NLP (Natural Language Processing) – на входе будет просто текст (полный текст статьи или разбитый на параграфы) и по сути мы можем не придумывать *признаки* вручную, а модель сама их извлечет, то есть выучит представление текста. *Целевым признаком* при этом будет, например, какое-либо расстояние (скажем, Левенштейна) между оригинальным текстом и поправленным редактором. Таким образом, это будет задачей регрессии, в которой для текста предсказывается, как сильно его следует изменить.

Опять стоит отметить, что в такой задаче скорее всего нужно много обучающих данных, чтобы описанный алгоритм хорошо заработал.

И еще стоит отметить, что задачи NLP зачастую пересекаются с лингвистикой, и это как раз такой пример. Вполне вероятно, что задачу можно решить без тоже машинного обучения и по-другому – на основе правил-эвристик, разработанных в сотрудничестве с лингвистами. Эдакая версия Grammarly для работы с научными

текстами.

Но дальше в примере будем считать, что лингвистов в команде нет, простые метрики читаемости текста, описанные выше, работают плохо, и мы решаем задачу регрессии, то есть используем машинное обучение.

Пример 3. Детекция симптомов COVID-19 на рентгенограммах

В этой задаче ключевые данные для обучения модели – это собственно изображение, рентгенограмма области грудной клетки и разметка, которая состоит из координат интересующей области изображения и типа области, в данном случае это одна из 4-х меток: ‘Negative for Pneumonia’ (нет пневмонии), ‘Typical Appearance’ (нормально), ‘Indeterminate Appearance’ (неразборчиво) и ‘Atypical Appearance’ (ненормально).

Конечно, у снимков есть разные метаданные, да и форматы медицинских данных обычно специфические, но нам для примера подойдет такое упрощение.

20.2.3 Модель

С моделированием знаком любой исследователь. Чтобы рассчитать минимальную толщину стекла вагона метро для защиты от птиц во время движения по открытым участкам, достаточно представить птицу цилиндром той же массы, и для данной задачи цилиндр будет подходящей моделью птицы.

Подобным же образом, в задачах машинного обучения с учителем *Модель* приближает *Целевой признак* и делает это с помощью *Данных и Параметров*. (Параметры – это неотъемлемая часть модели, и поэтому мы их не выносим как отдельный компонент задач машинного обучения).

Надо понимать, что предлагая модель, мы совершаём уже второе упрощение. Сначала, как мы говорили, *Целевой признак* заменяет нам то, что мы реально хотим знать. А теперь, к тому же, мы заменяем целевой признак на его прогноз с помощью модели.

Пример 1. Рекомендация новостного контента

В задачах рекомендации есть классический алгоритм ALS (Alternative Least Squares), но можно задачу решить и как задачу ранжирования. Это может быть предпочтительно, поскольку можно использовать боевую лошадку машинного обучения на табличных данных – градиентный бустинг (доступно, на русском про эту модель написано [тут](#) на Хабре в рамках курса [mlcourse.ai](#)). Бустинг подходит для задач классификации, регрессии и ранжирования, и его можно использовать также и в описанной задаче.

Также, если бустинг уже используется в компании в других задачах, скорее всего получится безболезненно переиспользовать опыт поддержки модели и соответствующей инфраструктуры в “продакшене” вместо того, чтобы отдельно все это разрабатывать для ALS.

Пример 2. Автоматическая оценка читаемости научной статьи

Описанную задачу, опять же с оговорками про возможность альтернативного подхода без всякого машинного обучения, скорее всего хочется решать с помощью языковых моделей, основанных на трансформерах. В частности, в этой задаче имеет смысл использовать [SciBERT](#), предобученный как раз на научных текстах. Это модель типа BERT (Bidirectional Encoder Representations from Transformers, [arXiv](#)), которая основана на архитектуре трансформеров, ставшей настоящей революцией в NLP (оригинальная статья – [Attention is all you need](#), NIPS 2017). BERT используется для представления текста на основе трансформеров и дообучения параметров под многие стандартные задачи NLP – классификация текстов, пар текстов, вопросно-ответные системы, распознавание именованных сущностей и т.д. Подробнее про BERT можно почитать в [постах Jay Alammar](#) (англ.) или в их переводах на русский: “[Transformer в картинках](#)” и “[Ваш первый BERT: иллюстрированное руководство](#)”.

Пример 3. Детекция симптомов COVID-19 на рентгенограммах

Подходов к детекции объектов на изображениях немало, но по соотношению скорости и качества работы особенно хорошо себя зарекомендовала модель [YOLOv5](#). Про принцип работы этой модели можно узнать из лекции “[Detection and segmentation](#)” курса cs231n, также на Хабре можно найти [статью](#) про YOLOv4.



Fig. 20.4: Пример обнаружения объектов на изображении. Источник

20.2.4 Функция потерь

Выбор функции потерь (loss function) зависит от конкретной задачи, и это вопрос, изучаемый в курсах машинного обучения. Функция потерь определена для объектов обучающей выборки и по сути говорит, насколько прогноз хорошо соответствует значению целевого признака.

Тут тонкий момент: примерно для того же нужны метрики качества, о которых речь пойдет ниже. Но функция потерь на практике чаще всего используется именно для того, чтобы задать цель обучения модели (для чего именно ей менять свои параметры) и также оценить, насколько хорошо модель обучилась, попросту, насколько хорошо она “сошлась”.

В отличие от метрик качества, функции потерь вполне могут быть плохо интерпретируемыми, например как логистическая функция потерь (logloss, на русском про нее можно прочитать в [посте](#) А.Г. Дьяконова), и на практике при разработке модели Data Scientist посмотрит на значение функции потерь всего несколько раз:

- при отладке модели стоит проверить, может ли она “переобучиться под мини-батч”, то есть может ли она при обучении всего с парой десятком примеров добиться почти нулевого значения функции потерь. Это важно, чтобы понять, нет ли где-то ошибки в коде описания модели и хватает ли модели сложности (capacity), чтобы подстроиться под данные
- чтобы избежать переобучения, стоит проверять (вручную или автоматически) значение функции потерь на отложенной выборке
- еще значения функции потерь можно сравнивать для разных версий модели, чтобы понять, какая из них лучше обучилась

Заметим, что дизайн функции потерь под задачу, как и придумывание признаков – порой интересный творческий процесс, а итоговая функция потерь, используемая для обучения модели может быть сложной, состоящей из нескольких более простых функций потерь.

Для примера, в задаче переноса стиля (style transfer), в классическом варианте задаются два изображения – “контентное” и “стилевое” – и генерируется третье изображение, которое похоже в целом на “контентное” изображение, но по стилю – на “стилевое”. При этом функция потерь складывается из двух других:

- одна – content loss – передает, насколько отличаются карты признаков (feature maps) генерируемого и “контентного” изображений
- вторая – style loss – соответственно передает, насколько похожи стили генерируемого и “стилевого” изображений. Делается это хитро, и за деталями лучше обратиться, например, к [лекции “Visualizing and Understanding”](#) стэнфордского курса cs231n.



Fig. 20.5: Пример решения задачи Neural Style Transfer из [задания 3](#) стэнфордского курса cs231n.

Функция потерь может включать и много составляющих, больше двух, если мы хотим чтобы модель выучила разные аспекты задачи. К примеру, в [этой статье на Хабре](#) (уровень - продвинутый NLP) Давид Дале описывает дистилляцию нескольких больших NLP моделей для получения маленькой версии русскоязычной модели BERT. Маленький BERT по сути “учится” у больших моделей RuBERT, LaBSE, USE и T5, а описание того, что маленькая модель должна уметь (предсказывать замаскированные токены по контексту, строить представления токенов подобно тому, как это делают большие модели, предсказывать правильный порядок токенов в предложении) – это и есть составление сложной функции потерь.

Пример 1. Рекомендация новостного контента

Задача свелась к задаче ранжирования на табличных данных, и тут можно использовать функцию потерь, которую можно оптимизировать с помощью градиентного бустинга (т.е. дифференцируемую, это важно), для задачи ранжирования. Например, [LambdaMART](#).

Пример 2. Автоматическая оценка читаемости научной статьи

Тут задача свелась к задаче регрессии с текстовым входными данными, и можно использовать простую функцию потерь – среднеквадратичную ошибку (Mean Squared Error). Подробнее о функциях потерь в задаче регрессии можно прочитать в [посте](#) А.Г. Дьяконова.

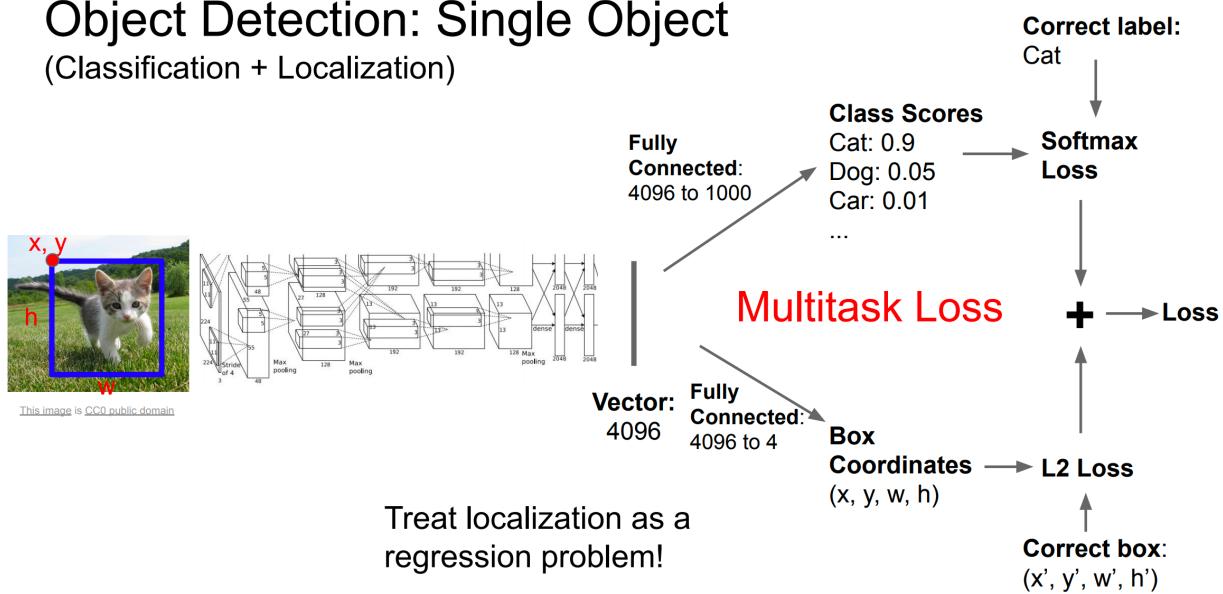
Пример 3. Детекция симптомов COVID-19 на рентгенограммах

В задаче детекции объектов на изображениях, как правило, для каждого объекта модель выдает 4 числа – координаты окна (bounding box) и его длину и ширину, а также вектор с числами – оценки принадлежности данного региона к каждому из классов. Поэтому функция потерь, как правило, складывается из двух других:

- Для сравнения предсказанного региона с реальным (согласно разметке в обучающей выборке) используется среднеквадратичная ошибка (Mean Squared Error).
- Для сравнения вектора оценок принадлежности региона к каждому из классов с реальным (согласно разметке в обучающей выборке) используется стандартная для задачи классификации функция потерь – логистическая, (также известная под термином “кросс-энтропия”), которую мы упоминали выше.

Object Detection: Single Object

(Classification + Localization)



Fei-Fei Li, Ranjay Krishna, Danfei Xu

Lecture 12 - 41

May 19, 2020

Fig. 20.6: Слайд стэнфордского курса cs231n про функцию потерь в задаче детекции объектов. [Источник](#).

20.2.5 Решатель

Когда мы определились с целевым признаком, обучающими данными, моделью и функцией потерь, мы наконец можем собственно обучать модель – то есть менять ее параметры так, чтобы все лучше и лучше предсказывать целевой признак, что меряется с помощью функции потерь.

Наиболее часто используемых вариантов два:

- явное решение (closed-form solution), в котором просто применяется формула для получения оптимальных параметров модели. Тут *Решателем* будет фреймворк, вычисляющий эту явно выписанную формулу, т.е. скорее всего производящий матричные умножения – например, NumPy.
- оптимизация параметров модели градиентными методами оптимизации. Тут *Решателем* будет алгоритм оптимизации и его конкретная реализация, например, в PyTorch

Самый известный пример первого варианта – это явное решение задачи наименьших квадратов. Существует прямо конкретная формула, включающая в себя перемножение матриц и векторов и взятие обратной матрицы, и дающая оптимальный (по минимизации среднеквадратичной ошибки) вектор весов линейной модели при наличии, конечно, обучающей выборки.

Получилось, что эта лекция вводная, без математических выкладок, поэтому не будем здесь приводить формулы без контекста, детали можно найти в любом классическом учебнике по статистике или машинному обучению или вкратце [в этой статье](#) на Хабре в разделе “Линейная регрессия”.

Во всех трех примерах задач, которые мы попутно рассматриваем и которые сводятся к обучению градиентного бустинга, SciBERT и YOLO соответственно, нет явного решения, которое простой формулой выдало бы оптимальные параметры модели, поэтому параметры подбираются с помощью методов оптимизации. В примере с градиентным бустингом “под капотом” – алгоритм обучения деревьев решений, с ним можно познакомиться [в этой статье](#). А в двух других примерах обучаются искусственные нейронные сети, и сегодня это делается как правило с помощью оптимизаторов, реализованных во фреймворках (PyTorch/TensorFlow/etc), причем аппаратная реализация, в которой работает *Решатель*, – это скорее видеокарта или TPU.

Ко второму из рассмотренных вариантов *Решателя* можно также отнести и автоматическое дифференцирование. Чтобы оптимизировать параметры модели градиентными методами, нужно знать собственно градиенты функции потерь – вектора частных производных функции потерь по параметрам. Попросту говоря, это ответ на вопрос, какие параметры модели стоит изменять и как это скажется на функции потерь, которую мы хотим минимизировать.

Иногда градиенты функции потерь можно найти аналитически, например, в случае линейной или логистической регрессии. Но как правило, в случае более сложных моделей с большим числом параметров аналитическое выражение для градиента функции потерь выглядит слишком громоздко, лучше вычислять его численно. Это задача методов автоматического дифференцирования. В частности, возможно, величайший алгоритм машинного обучения всех времен – алгоритм обратного распространения ошибки (backpropagation) – не что иное как численный метод нахождения производных функции потерь по параметрам модели.

20.2.6 Схема валидации и метрика качества

Наконец, когда мы определились с целевым признаком, обучающими данными, моделью, функцией потерь, а также научились подбирать параметры модели так, чтобы функция потерь уменьшалась, остался последний шаг – решить, как мы действительно поймем, что решаем задачу хорошо.

Выбор метрики качества напрямую связан с тем, чего мы хотим от модели машинного обучения в более широком контексте. Например, в бизнес-процессе могут быть ключевые показатели, которые мы *опосредованно* можем улучшать с помощью машинного обучения. Ключевыми показателями могут быть такие вещи как дневная аудитория приложения, Life-Time Value, показатели, связанные с удержанием (retention) клиентов/сотрудников, удовлетворенность клиентов и т.д. Многие из этих показателей нельзя замерять напрямую и оптимизировать, и тогда искусство Data Scientist-а заключается в том, чтобы выбрать простую метрику качества, которая бы задавалась понятной формулой (например, доля верных ответов или полнота) и при этом неплохо “коррелировала” с тем показателем бизнес-процесса, который хочется оптимизировать и таким образом приносить деньги компании или уменьшать операционные расходы.

Определение схемы валидации нужно, чтобы ответить на следующие вопросы:

- как понять, что модель сработает неплохо на новых, ранее не виденных данных
- как понять, что мы улучшили решение, поменяв модель, ее гиперпараметры или добавив новые признаки
- как понять, что одна модель лучше другой модели, один набор признаков лучше другого при фиксированной модели и т.д.

Во многом это связано с переобучением. Недостаточно просто замерить метрику качества на обучающей выборке. Надо хотя бы разбить выборку на 2 части: на одной обучать модель, на второй – проверить метрику качества. И чаще всего при больших объемах данных и больших моделях ровно так и делают. Но в мире “малых данных” и легковесных моделей более предпочтительна кросс-валидация. В этой схеме выборка делится на несколько частей, а модель обучается столько же раз. При этом каждая из подвыборок один раз является тестовой частью, на которой измеряется качество прогнозов, а все остальные разы она участвует в обучении модели. Таким образом, кросс-валидация дает более надежную оценку того, как модель сработает на новых данных, в сравнении с простым разбиением обучающей выборки на две части.

Вы вряд ли ошибетесь, если на практике будете применять 5-кратную стратифицированную (такую что распределение целевого признака примерно одинаковое в каждой подвыборке) кросс-валидацию, но надо понимать, что кросс-валидация – не панацея. И схема валидации, которую мы тут описываем, – это больше, чем просто кросс-валидация.

Часто выделяют “оффлайн” и “онлайн” режимы валидации. И метрики, полученные на кросс-валидации относятся к первому режиму, “оффлайн”. То есть мы один раз собрали обучающие данные, они больше не меняются, и вот с таким статическим срезом данных мы работаем, создаем признаки, обучаем модели, измеряем значения метрик на кросс-валидации. Но это не отвечает на вопрос, что получится, когда мы “выкатим модель в прод”, какие значения метрик ожидать на новых данных. И тут, как правило, устраиваются А/В-тесты, которые позволяют

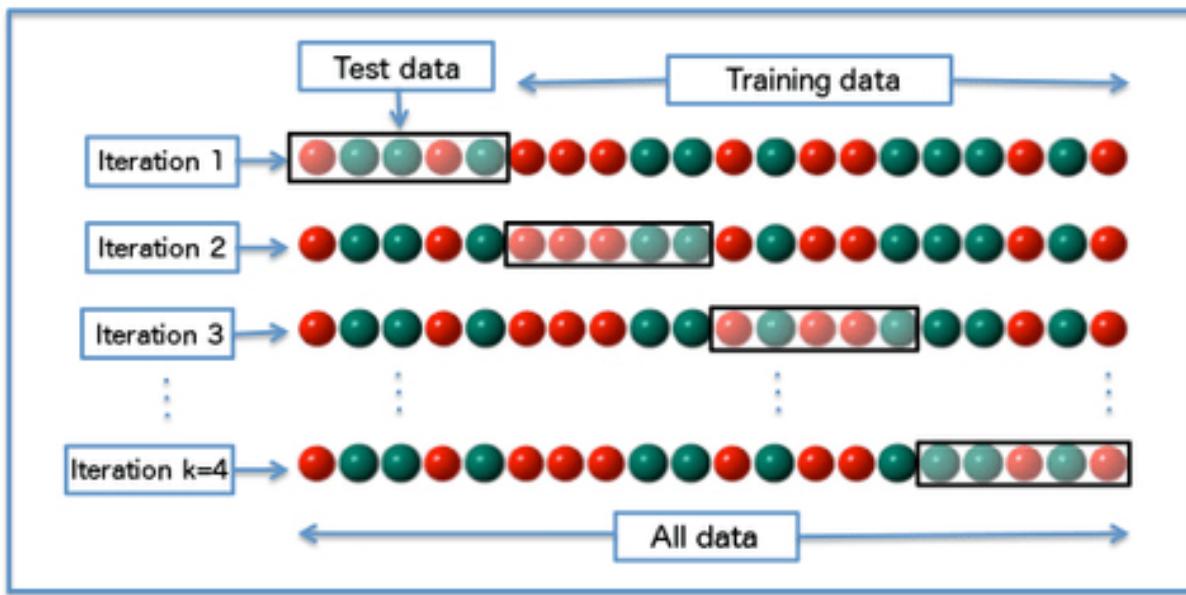


Fig. 20.7: Схема K-fold кросс-валидации.

понять, а действительно ли мы видим эффект от модели, лучше ли вообще с моделью, чем без нее, а если лучше с моделью, то какую модель выбрать. А/В-тестирование – это очень обширная тема, выходящая за рамки данной статьи, и развивать ее тут мы не будем. Только отметим, что определение схемы валидации – зачастую нетривиальный процесс и в зависимости от проекта может быть методологически сложным, в том числе и приводить к ошибкам.

Также заметим, что AutoML, о котором столь многие мечтают, решает некоторые вопросы, но далеко не все. В частности, алгоритмы AutoML не подберут вам правильную схему валидации. Они работают с уже имеющейся схемой валидации, и если валидация неверна, приводит к ошибкам, то тут и AutoML не поможет. Так что Data-Scientist-ов AutoML пока не вытеснит.

Пример 1. Рекомендация новостного контента

В этой задаче для выбора модели и ее гиперпараметров можно использовать обычную кросс-валидацию, но вот чтобы убедиться, что “в бою” модель также работает, лучше настроить А/В-тест (а точнее, для задач ранжирования намного лучше использовать [интерлидинг](#)). Пришло время сказать, что этот пример взят из практики автора этой лекции, Юрия Кашницкого, и в [выступлении на DataFest 2018](#) описывались сложности валидации модели в задаче рекомендации новостей. Вывод такой, что в задачах рекомендации (да и во многих других) надо устроить онлайн-проверку модели (А/В-тест, интерлидинг) помимо офлайн-проверки (кросс-валидации), только таким образом можно удостовериться, что модель действительно полезна, например, что использовать модель – лучше, чем просто показывать самый популярный контент.

Пример 2. Автоматическая оценка читаемости научной статьи

Здесь тоже может использоваться обычная кросс-валидация, хотя на практике из-за объемов данных и размера модели кросс-валидацию проводить будет дорого, придется удовлетвориться разбиением обучающей выборки на две части и проверкой модели на отложенной части.

Есть, конечно, детали. BERT не очень хорошо работает с длинными текстами, так что скорее всего мы разобьем полный текст статьи на параграфы и будем их подавать в модель по очереди. В таком случае лучше проводить [GroupKFold](#) кросс-валидацию так, чтобы на каждом этапе кросс-валидации в обучающей и проверочной выборке были параграфы из разных статей. Мы вряд ли хотим обучаться на одной половине статьи и проверять модель на второй ее половине – так бы мы получили слишком оптимистичную оценку качества модели.

Но даже при оговоренных тонкостях кросс-валидации самая большая сложность данной задачи – убедиться, что

метрика качества соответствует тому, что мы реально хотим получить в задаче. Это связано со сложностью определения “хорошо” и “плохо” написанного научного текста, о которой мы говорили выше. Поэтому в данной задаче, как и во многих других практических задачах машинного обучения, не обойтись без проверки результатов модели вручную. Такая проверка модели будет делаться уже после кросс-валидации, и в этом смысле она похожа на онлайн-оценку модели.

Пример 3. Детекция симптомов COVID-19 на рентгенограммах

Здесь схема проверки модели очень похожа на предыдущую. Сначала кросс-валидация или проверка модели на отложенной части, а потом – проверка предсказаний модели экспертами. Конечно, никто не будет по одной только кросс-валидации или результатам участников в соревновании Kaggle заключать, что модель прекрасно работает и ее можно нести врачам. В данном случае надо проверить модель на данных, приближенных к тем, которые будут использоваться врачами на практике. Чтобы не было [таких историй](#), как у Google Health, когда Deep Learning модель достигала 90% верных ответов при определении диабетической ретинопатии по фото зрачка, но при обучении на качественных снимках высокого разрешения. А при работе с менее качественными снимками система просто слишком часто отказывалась выносить вердикт из-за того, что была недостаточно уверена в прогнозе.

20.2.7 Заключение

В этой лекции мы описали, из чего складывается постановка задачи машинного обучения и рассмотрели, как общие компоненты проглядываются в разных по своей природе задачах. При этом мы поговорили о моделях-рабочих лошадках в трех разных областях: градиентном бустинге для табличных данных, BERT для текстов и YOLO для детекции изображений.

Немного пожертвовав, возможно, строгостью определения таких понятий как целевой признак или решатель, мы, надеюсь, описали все “пазлы” достаточно абстрактно, чтоб сложилось общее представление о том, как машинное обучение применяется в разных задачах, а также какие подводные камни стоит ожидать при боевом применении машинного обучения. Надеемся, это позволит лучше осознать взаимосвязь разных компонентов в квантово-классических схемах обучения, о которых речь пойдет далее в курсе.

CHAPTER
TWENTYONE

О БЛОКЕ “КВАНТОВЫЕ ВЫЧИСЛЕНИЯ”

Этот блок включает в себя:

- общий рассказ о том, что такое квантовый бит;
- введение в основные квантовые гейты.

Продвинутые темы блока дополнительно рассказывают:

- о квантовой физике, на которой базируется концепция кубита;
- о смешанных состояниях, операторе плотности и энтропии фон Неймана.

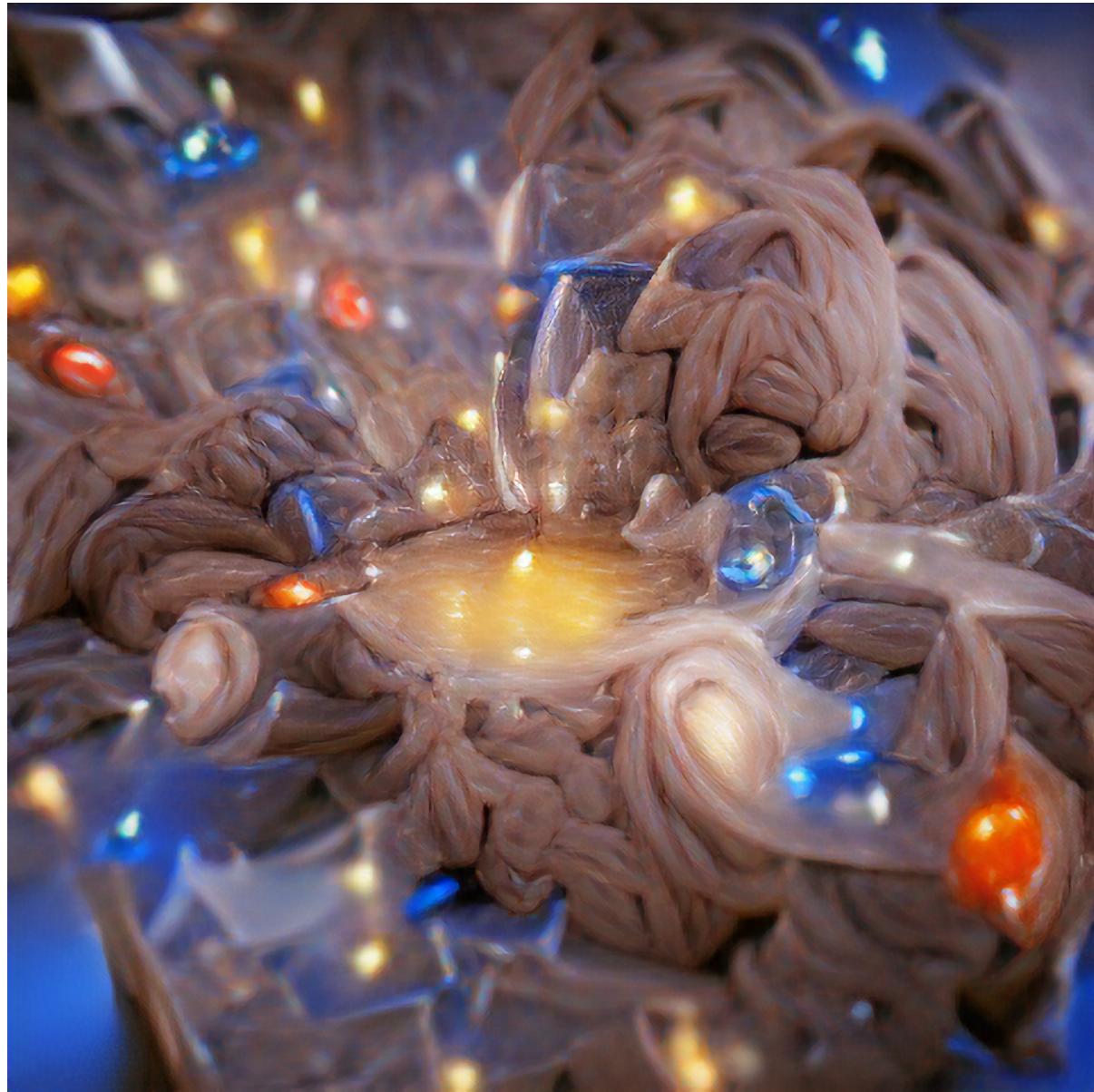


Fig. 21.1: Квантовый разум “глазами” VQGAN и CLIP, сгенерировано в Google Colab

КВАНТОВЫЙ БИТ

22.1 Описание лекции

Эта лекция расскажет:

- что такое кубит;
- в чем разница между значением и состоянием;
- что такое сфера Блоха;
- какие можно делать операции над кубитами;
- что такое измерение.

22.2 Введение

Это первая лекция основного блока нашего курса. Прежде чем мы начнем детально разбирать понятие кубита, давайте взглянем на общий пайплайн квантовых схем.

Любая квантовая схема включает в себя:

- кубиты, инициализируемые в начальное состояние, обычно 0;
- унитарные и обратимые операции над кубитами;
- измерение кубитов.

Эта лекция посвящена разбору операций для одного кубита. Начнем с понятия кубита и его отличий от бита классических компьютеров.

22.3 Что такое кубит

Классический компьютер оперирует двоичными числами – нулем и единицей. Минимальный объем информации для классического компьютера называется бит. Квантовый компьютер оперирует квантовыми битами или кубитами, которые тоже имеют два возможных значения – 0 и 1. Так в чем же разница? В чем особенности квантовых компьютеров, которые дают им преимущества над классическими компьютерами?

Разница в том, что для квантовомеханических систем (и кубитов в частности) их *состояния и значения* – это не одно и то же.

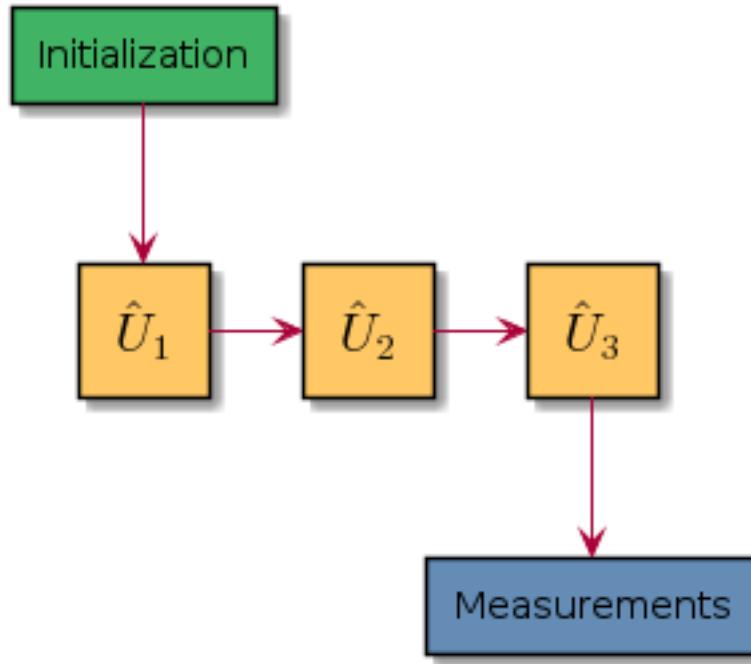


Fig. 22.1: Схема любого квантового алгоритма

22.4 Состояние vs значение

22.4.1 Состояние классического бита

Обычно мы не отличаем состояние классического бита от его значения и считаем, что если бит имеет значение **1**, то и состояние его описывается числом **1**.

22.4.2 Кот Шредингера

Давайте вспомним мысленный эксперимента Шредингера. Кот, который одновременно и жив, и мертв. Понятно, что *значение* кота точно одно: он либо жив, либо мертв. Но *состояние* его более сложное. Он находится в *superпозиции* состояний “жив” и “мертв” одновременно.

22.4.3 Состояние кубита

Состояние кубита, если можно так сказать, аналогично состоянию кота Шредингера. Оно отличается от *значения* кубита и описывается вектором из двух комплексных чисел. Мы будем обозначать состояния (или вектора) символом Ψ (кет – вектор-столбец) – это широко принятая в квантовой механике и квантовых вычислениях нотация Дирака:

$$\Psi = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$$

Note: Может возникнуть вопрос, а почему комплексные числа? Короткий ответ на этот вопрос дать сложно. Если в двух словах, то использование комплексных чисел связано с удобством представления матричных групп, используемых в квантовой механике.

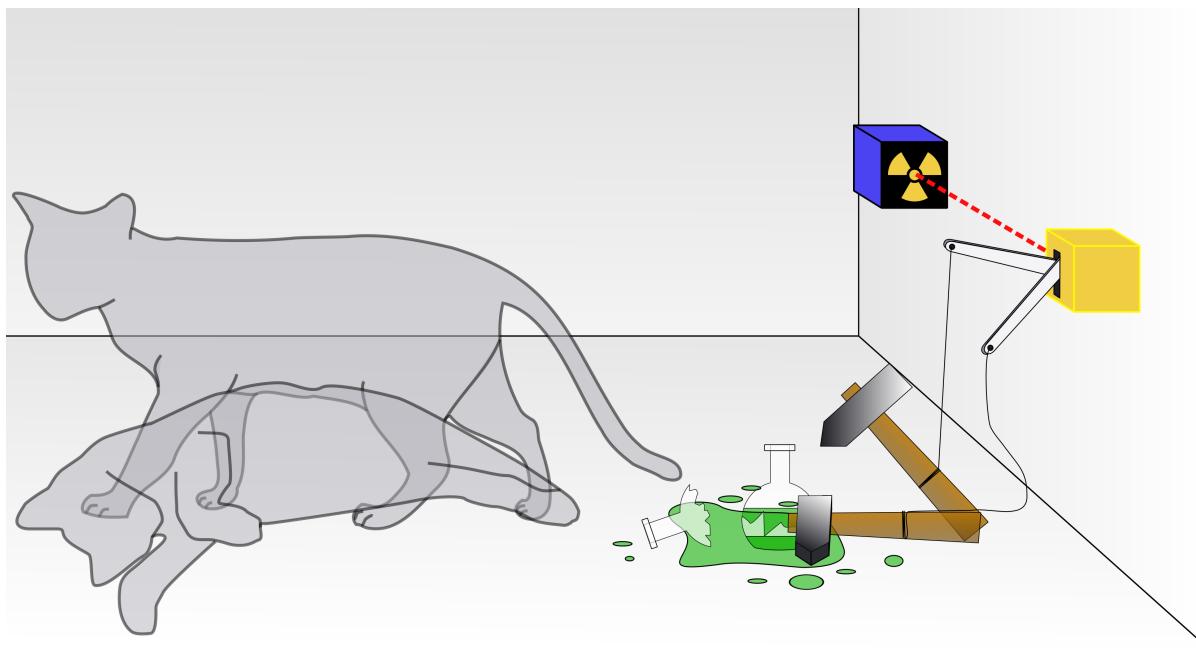


Fig. 22.2: Кот Шредингера

Все еще звучит сложно? Тогда нужно вспомнить, что изначально квантовая механика возникла в том числе из-за того, что физики экспериментально обнаружили у фундаментальных частиц свойство корпускулярно-волнового дуализма. Иными словами, электроны, фотоны и другие частицы проявляли как типичные свойства волнового движения (например, интерференцию и дифракцию), и свойства частиц – например, всегда есть минимальная порция (квант!) света или электрического поля. Кстати, часто вместо вектора состояния используется понятие волновой функции, которая описывает плотность вероятности обнаружить частицу в той или иной точке пространства (обычного или специального). Ко времени создания квантовой механики для описания волнового движения ученые уже привыкли использовать комплексные числа, которые позволяют упростить описание многих эффектов за счет разделения амплитуды и фазы процесса. Такое удобство справедливо и для многих задач квантовой физики.

Для более детального ответа авторы курса рекомендуют читать книги по истории квантовой физики (и по самой квантовой физике).

Значение чисел c_0 и c_1 мы обсудим чуть позже, а пока запишем наш кубит Ψ в коде Python. Для начала $c_0 = c_1 = \frac{1}{\sqrt{2}}$.

```
import numpy as np
qubit = np.array([1 / np.sqrt(2) + 0j, 1 / np.sqrt(2) + 0j]).reshape((2, 1))
```

Здесь мы создаем именно вектор-столбец размерности 2×1 .

$$\Psi = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

```
print(qubit.shape)
```

```
(2, 1)
```

22.4.4 Связь состояния и значения кубита

Разберем подробнее вектор Ψ и значение цифр c_0, c_1 . Посмотрим на состояния кубита, значение которого мы знаем точно. То есть “посмотрим на кота Шредингера”, но который точно жив или точно мертв.

Базисные состояния

Посмотрим, как выглядят состояния кубитов с точно определенными значениями:

$$0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad 1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Что мы можем сказать об этих состояниях? Как минимум следующее:

- они ортогональны ($0 \perp 1$);
- они имеют единичную норму;
- они образуют базис.

Что это значит для нас? А то, что любое состояние Ψ можно записать как линейную комбинацию векторов 0 и 1, причем коэффициентами в этой комбинации будут как раз c_0, c_1 :

```
basis_0 = np.array([1 + 0j, 0 + 0j]).reshape((2, 1))
basis_1 = np.array([0 + 0j, 1 + 0j]).reshape((2, 1))

c0 = c1 = 1 / np.sqrt(2)

print(np.allclose(qubit, c0 * basis_0 + c1 * basis_1))
```

True

$$\Psi = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

Амплитуды вероятностей

Квантовая механика устроена таким интересным образом, что если мы будем измерять **значение** кубита, то вероятность каждого из вариантов будет пропорциональна соответствующему коэффициенту в разложении **состояния**. Но так как амплитуды – это в общем случае комплексные числа, а вероятности должны быть строго действительные, нужно домножить амплитуды на комплексно сопряженные значения. В случае наших значений $c_0 = c_1 = \frac{1}{\sqrt{2}}$ получаем:

```
p0 = np.conj(c0) * c0
p1 = np.conj(c1) * c1

print(np.allclose(p0, p1))
print(np.allclose(p0 + p1, 1.0))
```

True
True

Видим еще одну важную вещь: сумма вероятностей всех состояний должна быть равна 100%. Это сразу приводит нас к тому, что состояния – это не любые комплексные вектора, а комплексные вектора с единичной нормой:

```
print(np.allclose(np.conj(qubit).T @ qubit, 1.0))
```

```
True
```

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = 1.0$$

Мы будем очень часто пользоваться транспонированием и взятием комплексно сопряженного от векторов. В квантовой механике это имеет специальное обозначение $\Psi = \Psi^{T^*} = \Psi^\dagger$ (бра – вектор-строка). Тогда наше правило нормировки из NumPy кода может быть записано в нотации Дирака так:

$$\Psi|\Psi = 1$$

22.5 Сфера Блоха

Описанный выше базис 0, 1 не является единственным возможным. Вектора 0, 1 – это лишь самый часто применимый базис, который называют **Z** базисом. Но есть и другие варианты.

22.5.1 Возможные базисы

Z-базис

Уже описанные нами 0 и 1.

X-базис

Базисные состояния $+$ = $\frac{0+i1}{\sqrt{2}}$ и $-$ = $\frac{0-i1}{\sqrt{2}}$:

```
plus = (basis_0 + basis_1) / np.sqrt(2)
minus = (basis_0 - basis_1) / np.sqrt(2)
```

Y-базис

Базисные состояния R = $\frac{0+i1}{\sqrt{2}}$ и L = $\frac{0-i1}{\sqrt{2}}$:

```
R = (basis_0 + 1j * basis_1) / np.sqrt(2)
L = (basis_0 - 1j * basis_1) / np.sqrt(2)
```

Легко убедиться, что все вектора каждого из этих базисов ортогональны:

```
print(np.allclose(np.conj(basis_0).T @ basis_1, 0))
```

```
True
```

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 0$$

```
print(np.allclose(np.conj(plus).T @ minus, 0))
```

```
True
```

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} = 0$$

```
print(np.allclose(np.conj(R).T @ L, 0))
```

```
True
```

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & -i\frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -i\frac{1}{\sqrt{2}} \end{bmatrix} = 0$$

Заметьте, что в наших векторных пространствах скалярное произведение – это (в случае действительных векторов) $\vec{a}\vec{b} = \langle a|b \rangle$ (бра-кет). Именно поэтому нужно делать транспонирование и комплексное сопряжение первого вектора в паре.

22.5.2 Сфера Блоха

Обозначения $0, 1, +, -, R, L$ выбраны неслучайно: они имеют геометрический смысл.

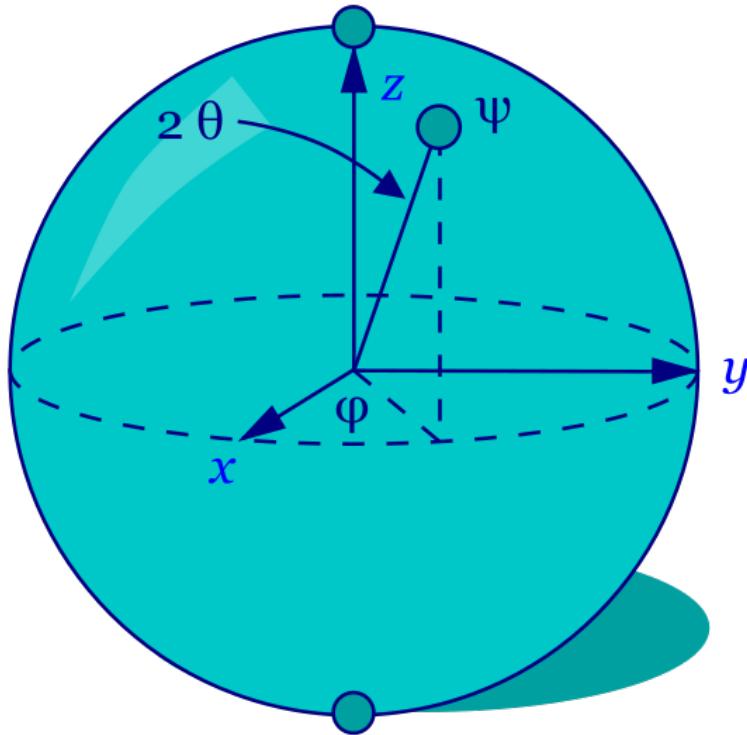


Fig. 22.3: Сфера Блоха

Принято считать, что ось **Z** – это основная ось, так как физически квантовые компьютеры измеряют именно по ней. Ось **X** “смотрит на нас” и поэтому обозначается $+$ и $-$. А ось **Y** направлена как бы вдоль, поэтому базис обозначают как “право” (R) и “лево” (L).

Вектор состояния кубита еще называют волновой функцией, и этот вектор может идти в любую точку сферы Блоха. Сама сфера имеет единичный радиус, и это гарантирует нам, что для всех состояний сумма квадратов амплитуд будет равна единице.

Состояние в полярных координатах

Состояние кубита можно выразить через полярные координаты на сфере Блоха:

$$\Psi = c_0 0 + c_1 1 = \cos \theta 0 + e^{i\phi} \sin \theta 1,$$

где θ, ϕ – это угловые координаты на сфере Блоха. В этом смысле сфера Блоха очень удобна для представления состояний одного кубита.

Note: Тут мы воспользовались формулой Эйлера, а также вынесли за скобки локальные фазы множителей c_0 и c_1 . Если у вас возникают трудности с подобными операциями над комплексными числами, то рекомендуем еще раз пересмотреть базовый [блок](#) нашего курса по линейной алгебре и комплексным числам, там эти моменты освещаются более подробно.

22.6 Что можно делать с таким кубитом?

22.6.1 Линейные операторы

Любое действие, которое мы совершаем с кубитом в состоянии Ψ , должно переводить его в другое состояние Φ . Что переводит один вектор в другой вектор в том же пространстве? Правильно, матрица. Другими словами, линейный оператор. Мы будем обозначать операторы как \hat{U} .

22.6.2 Унитарность

Как мы уже говорили, квадраты амплитуд – это вероятности. Следовательно, волновая функция должна быть нормирована на единицу. А значит, любой оператор, который переводит одно состояние в другое $\hat{U}\Psi = \Phi$, должен сохранять эту нормировку, то есть должен быть [унитарным](#). Более того, свойство унитарности приводит к тому, что любой квантовый оператор еще и сохраняет скалярное произведение:

$$\Psi \hat{U}^\dagger \hat{U} \Psi = \Psi \Psi$$

Другими словами, унитарный оператор удовлетворяет условию $\hat{U}^\dagger \hat{U} = \hat{I}$.

22.6.3 Обратимость

Одно из важных следствий унитарности операций над кубитами – это их обратимость. Если вы сделали какую-то последовательность унитарных операций над кубитами \hat{U} , то их можно вернуть в начальное состояние, ведь у унитарного оператора всегда есть обратный оператор $\hat{U}^{-1} = \hat{U}^\dagger$.

Note: Квантовый компьютер должен уметь делать несколько не унитарных операций, например, инициализацию кубита в определенное состояние (например, 0) и считывание состояния кубитов. Такие неунитарные операции приводят к потере информации и являются необратимыми.

22.6.4 Пример оператора

В дальнейших лекциях мы разберем много операторов, так как именно операторы (или квантовые гейты) являются основой квантовых вычислений. А пока рассмотрим простой пример: оператор Адамара (**Hadamard gate**), который переводит $0 \rightarrow +$.

Гейт Адамара

Начнем с того, что пока у нас лишь один кубит. Состояние одного кубита – это вектор размерности два. Значит, оператор, который переводит его в другой вектор размерности два – это матрица 2×2 . Запишем оператор Адамара в матричном виде, а потом убедимся, что он унитарный и действительно переводит состояние $0 \rightarrow +$.

$$\hat{H} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Реализация в Python

```
h = 1 / np.sqrt(2) * np.array([
    [1 + 0j, 1 + 0j],
    [1 + 0j, 0j - 1]
])
```

Унитарность

```
print(np.allclose(np.conj(h).T @ h, np.eye(2)))
```

```
True
```

$$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Проверка

Проверим, что гейт Адамара действительно переводит кубит из состояния 0 в состояние $+$.

```
print(np.allclose(h @ basis_0, plus))
```

```
True
```

$$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$

22.7 Измерение

Измерение в квантовых вычислениях выделяется отдельно именно потому, что оно “открывает” коробку с котом Шредингера: мы точно узнаем, жив он или мертв, и уже никогда не сможем это “забыть” обратно. Вся *суперпозиция* его состояния исчезает. То есть *измерение* – это как раз пример одной из не унитарных операций, которые должен уметь делать квантовый компьютер.

Note: Это интересный факт: исчезновение суперпозиции многим кажется парадоксом, именно поэтому и появляются разные интерпретации квантовой механики, например, многомировая интерпретация Эверетта. Действительно, это кажется немного странным, что полностью обратимая квантовая механика и непрерывная динамика волновых функций вдруг “ломаются” и мы получаем такой коллапс, который еще называют редукцией фон Неймана. Доктору Эверетт тоже это не нравилось и он предложил другую интерпретацию этого процесса. Согласно его теории, когда мы производим измерения, мы как бы “расщепляем” нашу вселенную на две ниточки: в одной кот остается жив, а в другой остается мертв.

Такие теории остаются на уровне спекуляций, так как почти невозможно придумать эксперимент, который бы подтверждал или опровергал такую гипотезу. Скорее это вопрос личного понимания и интерпретации процесса, так как математически подобные теории в итоге дают один и тот же наблюдаемый и измеримый результат.

Как мы уже говорили, состояние кубита может быть записано в разных базисах: $0, 1, +, -, R, L$. *Значение* кубита в каждом из этих базисов может быть измерено. Но что такое измерение с точки зрения математики?

22.7.1 Операторы Паули

На самом деле, любая наблюдаемая величина соответствует какому-то оператору. Например, измерения в разных базисах **X**, **Y**, **Z** соответствуют операторам Паули:

$$\hat{\sigma}^x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \hat{\sigma}^y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad \hat{\sigma}^z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

```
 pauli_x = np.array([[0 + 0j, 1 + 0j], [1 + 0j, 0 + 0j]])
 pauli_y = np.array([[0 + 0j, 0 - 1j], [0 + 1j, 0 + 0j]])
 pauli_z = np.array([[1 + 0j, 0 + 0j], [0 + 0j, 0j - 1j]])
```

Эти операторы очень важны, рекомендуется знать их наизусть, так как они встречаются в каждой второй статье по квантовым вычислениям, а также постоянно фигурируют в документации всех основных библиотек для квантового машинного обучения.

22.7.2 Собственные значения

Мы поняли, что есть связь между нашими измерениями и операторами. Но какая именно? Что значит, например, что измерения по оси **Z** соответствуют оператору $\hat{\sigma}^z$?

Здесь мы приходим к собственным значениям операторов. Оказывается (так устроен наш мир), что *измеряя* какую-то величину в квантовой механике, мы всегда будем получать одно из собственных значений соответствующего оператора, а состояние будет коллапсировать в соответствующий собственный вектор этого оператора. Другими словами, *измеряя* кота Шредингера, мы будем получать значения “жив” или “мертв”, а состояние кота будет переходить в состояние, соответствующее одному из этих значений. А еще *измерение* не является обратимой операцией: однажды открыв коробку с котом и поняв, жив он или мертв, мы уже не сможем закрыть ее обратно и вернуть кота в суперпозицию.

Описанное выше – не абстрактные рассуждения из квантовой физики. Оно пригодится, когда мы будем говорить о решении практических комбинаторных задач, таких как задача о выделении сообществ в графе.

22.7.3 Собственные векторы $\hat{\sigma}^z$

Вернемся к нашему оператору $\hat{\sigma}^z$. Легко убедиться, что его собственные значения равны 1 и -1, а соответствующие им собственные вектора – это $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ и $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$:

```
print(np.linalg.eig(pauli_z))
```

```
(array([ 1.+0.j, -1.+0.j]), array([[1.+0.j, 0.+0.j],
[0.+0.j, 1.+0.j]]))
```

Таким образом, измерение по оси **Z** всегда будет давать нам одно из этих двух значений и переводить состояние кубита в соответствующий собственный вектор.

Caution: Зачастую кубиты измеряют именно в **Z**-базисе; является неким “стандартом” для квантовых вычислений, так как это измерение “ближе к железу”. Также **Z**-базис удобен для нас из-за диагональности *оператора Паули* σ^z .

22.7.4 Формальная запись

Формально мы можем записать для любого эрмитова оператора \hat{U} , что собственные состояния этого оператора являются его собственными векторами, а собственные значения в этом случае являются наблюдаемыми значениями:

$$\hat{U}\Psi = u\Psi$$

22.7.5 Другие операторы Паули

Убедимся, что у остальных операторов собственные значения такие же:

```
print(np.linalg.eig(pauli_x))
print(np.linalg.eig(pauli_y))
```

```
(array([ 1.+0.j, -1.+0.j]), array([[ 0.70710678-0.j,  0.70710678+0.j],
[ 0.70710678+0.j, -0.70710678-0.j]]))
(array([ 1.+0.j, -1.+0.j]), array([[[-0.          , -0.70710678j,  0.70710678+0.j],
[ 0.70710678+0.j,  0.          , -0.70710678j]]]))
```

Note: Заметим, что собственные вектора могут отличаться на какой-то множитель. В частности, один из собственных векторов оператора $\hat{\sigma}^y$, возвращенный `np.linalg.eig` равен $\frac{1}{\sqrt{2}} \begin{bmatrix} -i \\ 1 \end{bmatrix}$, что отличается от $R = \frac{0+i1}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ i \end{bmatrix}$ только домножением на i .

Во-первых, по курсу линейной алгебры мы знаем, что собственные вектора можно домножать на любую константу и они все равно останутся собственными, поскольку если вектор x^* – решение уравнения $Ax^* = \lambda x^*$, то и любой вектор kx^* , $k \in \mathbb{C}$, $k \neq 0$ также будет решением этого уравнения.

Во-вторых, состояния, отличающиеся только множителем, по сути соответствуют одному и тому же состоянию. Любой множитель $c \in \mathbb{C}$, стоящий перед вектором состояния Ψ можно представить в виде $c = e^{i\phi}$, что соответствует повороту на некоторый угол ϕ . При вычислении амплитуды $|\Psi|^2$ этот множитель даст единицу, то есть, другими словами, поворот кубита никак не влияет на результат измерения кубита. Это известно как *глобальная фаза*, и в научной литературе часто можно встретить фразу “*up to a global phase factor*”, что означает одинаковые состояния с точностью до поворота (примерно как произвольная константа, добавляемая к интегралу).

Note: Также можно заметить, что у всех операторов Паули нет ни одного общего собственного вектора. Таким образом, мы приходим к ситуации, когда не можем одновременно точно провести измерения двумя разными операторами, так как наше измерение должно переводить состояние в соответствующий собственный вектор. В квантовой механике это называется **принципом неопределенности**.

22.7.6 Ожидаемое значение при измерении

Мы не будем писать с нуля полный симулятор кубитов, который включает измерения – это требует введения сложного случайного процесса. Но мы можем легко ответить на другой вопрос. А именно: можно ли сказать, какое будет *ожидаемое значение* оператора \hat{U} для состояния Ψ ? Другими словами, какое будет математическое ожидание большого числа измерений? Это можно записать следующим образом:

$$\mathbf{E}(\hat{U}) = \Psi \hat{U} \Psi$$

Например, оператор $\hat{\sigma}^z$ полностью не определен в состоянии $+$, то есть мы будем равновероятно получать значения -1 и 1 , а математическое ожидание, соответственно, будет равно нулю:

```
print(plus.conj().T @ pauli_z @ plus)
```

```
[[ -2.23711432e-17+0.j ]]
```

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = 0$$

С другой стороны, измеряя состояние $+$ в X -базисе мы всегда будем получать 1:

```
print(plus.conj().T @ pauli_x @ plus)
```

```
[[ 1.+0.j ]]
```

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = 1$$

22.7.7 Вероятности битовых строк

Последнее, чего мы коснемся в части измерений – это битовые строки и метод Шредингера. Мы много говорили о вероятностной интерпретации волновой функции и аналогиях с классическим битом, но пока этого никак не касались на практике. Как же получить вероятность определенной битовой строки для произвольного состояния? Если взять все битовые строки размерности вектора состояния и отсортировать их в лексикографическом порядке (например, $0 < 1, 00 < 01 < 10 < 11$, и т.д.), то вероятность каждой битовой строки получается следующим выражением:

$$\mathbf{P} = |\Psi \vec{s}|^2,$$

где \vec{s} – это вектор, каждая компонента которого соответствует порядковой битовой строке или вектор битовых строк. Другими словами, вероятность получить i -ю битовую строку равна квадрату i -го элемента амплитуды волновой функции. Кажется немного запутанным, но на самом деле $|\Psi|^2$ – это идеально и есть плотность вероятности.

22.7.8 Еще пара слов об измерениях

Измерение как проекция на пространство собственных векторов

Мы уже говорили, что при измерении мы как бы “выбираем” один из собственных векторов наблюдаемой. Более строго такой процесс называется проецированием на пространство собственных векторов. Для собственного вектора Φ проекция будет линейным оператором:

$$\hat{P}_\Phi = \Phi\Phi$$

```
super_position = h @ basis_0
eigenvectors = np.linalg.eig(pauli_z) [1]

proj_0 = eigenvectors[0].reshape((-1, 1)) @ eigenvectors[0].reshape((1, -1))
proj_1 = eigenvectors[1].reshape((-1, 1)) @ eigenvectors[1].reshape((1, -1))
```

$$\begin{aligned}\hat{P}_{\Phi_0} &= \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes [1 \ 0] = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \\ \hat{P}_{\Phi_1} &= \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes [0 \ 1] = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}\end{aligned}$$

Правило Борна

Как мы уже говорили, любому эрмитову оператору соответствует какая-либо наблюдаемая величина. А какая наблюдаемая величина соответствует оператору проекции на собственный вектор Φ , про который мы говорили выше? Ответ – вероятность наблюдения собственного значения, которое соответствует этому собственному вектору. Значит, чтобы получить вероятность измерения значения λ_i эрмитова оператора \hat{A} (которое соответствует собственному вектору Φ_i этого оператора) в состоянии Ψ , мы должны измерить величину $\Psi \hat{P}_{\Phi_i} \Psi$. Это называется правилом Борна.

$$\mathbf{P}(\lambda_i) = \Psi \hat{P}_{\Phi_i} \Psi$$

Считать ожидаемое значение оператора мы уже умеем. Давайте убедимся, что для состояния $\Psi = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ результаты измерений операторов проекций дадут 0.5 и совпадут с результатом упражнения, которое мы проделали ранее:

```
p_0 = super_position.conj().T @ proj_0 @ super_position
print(np.allclose(p_0, 0.5))
```

```
True
```

$$\mathbf{P}(\lambda_0) = \left(\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \right) \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \left(\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \right) = \frac{1}{2}$$

```
p_1 = super_position.conj().T @ proj_1 @ super_position  
  
print(np.allclose(p_1, 0.5))  
print(np.allclose(p_0 + p_1, 1.0))
```

```
True  
True
```

$$\mathbf{P}(\lambda_1) = \left(\begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \right) \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \left(\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \right) = \frac{1}{2}$$

22.8 Что мы узнали?

- Состояние и значение для кубита – это не одно и то же.
- Состояния представляют собой комплекснозначные вектора.
- Квантовые операторы – унитарные и самосопряженные.
- Измеряемые значения – собственные значения операторов.
- Измерение “ломает” суперпозицию.

КВАНТОВЫЕ ГЕЙТЫ

23.1 Описание лекции

Из этой лекции мы узнаем:

- какие есть основные однокубитные и многокубитные гейты;
- как записывать многокубитные состояния;
- как конструировать многокубитные операторы;
- как работать с библиотекой PennyLane.

23.2 Введение

Квантовые гейты являются основными *строительными* блоками для любых квантовых схем, в том числе и тех, что применяются для машинного обучения. Можно сказать, что это своеобразный алфавит квантовых вычислений. Он необходим, чтобы сходу понимать, например, что изображено на подобных схемах:

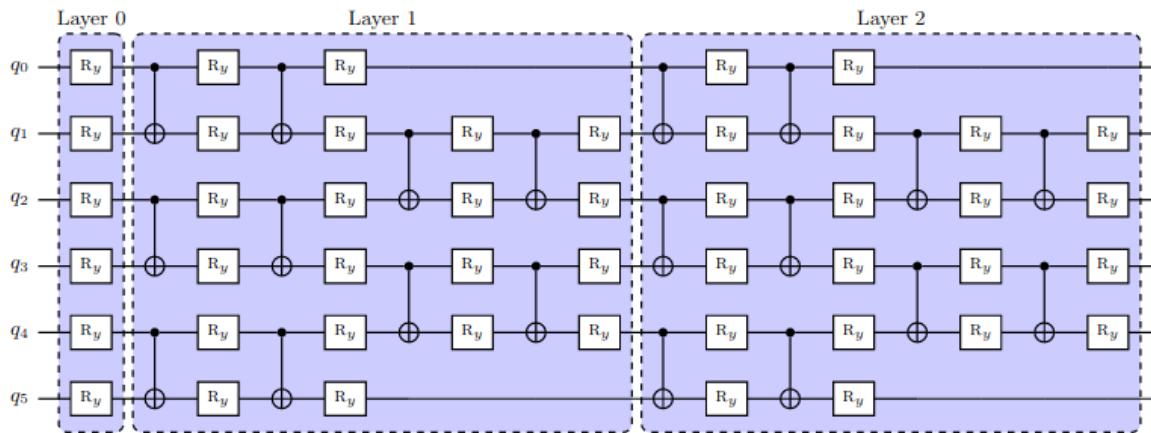


FIG. 2: L-VQE ansatz for a 6-qubit quantum state. R_y denotes rotation around the y -axis defined as $R_y(\theta) \equiv e^{-i\frac{\theta}{2}Y}$. Every R_y contains a parameter that is optimized over in the outer loop.

Fig. 23.1: Схема Layered-VQE

23.3 Основные однокубитные гейты

В прошлый раз мы познакомились с операторами Паули, а также гейтом Адамара. Как для обычных квантовых алгоритмов, так и для QML-алгоритмов нужны и другие гейты, потому что одни только эти гейты не позволяют перейти во все возможные квантовые состояния. Теперь давайте посмотрим, какие еще однокубитные гейты часто применяются в квантовых вычислениях и квантовом машинном обучении.

23.3.1 Т-гейт

Т-гейт очень популярен в универсальных квантовых вычислениях. Его матрица имеет вид:

$$\hat{T} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1+i}{\sqrt{2}} \end{bmatrix}$$

Любой однокубитный гейт можно аппроксимировать последовательностью гейтов Адамара и Т-гейтов. Чем точнее требуется аппроксимация, тем длиннее будет аппроксимирующая последовательность.

Помимо важной роли в математике квантовых вычислений, гейт Адамара и Т-гейт интересны тем, что именно на них построено большинство предложений по реализации квантовых вычислений с топологической защитой или с коррекцией ошибок. На сегодняшний день эти схемы реально пока не очень работают: никаких топологически защищенных кубитов продемонстрировано не было, а коррекция ошибок не выходит за пределы двух логических кубитов.

23.3.2 Гейты поворота вокруг оси

Поворотные гейты играют центральную роль в квантовом машинном обучении. Вспомним на секунду, как выглядят наши однокубитные состояния на сфере Блоха:

Любой однокубитный гейт можно представить как вращение вектора состояния Ψ на некоторый угол вокруг некоторой оси, проходящей через центр сферы Блоха.

Гейты $\hat{R}X(\phi), \hat{R}Y(\phi), \hat{R}Z(\phi)$ осуществляют поворот на определенный угол ϕ вокруг соответствующей оси на сфере Блоха.

Давайте внимательно рассмотрим это на примере гейта $\hat{R}Y$.

Гейт $\hat{R}Y$

Сам гейт определяется следующим образом:

$$\hat{R}Y(\phi) = \begin{bmatrix} \cos(\frac{\phi}{2}) & -\sin(\frac{\phi}{2}) \\ \sin(\frac{\phi}{2}) & \cos(\frac{\phi}{2}) \end{bmatrix}$$

```
import numpy as np

def ry(state, phi):
    return np.array([
        [np.cos(phi / 2), -np.sin(phi / 2)],
        [np.sin(phi / 2), np.cos(phi / 2)]
    ]) @ state
```

Запишем наше состояние 0:

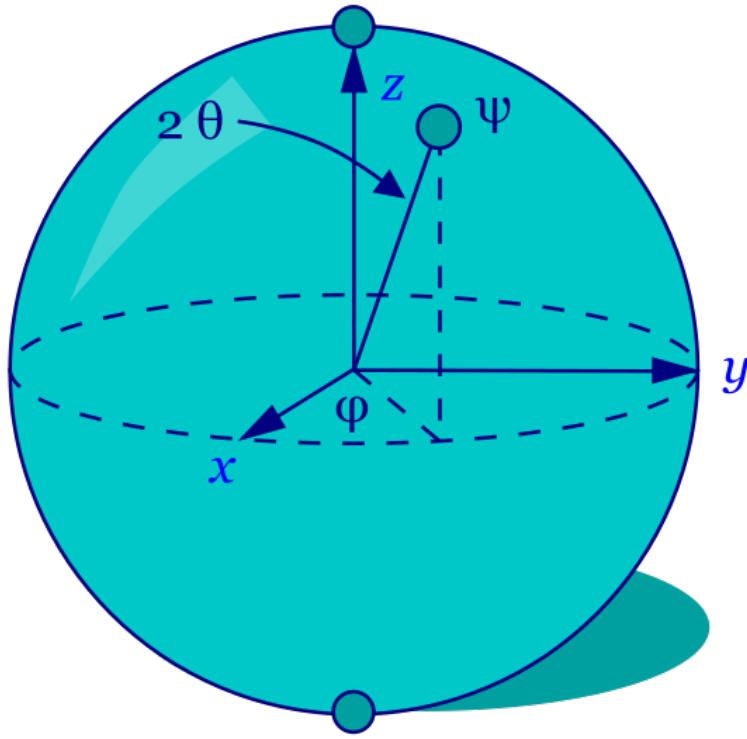


Fig. 23.2: Сфера Блоха

```
basis = np.array([1 + 0j, 0 + 0j]).reshape((2, 1))
```

Внимательно посмотрим на сферу Блоха. Можно заметить, что если повернуть состояние из 0 на $\frac{\pi}{2}$ и измерить значение $\hat{\sigma}^x$, то получится 1. А если повернуть на $-\frac{\pi}{2}$, то получится -1:

```
def expval(state, op):
    return state.conj().T @ op @ state

pauli_x = np.array([[0 + 0j, 1 + 0j], [1 + 0j, 0 + 0j]])

print(np.allclose(expval(ry(basis, np.pi / 2), pauli_x), 1.0))
print(np.allclose(expval(ry(basis, -np.pi / 2), pauli_x), -1.0))
```

```
True
True
```

Убедимся также, что вращение на угол, пропорциональный 2π , не меняет результат измерения. Возьмем случайное состояние:

$$\Psi = \begin{bmatrix} 0.42 \\ \sqrt{1 - 0.42^2} \end{bmatrix}$$

```
random_state = np.array([0.42 + 0j, np.sqrt(1 - 0.42**2) + 0j]).reshape((2, 1))
```

Измерим его по осям **X** и **Z**, затем повернем на угол 2π и измерим снова:

```

pauli_z = np.array([[1 + 0j, 0 + 0j], [0 + 0j, 0j - 1]])

print("Z:\n\t" + str(expval(random_state, pauli_z)) + "\n")
print("X:\n\t" + str(expval(random_state, pauli_x)) + "\n")

print("Z after RY:\n\t" + str(expval(ry(random_state, 2 * np.pi), pauli_z)) + "\n")
print("X after RY:\n\t" + str(expval(ry(random_state, 2 * np.pi), pauli_x)) + "\n")

```

```

Z:
[[[-0.6472+0.j]]]

X:
[[0.76232025+0.j]]

Z after RY:
[[[-0.6472+0.j]]]

X after RY:
[[0.76232025+0.j]]

```

Другие гейты вращений

Аналогичным образом определяются гейты $\hat{R}X$ и $\hat{R}Z$:

$$\hat{R}X(\phi) = \begin{bmatrix} \cos(\frac{\phi}{2}) & -i \sin(\frac{\phi}{2}) \\ -i \sin(\frac{\phi}{2}) & \cos(\frac{\phi}{2}) \end{bmatrix} \quad \hat{R}Z(\phi) = \begin{bmatrix} e^{-\frac{i\phi}{2}} & 0 \\ 0 & e^{\frac{i\phi}{2}} \end{bmatrix}$$

Общая форма записи однокубитных гейтов

В общем случае однокубитные гейты могут быть также записаны следующим образом:

$$\hat{R}^{\vec{n}(\alpha)} = e^{-\frac{i\alpha\hat{\sigma}\cdot\vec{n}}{2}},$$

где α – это угол поворота, \vec{n} – единичный вектор в направлении оси поворота, а $\hat{\sigma} = \{\hat{\sigma}^x, \hat{\sigma}^y, \hat{\sigma}^z\}$ – это вектор, составленный из операторов Паули. Если использовать покоординатную запись и $\vec{n} = \{n_x, n_y, n_z\}$ задает ось вращения, то

$$\hat{R}^{\vec{n}(\alpha)} = e^{-i\frac{\alpha}{2}(\hat{\sigma}^x n_x + \hat{\sigma}^y n_y + \hat{\sigma}^z n_z)}.$$

Забегая вперед, можно сказать, что именно гейты вращений – это основа *квантовых вариационных схем*, главного инструмента этого курса.

23.3.3 Phase-shift гейт

Другой важный гейт – это так называемый phase-shift гейт, или \hat{U}_1 гейт. Его матричная форма имеет следующий вид:

$$\hat{U}_1(\phi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}$$

```
def u1(state, phi):
    return np.array([[1, 0], [0, np.exp(1j * phi)]]) @ state
```

Легко видеть, что с точностью до глобального фазового множителя, который ни на что не влияет, Phase-shift-гейт – это тот же $\hat{R}Z(\phi)$. Он играет важную роль в квантовых ядерных методах.

23.3.4 Гейты \hat{U}_2 и \hat{U}_3

Более редкие в QML гейты, которые однако все равно встречаются в статьях.

$$\hat{U}_2(\phi, \lambda) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi+\lambda)} \end{bmatrix} = \hat{U}_1(\phi + \lambda) \hat{R}Z(-\lambda) \hat{R}Y\left(\frac{\pi}{2}\right) \hat{R}Z(\lambda)$$

Давайте убедимся в справедливости этого выражения:

```
def rz(state, phi):
    return np.array([[np.exp(-1j * phi / 2), 0], [0, np.exp(1j * phi / 2)]]) @ state

def u2_direct(phi, l):
    return (
        1
        / np.sqrt(2)
        * np.array([[1, -np.exp(1j * l)], [np.exp(1j * phi), np.exp(1j * (phi + l))]])
    )

def u2_inferenced(phi, l):
    return (
        u1(np.eye(2), phi + l)
        @ rz(np.eye(2), -l)
        @ ry(np.eye(2), np.pi / 2)
        @ rz(np.eye(2), l)
    )

print(np.allclose(u2_direct(np.pi / 6, np.pi / 3), u2_inferenced(np.pi / 6, np.pi / 3)))
```

True

Схожим образом определяется $\hat{U}_3(\theta, \phi, \lambda)$:

$$\hat{U}_3(\theta, \phi, \lambda) = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{1j\lambda} \sin\left(\frac{\theta}{2}\right) \\ e^{1j\phi} \sin\left(\frac{\theta}{2}\right) & e^{1j(\phi+\lambda)} \cos\left(\frac{\theta}{2}\right) \end{bmatrix} = \hat{U}_1(\phi + \lambda) \hat{R}Z(-\lambda) \hat{R}Y(\theta) \hat{R}Z(\lambda)$$

Читатель может сам легко убедиться, что эти формы записи эквивалентны. Для этого надо написать примерно такой же код, что мы писали раньше для \hat{U}_2 .

23.3.5 Еще пара слов об однокубитных гейтах

На этом мы завершаем обзор основных однокубитных гейтов. Маленько замечание: гейты, связанные со сдвигом фазы, никак не меняют состояние кубита, если оно сейчас 0. Так как мы всегда предполагаем, что начальное состояние кубитов – это именно 0, то перед применением, например, \hat{U}_1 , рекомендуется применить гейт Адамара:

```
print(np.allclose(u1(basis, np.pi / 6), basis))

h = 1 / np.sqrt(2) * np.array([[1 + 0j, 1 + 0j], [1 + 0j, 0j - 1]])
print(np.allclose(u1(h @ basis, np.pi / 6), h @ basis))
```

```
True
False
```

Единичный гейт

Самое последнее об однокубитных гейтах – это единичный гейт \hat{I} :

$$\hat{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

```
identity_gate = np.eye(2, dtype=np.complex128)
print(identity_gate)
```

```
[[1.+0.j 0.+0.j]
 [0.+0.j 1.+0.j]]
```

Он не делает с кубитом ровным счетом ничего. Но единичный гейт понадобится нам позже, когда мы будем конструировать многокубитные операторы.

23.4 Многокубитные состояния и гейты

Очевидно, что с одним кубитом ничего интересного, кроме разве что генератора истинно-случайных чисел, мы не сделаем. Для начала разберемся, как выглядят состояния для многокубитных систем.

23.4.1 Многокубитные состояния

В классическом компьютере один бит имеет два значения – 0 и 1. Два бита имеют четыре значения – 00, 01, 10, 11. Три бита имеют восемь значений и так далее. Аналогично состояние двух кубитов – это вектор в пространстве C^4 , состояние трех кубитов – вектор в пространстве C^8 , то есть состояние N кубитов описывается вектором размерности 2^N в комплексном пространстве. Вероятности каждой из возможных битовых строк (0000...00, 0000...01, 0000...10, и так далее) получаются по методу Шредингера, который мы обсуждали в конце прошлой лекции:

$$\mathbf{P}(\vec{s}) = |\Psi \vec{s}|^2$$

Нужно отсортировать наши битовые строки в лексикографическом порядке – и вероятность i -й битовой строки будет равна квадрату i -го элемента вектора Ψ .

Формально, многокубитные состояния описываются с помощью математического концепта так называемого *тензорного произведения*, которое в случае линейных операторов идентично *произведению Кронекера*, обозначаемого значком \otimes . Так, если $\Psi_A \in H_A$ и $\Psi_B \in H_B$, то $\Psi_{AB} = \Psi_A \otimes \Psi_B \in H_{AB} = H_A \otimes H_B$. О том, как элементы вектора Ψ_{AB} выражаются через элементы векторов Ψ_A и Ψ_B , можно прочитать на Википедии в статье “Произведение Кронекера”.

23.4.2 Многокубитные операторы

Как мы уже обсуждали ранее, квантовые операторы должны переводить текущее состояние в новое в том же пространстве и сохранять нормировку, а еще должны быть обратимыми. Значит, оператор для состояния из N кубитов – это унитарная комплексная матрица размерности $2^N \times 2^N$.

23.4.3 Конструирование многокубитных операторов

Прежде чем мы начнем обсуждать двухкубитные операторы, рассмотрим ситуацию. Представим, что у нас есть состояние из двух кубитов и мы хотим подействовать на первый кубит оператором Адамара. Как же тогда нам написать такой двухкубитный оператор? Мы знаем, что действуем на первый кубит оператором, а что происходит со вторым кубитом? Ничего не происходит – и это эквивалентно тому, что мы действуем на второй кубит единичным оператором. А финальный оператор $2^2 \times 2^2$ записывается через произведение Кронекера:

$$\hat{H} \otimes \hat{I} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} \hat{I} & \hat{I} \\ \hat{I} & -\hat{I} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

Учитывая, что многокубитные состояния конструируются аналогичным образом через произведение Кронекера, мы можем убедиться в верности нашего вывода:

```
print(np.allclose(np.kron(h @ basis, basis), np.kron(h, identity_gate) @ np.
    ↪kron(basis, basis)))
```

```
True
```

23.4.4 Наблюдаемые для многокубитных гейтов

Аналогичным образом можно сконструировать и наблюдаемые. Например, если мы хотим измерять одновременно два спина по оси **Z**, то наблюдаемая будет выглядеть так:

$$\mathbf{ZZ} = \hat{\sigma}^z \otimes \hat{\sigma}^z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
print(np.kron(basis, basis).conj().T @ np.kron(pauli_z, pauli_z) @ np.kron(basis, ↪basis))
```

```
[[1.+0.j]]
```

23.5 Основные двухкубитные гейты

Основные многокубитные гейты, которые предоставляют современные квантовые компьютеры, – это двухкубитные гейты.

23.5.1 CNOT (CX)

Квантовый гейт контролируемого инвертирования – это гейт, который действует на два кубита: *рабочий* и *контрольный*. В зависимости от того, имеет ли контрольный кубит значение 1 или 0, этот гейт инвертирует или не инвертирует рабочий кубит.

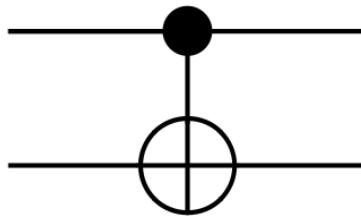


Fig. 23.3: Гейт CNOT

Иногда этот гейт также называют гейтом CX. В матричном виде этот оператор можно записать так:

$$\hat{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

```
cnot = (1 + 0j) * np.array(
    [
        [1, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 0, 1],
        [0, 0, 1, 0],
    ]
)

print(np.allclose(cnot @ np.kron(basis, basis), np.kron(basis, basis)))
print(np.allclose(
    cnot @ np.kron(pauli_x @ basis, basis), np.kron(pauli_x @ basis, pauli_x @ basis)
))
```

```
True
True
```

Заметьте, тут мы воспользовались тем, что $\hat{\sigma}^x$ работает так же, как инвертор кубитов: он превращает 0 в 1 и наоборот.

23.5.2 Гейты CY и CZ

Схожие по принципу гейты – это гейты \hat{CY} и \hat{CZ} . В зависимости от значения управляющего кубита к рабочему кубиту применяют соответствующий оператор Паули:

$$\hat{CY} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{bmatrix} \quad \hat{CZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

23.5.3 Гейт iSWAP

Гейты \hat{CX} , \hat{CY} и \hat{CZ} эквивалентны с точностью до однокубитных гейтов. Это означает, что любой из них можно получить, добавив необходимые однокубитные гейты до и после другого гейта. Например:

$$\hat{CZ} = (\hat{I} \times \hat{H}) \hat{CX} (\hat{I} \times \hat{H}).$$

Этим свойством обладают отнюдь не все двухкубитные гейты. Например, таковым является гейт iSWAP:

$$\text{iSWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & i & 0 \\ 0 & i & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

23.5.4 Гейт fSim

Для разных архитектур квантовых процессоров “естественный” гейт может выглядеть по-разному. Например, в квантовом процессоре Google Sycamore естественным является так называемый fermionic simulation gate или fSim. Это двухпараметрическое семейство гейтов вида:

$$\text{fSim}(\theta, \phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -i \sin \theta & 0 \\ 0 & -i \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & e^{-i\phi} \end{bmatrix}.$$

Впрочем, и fSim-гейт не является эквивалентным всему множеству двухкубитных гейтов. В общем случае, чем больше кубитов, тем сложнее будет выглядеть декомпозиция произвольного гейта на физически реализуемые в “железе”.

23.6 Первое знакомство с PennyLane

На сегодняшний день существует достаточно много фреймворков для программирования квантовых компьютеров. Для целей этого курса мы будем использовать [PennyLane](#). Эта библиотека предоставляет высокоуровневый Python API и создана специально для решения задач квантового машинного обучения.

```
import pennylane as qml
```

23.6.1 Device

Для объявления квантового устройства используется класс `Device`. PennyLane поддерживает работу с большинством существующих квантовых компьютеров, но для целей курса мы будем запускать все наши программы лишь на самом простом симуляторе идеального квантового компьютера:

Note: Разработка и отладка квантовых алгоритмов, как правило, происходит на симуляторах. Но надо понимать, что это работает только пока алгоритмы – “игрушечные” и задействуют пару-тройку или пару десятков кубитов. Надо понимать, что при добавлении каждого следующего кубита требуется вдвое больше ресурсов, чтобы просимулировать квантовый компьютер. Поэтому симуляторы –принципиально плохо масштабируемые аналоги реальных квантовых компьютеров.

```
device = qml.device("default.qubit", 2)
```

Первый аргумент тут – указание устройства, а второй – число кубитов.

23.6.2 QNode

Основной *строительный блок* в PennyLane – это `qnode`. Это функция, которая отмечена специальным декоратором и включает в себя несколько операций с кубитами. Результатом такой функции всегда является измерение. Напишем функцию, которая поворачивает первый кубит на 45° , после чего измеряет оба кубита по оси **Z**.

Сначала на NumPy

```
state = np.kron(basis, basis)
op = np.kron(ry(np.eye(2), np.deg2rad(45)), np.eye(2, dtype=np.complex128))
measure = np.kron(pauli_z, pauli_z)

print((op @ state).conj().T @ measure @ (op @ state))
```

```
[[0.70710678+0.j]]
```

Теперь через QNode

```
@qml.qnode(device)
def test(angle):
    qml.RY(angle, wires=0)
    return qml.expval(qml.PauliZ(0) @ qml.PauliZ(1))

print(test(np.deg2rad(45)))
```

```
0.7071067811865475
```

23.7 Заключение

Это последняя вводная лекция, где мы сами писали операторы и операции на чистом NumPy: это должно помочь лучше понять ту математику, которая лежит “под капотом” у квантовых вычислений. Дальше мы будем пользоваться только PennyLane и в отдельной лекции расскажем, как работать с этим фреймворком.

Итого:

- мы знаем, что такое кубит;
- понимаем линейную алгебру, которая описывает квантовые вычисления;
- понимаем, как можно сконструировать нужный нам оператор и как его применить;
- знаем, что такое измерение и наблюдаемые.

Теперь мы готовы к тому, чтобы знакомиться с квантовыми вариационными схемами и переходить непосредственно к построению моделей квантового машинного обучения.

23.8 Задачи

- Как связаны ось и угол вращения на сфере Блоха с собственными значениями и собственными векторами матрицы однокубитного гейта? Для этого найдите собственные векторы и собственные значения гейта $R^{\vec{n}(\alpha)}$.
- Вокруг какой оси и на какой угол вращает состояние гейт Адамара?
- Гейт SWAP меняет кубиты местами. Его унитарная матрица имеет вид:

$$\text{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Попробуйте составить последовательность гейтов, реализующую SWAP, из гейтов iSWAP, \hat{CZ} и $\hat{RZ}(\phi)$.

СМЕШАННЫЕ СОСТОЯНИЯ И МАТРИЦЫ ПЛОТНОСТИ

24.1 Описание лекции

Из этой лекции мы узнаем:

- Что такое матрица плотности
- Как ввести в описание квантовых состояний новый уровень случайности
- Как связаны смешанные (mixed) и запутанные (entangled) состояния
- Как можно описать насколько “сильно” квантовое состояние является смешанным

24.2 Смешанные состояния

Эта глава кратко рассматривает довольно сложный, но крайне важный аспект квантового описания мира: как описать квантовую систему, в которой случайным является не только результат измерения, но и само состояние квантовой системы, иными словами, когда мы не можем описать квантовую систему определенным вектором состояния.

Как могут появиться такие ситуации?

Прежде всего, в любой реалистичной ситуации квантовая система будет испытывать влияние окружающей среды. Это воздействие окружающей среды на квантовый компьютер, как правило, может быть охарактеризовано некоторой температурой этой самой среды и скоростью термализации системы с ней. В отличие от классических компьютеров, где шумы редко приводят к ошибкам в вычислениях, квантовые компьютеры очень плохо защищены от шумов. Любое масштабное квантовое вычисление является гонкой с процессами декогеренции и релаксации: за время выполнения алгоритма шумы не должны испортить результат вычислений настолько, чтобы его нельзя было использовать.

Вторая, более важная причина использования смешанных состояний состоит в их тесной связи с запутанными состояниями, как показано ниже.

24.3 Почему нельзя обойтись волновой функцией

Рассмотрим реальный физический кубит в состоянии равновесия, например, спин ядра атома в магнитном поле. Статистическая физика говорит нам, что вероятность обнаружить этот спин ориентированным вдоль внешнего поля выше, чем в противоположном полю направлении. Для расчета отношений этих вероятностей можно использовать распределение Больцмана: $p_{\uparrow}/p_{\downarrow} = e^{\frac{2\mu\Delta H}{k_B T}}$, где $2\mu\Delta H$ – разность энергий состояний со спином вдоль поля и в противоположном полю направлений, k_B – постоянная Больцмана, а T – температура. Так как состояний у этой системы всего два, сумма их вероятностей давать единицу: $p_{\uparrow} + p_{\downarrow} = 1$. Таким образом, вероятности состояний однозначно определены.

Волновую функцию, отвечающую такому тепловому состоянию, можно записать в виде

$$\psi = \begin{pmatrix} \sqrt{p_{\uparrow}} \\ \sqrt{p_{\downarrow}} e^{i\phi} \end{pmatrix},$$

где значение фазы ϕ не определено. Однако ϕ определяет поведение системы не в меньшей мере, чем вероятности p_{\uparrow} и p_{\downarrow} . ϕ может равновероятно принимать любые значения. Таким образом, уже даже состояние теплового равновесия нельзя описать одной волновой функцией – это будет распределением вероятности по волновым функциям с разным ϕ .

24.4 Матрица плотности

Оказывается, что вместо распределения вероятностей по волновым функциям можно использовать более простую конструкцию – матрицу плотности. Если система находится в состояниях Φ_n с вероятностями p_n , то матрицу плотности можно определить как

$$\rho = \sum_n p_n \Phi_n \Phi_n^*. \quad (24.1)$$

Выражение $\Phi_n \Phi_n^*$ обозначает произведение вектора-столбца на вектор-строку – результатом будет матрица. Важно, что значение любой ожидаемой величины (отвечающей оператору \hat{A}) можно записать через ρ :

$$\mathbb{E}[A] = \sum_n p_n \Phi_n \hat{A} \Phi_n^* = \sum_n p_n \text{Tr} [\Phi_n \Phi_n^* \hat{A}] = \text{Tr} [\rho \hat{A}]$$

Математическое обоснование этой циклической перестановки можно получить, расписав матричные произведения покомпонентно:

$$\Phi_n \hat{A} \Phi_n^* = \sum_{i,j} \Phi_n^{i*} A_{ij} \Phi_n^j = \text{Tr} [\Phi_n \Phi_n^* \hat{A}].$$

Любую наблюдаемую физическую величину можно выразить в виде ожидаемой величины некоторого эрмитова оператора – а значит, описание с помощью матрицы плотности является универсальным для любых случайных квантовых систем.

Стоит заметить, что матрица плотности для подсистем была впервые введена в научный оборот знаменитым советским физиком, лауреатом Нобелевской премии Львом Ландау. [15a]

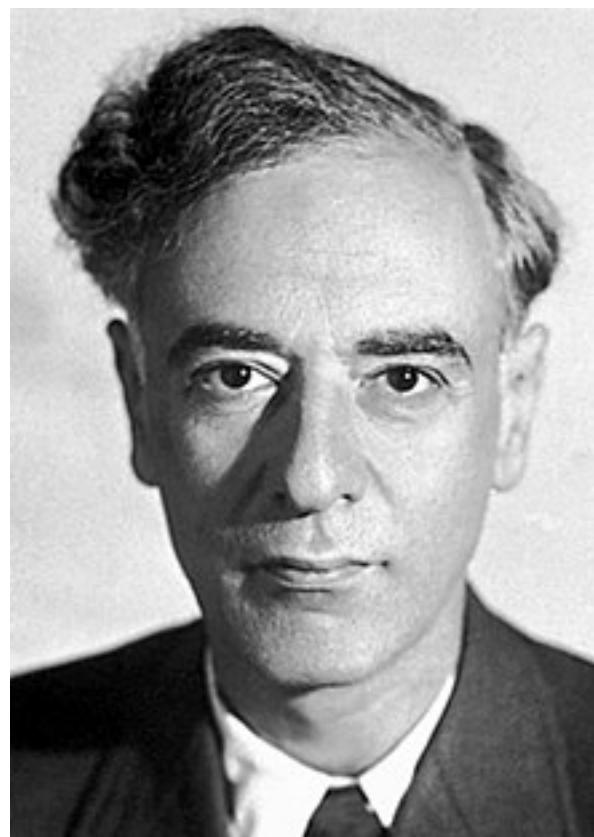


Fig. 24.1: Лев Ландау, 1908-1968

24.5 Чистые и смешанные состояния

Состояния, которые описываются одной единственной волновой функцией Ψ , называются *чистыми* (англ. *pure states*). Для таких состояний выражение для матрицы плотности получается тривиальным:

$$\rho = \Psi\Psi \quad (24.2)$$

Матрица плотности чистого состояния является *оператором-проектором*: действие оператора на волновую функцию произвольного состояния Φ дает проекцию Φ на Ψ . Состояния, которые нельзя описать одним вектором состояния, а можно лишь матрицей плотности, называются *смешанными* (англ. *mixed states*).

24.5.1 Чистота состояния

Можно легко показать, для оператора-проектора (24.2) выполняется тождество

$$\rho^2 = \Psi\Psi\Psi\Psi = \Psi\Psi = \rho$$

и следовательно, для чистого состояния

$$\text{Tr}(\rho^2) = 1$$

(напомним, что мы всегда считаем вектора состояния нормированными на единицу).

Аналогичным образом, но после более длинных выкладок можно показать, что в общем случае

$$\text{Tr}(\rho^2) \leq 1,$$

причем знак равенства в последней формуле возможен, только если в формуле (24.1) сумма имеет лишь одно слагаемое (т.е. состояние является чистым). Последнее свойство матрицы плотности позволяет ввести ряд величин, характеризующих смешанные и запутанные состояния, о чём будет рассказано в заключительном разделе этой лекции. Величина

$$\gamma = \text{Tr}(\rho^2) \quad (24.3)$$

называется *чистотой состояния* (*quantum state purity*).

24.5.2 Энтропия

Энтропия фон Неймана – это другая численная характеристика того, насколько сильно наша система смешанная. Её выражение очень похоже на выражение для классической *энтропии Шеннона*. Только в отличие от классики, в квантовой механике мы имеем *матрицу* плотности, поэтому в выражении у нас фигурирует *матричный логарифм*:

$$S = -\text{Tr}(\rho \cdot \ln(\rho)) \quad (24.4)$$

24.5.3 Спектральная декомпозиция матрицы плотности

Определение матрицы плотности (24.1) представляет собой сумму матриц плотностей чистых состояний, взятых с некоторыми вероятностями. Интересно, что совершенно разным комбинациям чистых состояний могут соответствовать одинаковые матрицы плотности. Например, состояние кубита, который с вероятностью

50% находится в состоянии 0 и с вероятностью 50% в состоянии 1 совершенно неотличимо от такой же равновероятной смеси состояний + и -:

$$\frac{1}{2} (00 + 11) = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} = \frac{1}{2} (+ + + --)$$

Соотношение (24.1) также можно рассматривать как спектральную декомпозицию матрицы плотности. В этом случае состояния Ψ_n образуют ортонормированный базис, а вероятности p_n – это собственные значения ρ . Как чистота (24.3), так и энтропия (24.4) зависят лишь от этих собственных значений.

24.6 Смешанные состояния и запутанность

Рассмотрим ситуацию, когда описываемую квантовую систему Φ можно разделить на две подсистемы, ϕ и ψ , и состояние этой системы Φ является суперпозицией состояний двух подсистем:

$$\Phi = \sum_{i,j} c_{i,j} \phi_i \psi_j.$$

Здесь два ортонормированных набора векторов состояния ϕ_i и ψ_j описывают две части всей системы. Для такого состояния не всегда можно сказать, в каком именно состоянии находится каждая подсистема.

Note: Вспомним обсуждение кота Шредингера в [лекции про кубит](#) – упрощая до предела, можно считать, что радиоактивный атом является одной системой, а несчастный кот – второй.

До измерения ни атом, ни кот не имеют определенного состояния, а находятся в суперпозиции возможных состояний.

Что же мы можем сделать, если у нас есть доступ лишь к одной из двух подсистем, а измерить состояние второй мы уже не можем? В таком случае, все наши наблюдаемые величины будут отвечать операторами A_ψ , которые действует *только* на вторую подсистему. В примере кота Шредингера можно допустить, что экспериментально пронаблюдать мы можем лишь состояние кота, а состояние радиоактивного атома недоступно нам для измерения. В таком случае состояние второй подсистемы можно полностью описать используя *редуцированную* матрицу плотности. Редуцированная матрица плотности получается из матрицы плотности чистого состояния всей системы суммированием по вероятностям различных состояний первой подсистемы:

$$\rho_\psi = \text{Tr}_\phi(\Phi\Phi) = \sum_{i,j} d_{i,j} \psi_j \psi_i, \quad (24.5)$$

где

$$d_{i,j} = \sum_k c_{j,k} c_{i,k}^*,$$

а Tr_ϕ означает частичный след по второй подсистеме, звездочка – комплексное сопряжение. В таких обозначениях значение для среднего от оператора A вычисляется по формуле

$$A_\psi = \text{Tr}_\psi(A\rho_\psi)$$

Здесь след матрицы уже вычисляется по первой подсистеме.

Чаще всего при обсуждении смешанных состояний рассматривают только одну “подсистему”, считая, что вторая – это некоторый макроскопический объект (“резервуар”, например лаборатория или даже вся Вселенная). В этом случае определение матрицы плотности (24.1) можно рассматривать как редуцированную матрицу плотности (24.5), из которой убрали нерелевантные и неконтролируемые степени свободы.

24.6.1 Запутанные и сепарабельные состояния

Давайте вернемся к представлению состояния составной системы и зададимся вопросом: что можно сказать о связи между частями системы с точки зрения квантового описания? Для системы из двух кубитов такая составная система в общем случае может быть записана в явном виде (в этом разделе мы в основном следуем изложению из книги [01]):

$$\Phi = a_0|0_B\rangle\langle 0_B| + b_0|1_B\rangle\langle 1_B| + c_1|0_B\rangle\langle 1_A| + d_1|1_B\rangle\langle 0_A|, \quad (24.6)$$

где индексы A и B здесь обозначают первый и второй кубиты, соответственно, а условие нормировки дает

$$\langle \Phi | \Phi \rangle = |a|^2 + |b|^2 + |c|^2 + |d|^2 = 1.$$

Теперь, как можно показать, состояние типа (24.6) может быть представлено в виде произведения состояний двух отдельных кубитов, если $ad = dc$:

$$\Phi_s = (a_A|0_A\rangle\langle 0_A| + b_A|1_A\rangle\langle 1_A|) \otimes (a_B|0_B\rangle\langle 0_B| + b_B|1_B\rangle\langle 1_B|),$$

где для выражения (24.6)

$$\begin{aligned} a &= a_A a_B & b &= a_A b_B \\ c &= b_A a_B & d &= b_A b_B \end{aligned}$$

В других случаях, когда $ad \neq dc$, состояние составной системы не представимо в виде произведения состояний подсистем, такие состояния называют несепарабельными (nonseparable). Другими словами, результат измерения состояния подсистемы A будет зависеть от состояния подсистемы B . Это означает, что для квантовых систем возможна нелокальная корреляция. Такое свойство квантовых систем называется *запутыванием* (*entanglement*), а сами состояния *запутанными* (*entangled*).

Note: В отличие от английского языка, в русском языке не сложилось единой терминологии в отношении запутанных состояний. На момент написания этой лекции (осень 2021 года) в статье [Квантовая запутанность](#) русскоязычной Википедии указывается восемь (!) отличающихся терминов для этого явления, например, “запутанность”, “перепутанность” или “сцепленность”. Сделать с этим что-то сложно, остается только иметь в виду имеющиеся обстоятельства. Мы будем стараться употреблять термин *запутанность* и, соответственно, *запутанные* состояния.

Приведем пару примеров запутанных состояний:

1. $b = c = 0, a = d = \pm 1/\sqrt{2}$ – состояние “шредингеровского кота”, см. [01] и [07], такая формула для вектора состояния возникает для суперпозиции двух макроскопически различных состояний одной из подсистем, например, живой или мертвый кот.
2. $a = d = 0, b = -c = \pm 1/\sqrt{2}$ – такое состояние называется ЭПР-парой (EPR, от Einstein-Podolsky-Rosen) и это очень важный пример из истории изучения запутанности в квантовой физике.

Note: в 1930-е происходили многочисленные споры об “интерпретации” (сущности) квантовой механики. Именно тогда Эйнштейн, Шредингер и их коллеги обратили внимание на несепарабельные состояния и затем Эйнштейном, Подольским и Розеном был сформулирован “парадокс” – что квантовая механика либо нелокальна (т.е. несовместима с теорией относительности), либо неполна (мы учтем не все параметры при описании состояния квантовых систем). Именно с дискуссией о сути запутанности связана знаменитая цитата Эйнштейна “Бог не играет в кости” и менее известный ответ Нильса Бора, “Альберт, не указывай Богу, что ему делать.”

Довольно долго изучение запутанности и связанных с ней трудностей считались сложным, но не основными вопросами квантовой физики. Но с развитием квантовой информатики стало понятно, что без запутанности

нельзя разрабатывать квантовые компьютеры и системы квантовой связи. В настоящее время существуют устоявшиеся методы создания запутанных состояний в эксперименте. А для целей нашего курса, в симуляциях, достаточно использовать [двукубитные гейты](#), которые обсуждались в предыдущей лекции, например CNOT или CZ, который используется в лекции про [Градиенты квантовых схем](#).

В качестве примера давайте посмотрим, как можно создать запутанное состояние в PennyLane. Начнем с импортов и создания двухкубитной схемы:

```
import pennylane as qml
from pennylane import numpy as np

dev = qml.device("default.qubit", wires=2)
```

Далее применим к первому кубиту операцию поворота $\hat{R}X$, запутаем кубиты с помощью $C\hat{N}OT$ и далее оценим запутанность с помощью измерения оператора Паули $\hat{\sigma}^z$:

```
@qml.qnode(dev)
def circuit(param):
    qml.RX(param, wires=0)
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.PauliZ(0)), qml.expval(qml.PauliZ(1))
```

В этом примере значение переменной `param` определяет степень запутанности, и для $\pi/2$ запутанности будет максимальна. В результате оба кубита будут максимально смешанными и средний результат измерения будет нулевым:

```
print(circuit(np.pi / 2))
```

```
[2.22044605e-16 2.22044605e-16]
```

(код взят из официальной [демонстрации](#) для библиотеки PennyLane)

24.6.2 Разложение Шмидта

В этом разделе мы познакомимся с важной процедурой – разложением Шмидта, которое тесно связано со спектром редуцированных матриц плотности состояния составной квантовой системы и благодаря которому легко увидеть, является ли состояние системы *запутанным* или нет.

Снова запишем чистое двухчастичное состояние Φ квантовой системы, в пространстве $H_\phi \otimes H_\psi$ двух подсистем ϕ и ψ . Покажем тогда, что *существуют* ортонормированные состояния i_ϕ системы ϕ и ортонормированные состояния i_ψ системы ψ , которые дадут нам разложение:

$$\Phi = \sum_i \lambda_i i_\phi i_\psi, \quad (24.7)$$

где λ_i – неотрицательные числа (коэффициенты Шмидта), удовлетворяющие условию $\sum_i \lambda_i^2 = 1$.

Для доказательства рассмотрим случай, когда обе подсистемы имеют пространство одинаковой размерности. Пусть тогда n и k состояния образуют произвольный ортонормированный базис для подсистем ϕ и ψ . Соответственно, тогда состояние системы Φ может быть представлено в виде разложения:

$$\Phi = \sum_{n,k} a_{nk} n k, \text{ где } a_{nk} = n k \Phi \quad (24.8)$$

Константы разложения a_{nk} образуют эрмитово-сопряженную комплексную матрицу A , которую можно привести к диагональному виду. Для этого применим к этой матрице сингулярное разложение (или SVD-разложение) вида: $A = U \cdot S \cdot V^*$, где U и V – унитарные матрицы, а S – диагональная матрица с неотрицательными

действительными числами на диагонали (эти числа называют сингулярными числами матрицы A , и их набор однозначно определяется матрицей). Тогда разложение (24.8) можно привести к виду:

$$\Phi = \sum_{i,n,k} u_{in} s_{ii} v_{ik} n k. \quad (24.9)$$

Теперь переопределим базис состояний в подсистемах ϕ и ψ :

$$i_\phi = \sum_n u_{ni} n, i_\psi = \sum_k v_{ik} k \quad (24.10)$$

и обозначим $s_{ii} \equiv \lambda_i$. В результате разложение (24.9) преобразуется к виду:

$$\Phi = \sum_i \lambda_i i_\phi i_\psi$$

В силу унитарности U и V наборы базисных состояний i_ϕ и i_ψ в (24.10) образуют полную ортонормированную систему, или базис Шмидта, а само представление (24.7) называют **разложением Шмидта**. Число ненулевых значений коэффициентов Шмидта λ_i называется числом (или рангом $\text{rank}(A) = \dim s_{ii} : s_{ii} > 0$) Шмидта для состояния Φ . В теории мер квантовой запутанности это число характеризует степень (или меру) запутанности состояний сложной системы. Чистое двухчастичное состояние запутанно тогда и только тогда, когда его число Шмидта > 1 , и чем больше число Шмидта, тем сильнее запутано состояние.

Следствием приведенных выше свойств является важная связь между коэффициентами Шмидта чистого запутанного состояния со спектром его редуцированных матриц плотности $\rho_\phi = \text{Tr}_\psi(\Phi\Phi)$ и $\rho_\psi = \text{Tr}_\phi(\Phi\Phi)$. Несложно убедиться, что собственные значения редуцированных матриц ρ_ϕ и ρ_ψ совпадают и представляют собой квадраты коэффициентов Шмидта, а их собственные вектора представляют собой состояния i_ϕ и i_ψ соответственно. Эти свойства дают нам удобный алгоритм вычисления разложения Шмидта двухчастичного состояния Φ через редуцированные матрицы его подсистем: (1) на первом этапе следует вычислить редуцированные матрицы плотности ρ_ϕ и ρ_ψ ; (2) на втором этапе найти общие собственные значения a_i и соответствующие им собственные векторы i_ϕ и i_ψ для матриц ρ_ϕ и ρ_ψ ; (3) записать разложение Шмидта в виде:

$$\Phi = \sum_i \sqrt{a_i} i_\phi i_\psi.$$

24.7 Описания эволюции смешанного состояния

24.7.1 Квантовая динамика

Напомним, что квантовая динамика в терминах волновых функций Ψ описывается при помощи уравнения Шредингера:

$$i\hbar \frac{\partial \Psi(x, t)}{\partial t} = \hat{H}\Psi(x, t).$$

Аналогичное уравнение можно получить и для матриц плотности. Оно называется уравнением фон Неймана и записывается через коммутатор $[\cdot]$, который определен как $[\hat{A}, \hat{B}] = \hat{A}\hat{B} - \hat{B}\hat{A}$:

$$i\hbar \frac{\partial \rho}{\partial t} = [\hat{H}, \rho].$$

Аналогично, если действие каких-то унитарных операций изменяет вектор состояния Ψ на $\hat{U}\Psi$, то матрицу плотности оно должно изменять как

$$\sum_n p_n \hat{U}\Psi_n \Psi_n^\dagger \hat{U}^\dagger = \hat{U}\rho\hat{U}^\dagger.$$

Важное свойство унитарных матриц – их собственные значения по модулю равны единице. Действие унитарного оператора не изменяет собственных значений матрицы плотности, но вращает собственный ее базис. Исходя из этого можно сделать вывод о том, что ни чистота, ни энтропия не могут изменяться в результате унитарных операций.

24.7.2 Измерения и томография

Квантовая механика работает так, что любое измерение приводит к коллапсу волновой функции и является необратимым. А еще измерения, например, состояния + и - не различимы при измерениях по оси **Z** – для обоих состояний мы будем получать 1 и 0 с вероятностью 0.5, то есть нам нужно измерять по *всем* базисам. В общем, получается, что восстановить амплитуду и фазу волновой функции Ψ это большая проблема, если добавить сюда вероятностный характер измерения.

Note: Строго говоря это не просто “большая” проблема, а настоящая NP-полная задача оптимизации!

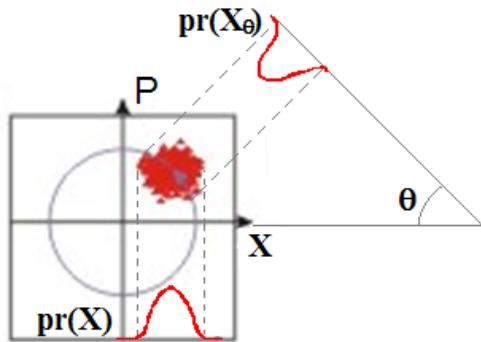


Fig. 24.2: Иллюстрация фазовой проблемы.

Эта задача обычно решается при помощи квантовой томографии, и восстанавливают как раз не волновую функцию Ψ , а матрицу плотности ρ (потому что в реальных экспериментах и задачах почти не бывает чистых состояний). Представим, что наша квантовая система описывается базисом y_i – набором из 2^N векторов, причем каждому из этих базисных векторов соответствует свое собственное значение – результат измерения (подробнее об этом было в [лекции про кубит](#)). Тогда если у нас будет достаточно много результатов измерений, то мы сможем восстановить нашу матрицу плотности ρ методом максимизации правдоподобия. Выражение для правдоподобия в этом случае можно записать как:

$$L(\rho) = \prod_i y_i \rho y_i^{q_i},$$

где q_i – это частота получения собственного значения, соответствующего волновой функции y_i (потому что измерение переводит наше состояние в базисный вектор, соответствующий результату измерения). В итоге увеличивая число измерений мы приближаем частоты q_i к вероятностям $\Psi(y_i)y_i\Psi$, а нашу матрицу ρ к ее истинному виду.

Методы квантовой томографии являются критической частью, в том числе, квантовой связи, так как системы там обычно небольшие, но восстанавливать надо всю матрицу плотности.

24.8 Что мы узнали?

- Формализм матрицы плотности позволяет описывать составные системы (например, один кубит в многокубитной системе)
- Чаще всего в реальных экспериментах у нас ситуация “мы приготовили состояние, но точно не знаем какое” и волновая функция нам не подходит
- Что такое квантовая запутанность и как ее можно описать

- Состояние части запутанного состояния – смешанное
- Отличие смешанного состояния от чистого можно охарактеризовать параметром типа энтропии

CHAPTER
TWENTYFIVE

О БЛОКЕ “ФРЕЙМВОРКИ”

Этот блок включает в себя обзор фреймворков и библиотек для квантовых вычислений. Основная часть курса будет строиться вокруг библиотеки PennyLane. Этот фреймворк кажется наиболее простым в освоении, а также является платформо-независимым, так как представляет собой высокоуровневый API. Дополнительные лекции этого блока расскажут также про:

- Qiskit от компании IBM;
- cirq и Tensorflow Quantum от компании Google.

ОБЗОР ФРЕЙМВОРКОВ ДЛЯ КВАНТОВЫХ ВЫЧИСЛЕНИЙ

В этом обзоре мы кратко пробежимся по существующим сегодня фреймворкам для программирования квантовых компьютеров, а также по тем компьютерам, с которыми они совместимы.

Note: Цель этой лекции – исключительно показать как визуально выглядит тот, или иной фреймворт для программирования квантовых компьютеров. Поэтому не стоит пытаться вникнуть и понять, что именно делает тот, или иной пример приведенного тут кода! Также хочется сразу сказать, что разделение на низкоуровневые и высокоуровневые языки квантового относительно субъективно и эта лекция выражает именно точку зрения ее автора.

26.1 Квантовые ассемблеры

26.1.1 OpenQASM

В нашем курсе мы в основном будем пользоваться относительно высокоуровневые фреймворки. Однако это не значит, что квантовый программист не имеет возможности программировать на низком уровне. Например, компания IBM дает доступ к своему квантовому ассемблеру OpenQASM [CBSG17], код на котором выглядит примерно так:

```
/* Measuring the relaxation time of a qubit
 * This example demonstrates the repeated use of fixed delays.
 */
OPENQASM 3.0;
include "stdgates.inc";

duration stride = 1us;           // time resolution of points taken
const int[32] points = 50;        // number of points taken
const int[32] shots = 1000;        // how many shots per point

int[32] counts0;
int[32] counts1 = 0;    // surviving |1> populations of qubits

extern tabulate(int[32], int[32], int[32]);

bit c0;
bit c1;

defcalgrammar "openpulse";

// define a gate calibration for an X gate on any qubit
```

(continues on next page)

(continued from previous page)

```

defcal x $q {
    play drive($q), gaussian(100, 30, 5);
}

for p in [0 : points-1] {
    for i in [1 : shots] {
        // start of a basic block
        reset $0;
        reset $1;
        // excite qubits
        x $0;
        x $1;
        // wait for a fixed time indicated by loop counter
        delay[p * stride] $0;
        // wait for a fixed time indicated by loop counters
        delay[p * durationof({x $1;})];
        // read out qubit states
        c0 = measure $0;
        c1 = measure $1;
        // increment counts memories, if a 1 is seen
        counts0 += int[1](c0);
        counts1 += int[1](c1);
    }
    // log survival probability curve
    tabulate(counts0, shots, p);
    tabulate(counts1, shots, p);
}

```

Пример этого кода взят из официального репозитория.

26.1.2 PyQuil

Другим примером относительно низкоуровневого языка квантового программирования является фреймворк PyQuil [SCZ17]. Он разработан компанией Rigetti Computing, производителем одноименных квантовых компьютеров и включает в себя компилятор и виртуальную машину QVM. Вот так выглядит код на этом фреймворке:

```

from pyquil import get_qc, Program
from pyquil.gates import CNOT, H, MEASURE

qvm = get_qc('2q-qvm')

p = Program()
p += H(0)
p += CNOT(0, 1)
ro = p.declare('ro', 'BIT', 2)
p += MEASURE(0, ro[0])
p += MEASURE(1, ro[1])
p.wrap_in_numshots_loop(10)

qvm.run(p).readout_data['ro'].tolist()

```

Пример взят из официального репозитория.

26.2 Высокоуровневое программирование

В таких фреймворках программист имеет возможность уже не только напрямую манипулировать кубитами на низком уровне, но также вызывать высокоуровневые функции, например, квантовое преобразование *Фурье*.

26.2.1 Cirq

Одним из таких фреймворков является *Cirq* от компании Google. Код, написанный на нем можно запускать на симуляторах локально или в облаке, а также используя специально разработанные компанией Google симуляторы на базе Tensor Processing Units или TPU используя Floq API. Для целей квантового машинного обучения в связке с *Cirq* можно использовать Tensorflow Quantum – расширение библиотеки для обычного Deep Learning, которое позволяет комбинировать классические и квантовые нейронные сети вместе. Еще в экосистему *Cirq* входит также библиотека для квантовой химии *OpenFermion*.

Вот так выглядит код, написанный на *Cirq*:

```
import cirq
qubit = cirq.GridQubit(0, 0)

circuit = cirq.Circuit()
circuit.append(cirq.H(qubit))
circuit.append(cirq.measure(qubit))
print(circuit)
```

Еще больше примеров и более детальный обзор этого фреймворка читайте в [нашей статье про Tensorflow Quantum](#). Также очень рекомендуем ознакомиться с блогом “квантового” подразделения компании Google – *Google Quantum AI*.

26.2.2 Q#

Q# это полноценный язык квантового программирования от компании Microsoft. Этот техногигант также, как и другие, разрабатывает сейчас свои квантовые компьютеры и облачные симуляторы, а также предоставляет облачный доступ к квантовому железу других производителей. Так что программы, написанные на *Q#* можно запускать в облаке *Azure Quantum*.

В отличии от большинства фреймворков, которые обычно написаны как библиотека для языка Python, *Q#* использует C# для запуска и в целом имеет синтаксис, больше похожий на C# или Java:

```
namespace QuantumRNG {
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Measurement;
    open Microsoft.Quantum.Canon;

    @EntryPoint()
    operation GenerateRandomBits() : Result[] {
        use qubits = Qubit[4];
        ApplyToEach(H, qubits);
        return MultiM(qubits);
    }
}
```

Этот пример взят из [официальной документации](#).

На самом деле *Q#* это не просто язык для программирования квантовых компьютеров, а целая экосистема, в которую кроме самого языка и его стандартной библиотеки также входят:

- Библиотека `Microsoft.Quantum.Chemistry` для квантовой химии
- Библиотека `Microsoft.Quantum.MachineLearning` для квантового машинного обучения

В качестве отличного источника информации про квантовые вычисления с примерами на Q#-стеке можно использовать потрясающий репозиторий на GitHub, который называется [QuantumKatas](#).

26.2.3 Strawberry-Fields

`Strawberry-Fields` это библиотека для работы с фотонными квантовыми компьютерами компании Xanadu. Эти устройства являются *continuous-variable quantum computers* и в их основе лежат не бинарные состояния – кубиты, а состояния с непрерывными значениями. Учитывая сложность этого концепта, а также специфичность решаемых на таких устройствах задач мы мало будем о них говорить, но любопытный читатель может ознакомиться, например, с [этой статьей](#).

Вот так выглядит код, написанный с использованием этого фреймворка (пример из [официальной документации](#)):

```
import strawberryfields as sf
from strawberryfields import ops

# create a 3-mode quantum program
prog = sf.Program(3)

with prog.context as q:
    ops.Sgate(0.54) | q[0]
    ops.Sgate(0.54) | q[1]
    ops.Sgate(0.54) | q[2]
    ops.BSgate(0.43, 0.1) | (q[0], q[2])
    ops.BSgate(0.43, 0.1) | (q[1], q[2])
    ops.MeasureFock() | q
```

Также к `Strawberry-Fields` идет большое число высокогенерируемых API для конкретных приложений к задачам реального мира. Например:

- Вычисление схожести графов
- Gaussian Boson Sampling
- Поиск Clique в графах
- Поиск плотных подграфов

И многие другие, с которыми можно ознакомиться на странице официальной документации.

26.2.4 Qiskit

`Qiskit` – это один из самых популярных в мире фреймворков для квантовых вычислений от компании IBM. На самом деле представляет собой даже не фреймворк, а целую экосистему, в которую входят, например:

- Уже [упомянутый нами](#) квантовый ассемблер OpenQASM
- Фреймворк для программирования кубитов, написанный на OpenQASM
- API для применения квантовых вычислений в финансах
- API для применения квантовых вычислений в ML
- API для применения квантовых вычислений в оптимизации
- API для применения квантовых вычислений в квантовой химии

- Открытый CAD для проектирования квантовых компьютеров (!)

И многое другое.

Код, написанный на Qiskit выглядит так (пример из официальной документации):

```
import numpy as np
from qiskit import QuantumCircuit, transpile
from qiskit.providers.aer import QasmSimulator
from qiskit.visualization import plot_histogram

# Use Aer's qasm_simulator
simulator = QasmSimulator()

# Create a Quantum Circuit acting on the q register
circuit = QuantumCircuit(2, 2)

# Add a H gate on qubit 0
circuit.h(0)

# Add a CX (CNOT) gate on control qubit 0 and target qubit 1
circuit.cx(0, 1)

# Map the quantum measurement to the classical bits
circuit.measure([0,1], [0,1])

# compile the circuit down to low-level QASM instructions
# supported by the backend (not needed for simple circuits)
compiled_circuit = transpile(circuit, simulator)

# Execute the circuit on the qasm simulator
job = simulator.run(compiled_circuit, shots=1000)

# Grab results from the job
result = job.result()

# Returns counts
counts = result.get_counts(compiled_circuit)
print("\nTotal count for 00 and 11 are:",counts)

# Draw the circuit
circuit.draw()
```

Код, написанный на Qiskit может быть запущен на квантовых компьютерах от компании IBM. Для изучения квантовых вычислений с экосистемой Qiskit можно рекомендовать прекрасную онлайн книгу. Более подробный обзор библиотеки также в *отдельной лекции нашего курса*.

26.2.5 Pennylane

Pennylane это относительно высокоуровневая библиотека для квантовых вычислений и квантового машинного обучения от компании Xanadu. Вот так выглядит написанный на ней код:

```
import pennylane as qml
from pennylane import numpy as np

dev = qml.device('default.qubit', shots=1000, wires=2)
```

(continues on next page)

(continued from previous page)

```
def make_entanglement():
    qml.Hadamard(wires=0)
    qml.CNOT(wires=[0, 1])
    return qml.probs(wires=[0, 1])

circuit = qml.QNode(make_entanglement, dev)
circuit()
```

Эта библиотека может использовать в качестве backend любое из:

- Cirq
- Qiskit
- Q#
- Strawberry-Fields
- PyQuil (через расширение [Pennylane-Forest](#))

Но для нас самое главное преимущество этой библиотеки – это огромное количество готовых API для машинного обучения, которые позволяют автоматически считать градиенты и дифференцировать квантовые схемы, или обновлять параметры таких схем градиентным спуском.

В нашем курсе библиотеке Pennylane посвящена [целая отдельная лекция](#).

26.3 Заключение

26.3.1 Что мы будем использовать в курсе?

В рамках нашего курса мы в основном будем пользоваться библиотекой Pennylane. Некоторые отдельные лекции будут использовать Qiskit, но это скорее в порядке исключения.

26.3.2 Почему Pennylane?

Но почему Pennylane, если самым популярным фреймворком является именно Qiskit? Приведем два основных аргумента.

1. Сегодня трудно сказать, какое квантовое железо в итоге победит, а большим преимуществом Pennylane является то, что эта библиотека является верхнеуровневой и может использовать различные backend для запуска на квантовых компьютерах разных производителей. Включая Qiskit и IBM.
2. Наш курс больше про квантовое машинное обучение и именно для этих целей Pennylane подходит лучше всего. Там есть рутины и API разных уровней, позволяющих где-то самому писать градиенты квантовых схем и обновлять их параметры, а где-то просто вызывать готовый метод из API, чтобы не повторять 10 раз то, что мы уже разбирали в более ранних лекциях.

CHAPTER
TWENTYSEVEN

PENNYLANE

PennyLane – библиотека Python для квантового машинного обучения, которую можно использовать для обычных квантовых вычислений. Программы, написанные на PennyLane, можно запускать, используя в качестве бэкенда настоящие квантовые компьютеры от IBM Q, Xanadu, Rigetti и другие, либо квантовые симуляторы.

Кубиты в PennyLane называются по-особому – wires (от англ. wires – провода). Такое название, скорее всего, связано с тем, что на квантовых схемах кубиты изображаются в виде продольных линий.

Последовательность квантовых операций называется *квантовой функцией*. Такая функция может принимать в качестве аргументов только хэшируемые объекты. В качестве возвращаемого значения выступают величины, связанные с результатами измерения: ожидаемое значение, вероятности состояний или результаты сэмплирования.

Квантовая функция существует не сама по себе, она запускается на определенном устройстве – симуляторе либо настоящем квантовом компьютере. Такое устройство в PennyLane называется device.

27.1 QNode

Квантовые вычисления при использовании PennyLane раскладываются на отдельные узлы, которые называются QNode. Для их создания используются квантовые функции совместно с device.

Создавать объекты квантовых узлов можно двумя способами: явно либо с помощью декоратора QNode.

Рассмотрим первый способ – явное создание узла.

```
import pennylane as qml
from pennylane import numpy as np
dev = qml.device('default.qubit', shots=1000, wires=2)
def make_entanglement():
    qml.Hadamard(wires=0)
    qml.CNOT(wires=[0, 1])
    return qml.probs(wires=[0, 1])

circuit = qml.QNode(make_entanglement, dev)
circuit()
```

```
tensor([0.484, 0.      , 0.      , 0.516], requires_grad=False)
```

Работая с библиотекой PennyLane для математических операций, можно использовать интерфейс NumPy, но при этом также пользоваться преимуществами автоматического дифференцирования, которое обеспечивает autograd. Именно поэтому мы не импортировали NumPy обычным способом: `import numpy as np`, а сделали это так: `from pennylane import numpy as np`.

Второй способ создания квантовых узлов – с помощью декоратора `QNode`. Пропускаем импорт библиотек и создание устройства, так как в начале код тот же самый:

```
@qml.qnode(dev)
def circuit():
    qml.Hadamard(wires=0)
    qml.CNOT(wires=[0, 1])
    return qml.probs(wires=[0, 1])

print(circuit())
```

```
[0.496 0.     0.504]
```

В данном примере мы взяли двухкубитную систему и создали запутанное состояние, а затем с помощью метода `probs` вычислили вероятности получения состояний $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$.

27.2 Операторы

В квантовой функции можно применять операторы X, Y, Z, S, T (`qml.PauliX`, `qml.PauliY`, `qml.PauliZ`, `qml.S`, `qml.T` соответственно), а также операторы, в которых можно задавать угол вращения вокруг одной из осей в радианах: `qml.RX`, `qml.RY`, `qml.RZ`. Здесь и далее будем использовать `qml` как псевдоним библиотеки `PennyLane`.

В этой функции мы вращаем кубит под индексом 0 вокруг оси X на 90 градусов из начального состояния $|0\rangle$ и возвращаем **ожидаемое значение** `qml.PauliZ` для этого кубита с помощью `qml.expval`. Вероятности получения состояний $|0\rangle$ и $|1\rangle$ равны, так что мы получаем ожидаемое значение, близкое к нулю, что легко проверить:

$$0.5 \cdot 1 + 0.5 \cdot (-1) = 0$$

```
@qml.qnode(dev)
def circuit(x):
    qml.RX(x, wires=0)
    return qml.expval(qml.PauliZ(0))

circuit(np.pi/2)
```

```
tensor(0.058, requires_grad=True)
```

В следующем примере мы вращаем кубит на тот же угол 90 градусов, но уже вокруг оси Y. Ожидаемое значение в этот раз ищем для `qml.PauliX` и получаем 1, что соответствует вычислениям:

$$1 \cdot 1 + 0 \cdot (-1) = 1$$

```
@qml.qnode(dev)
def circuit(x):
    qml.RY(x, wires=0)
    return qml.expval(qml.PauliX(0))

circuit(np.pi/2)
```

```
tensor(1., requires_grad=True)
```

В начале этого урока мы создали устройство, которое создает и запускает одну и ту же схему 1000 раз, каждый раз производя измерения. Поменяем этот параметр:

```
dev.shots = 5
```

Посмотрим на результат каждого из этих пяти запусков и измерений для `qml.PauliZ`. Квантовая схема будет простой: применим к кубиту с индексом 1 оператор Адамара:

```
@qml.qnode(dev)
def circuit():
    qml.Hadamard(wires=1)
    return qml.sample(qml.PauliZ([1]))

circuit()
```

```
array([ 1,  1, -1,  1, -1])
```

Мы получаем разные результаты: то 1, что соответствует состоянию $|0\rangle$, то -1, что соответствует состоянию $|1\rangle$.

Если вместо `qml.PauliZ` брать сэмплы для `qml.PauliX`, то результат все время будет один и тот же: 1, что соответствует состоянию $|+\rangle$ (вектор базиса Адамара).

```
@qml.qnode(dev)
def circuit():
    qml.Hadamard(wires=1)
    return qml.sample(qml.PauliX([1]))

circuit()
```

```
array([1, 1, 1, 1, 1])
```

27.3 QubitUnitary

В PennyLane можно использовать готовые операторы, либо задавать операторы явно с помощью матриц.

Для этого можно использовать класс `qml.QubitUnitary`, который принимает два параметра: `U` – квадратную унитарную матрицу и `wires` – кубиты, на которые действует оператор `U`.

В качестве примера создадим оператор, осуществляющий обмен состояний между кубитами (SWAP). Такой оператор уже есть в библиотеке PennyLane (`qml.SWAP`), но мы создадим его с помощью `qml.QubitUnitary`. Сначала мы зададим саму матрицу в виде двумерного массива, используя интерфейс NumPy:

```
U = np.array([[1, 0, 0, 0],
              [0, 0, 1, 0],
              [0, 1, 0, 0],
              [0, 0, 0, 1]])
```

Создадим заново устройство, при этом зададим число запусков схемы как `shots=1`: чтобы убедиться, что все работает правильно, нам будет достаточно одного запуска.

```
dev = qml.device('default.qubit', shots=1, wires=2)
```

Создадим и запустим схему, в которой перед применением операции SWAP, реализованной с помощью `qml.QubitUnitary`, один кубит будет находиться в состоянии 1, а другой – в состоянии 0.

```
@qml.qnode(dev)
def circuit(do_swap):
    qml.PauliX(wires=0)
    if do_swap:
        qml.QubitUnitary(U, wires=[0, 1])
    return qml.sample(qml.PauliZ([0])), qml.sample(qml.PauliZ([1]))
```

Запустим схему сначала без применения операции SWAP:

```
circuit(do_swap=False)
```

```
array([[ -1,  1]])
```

А затем – с применением:

```
circuit(do_swap=True)
```

```
array([[ 1, -1]])
```

Видим, что во втором случае операция SWAP сработала: состояния кубитов поменялись местами. Можно посмотреть, как выглядит такая схема:

```
print(circuit.draw())
```

```
0: —X— rU0—| Sample[Z]
1: _____ lU0—| Sample[Z]
U0 =
[[1 0 0 0]
 [0 0 1 0]
 [0 1 0 0]
 [0 0 0 1]]
```

28.1 Введение

`Cirq` – это библиотека для работы с квантовыми компьютерами и симуляторами компании *Google*. В рамках темы квантового машинного обучения нам также интересен фреймворк `Tensorflow Quantum` или сокращенно `TFQ`. Это высокоуровневая библиотека, которая содержит готовые функции для квантового и гибридного машинного обучения. В качестве системы автоматического дифференцирования, а также для построения гибридных квантово-классических нейронных сетей там используется библиотека `Tensorflow`.

Warning: Во всех дальнейших лекциях мы будем использовать в основном библиотеку `PennyLane`, так что данная лекция исключительно обзорная и факультативная. В ней мы посмотрим несколько примеров *end2end* обучения квантовых схем на `TFQ` без детального объяснения теории и вообще того, что происходит. Основная цель данной лекции – исключительно обзор еще одного инструмента, а не изучение `QML`! Заинтересованный читатель может вернуться к этому обзору после изучения глав про *VQC*, *Градиенты* и **Квантовые нейросети**.

28.2 Работа с кубитами

28.2.1 Импорты и схема

Warning: `Tensorflow Quantum` нельзя установить на `Windows`, поэтому те `Makefiles`, которые мы сделали для слушателей не содержат шаг установки `tensorflow-quantum`. Если ваша система `POSIX`-совместимая и вы хотите обязательно попробовать запустить примеры кода из этой лекции, то просто в корне данного репозитория наберите команду:

```
poetry run pip install tensorflow-quantum==0.5.1
```

Обладателям компьютеров на системе `Windows` мы можем лишь предложить использовать `WSL2` или `Docker`. В целом эта лекция факультативная и нигде далее `TFQ` у нас не используется.

Больше информации можно найти в [разделе про установку](#) в официальной документации этой библиотеки.

Для начала импортируем `cirq`.

```
import cirq
```

`Cirq` рассчитан на работу с квантовым компьютером от компании *Google*, который представляет собой решетку кубитов. Например, вот так выглядит решетка кубитов квантового компьютера *Sycamore*:

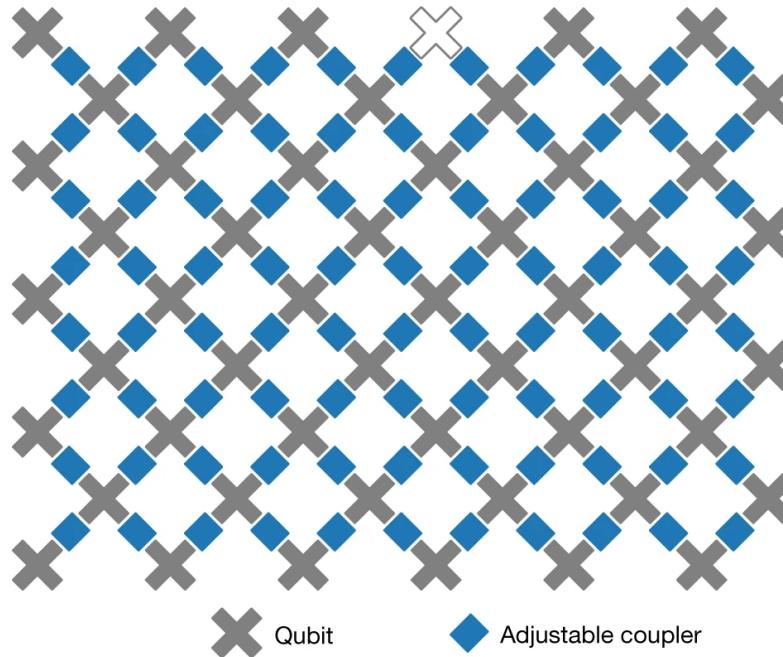


Fig. 28.1: Изображение из [AAB+19]

Поэтому в нем все строится вокруг работы с решеткой кубитов – объектом `cirq.GridQubit`. Давайте создадим кубит на решетке, который имеет координаты $(0, 0)$:

```
qubit = cirq.GridQubit(0, 0)
```

Следующей важной концепцией в `Cirq` является непосредственно квантовая схема. Давайте создадим схему, которая переводит кубит в суперпозицию состояний 0 и 1 и измеряет его:

```
circuit = cirq.Circuit()
circuit.append(cirq.H(qubit))
circuit.append(cirq.measure(qubit))
print(circuit)
```

```
(0, 0): —H—M—
```

28.2.2 Запуск и симуляция

Теперь создадим квантовый симулятор, который посчитает нам результат этой простой схемы на классическом компьютере:

```
sim = cirq.Simulator()
```

Как мы знаем, результат измерения такой схемы равен 50% для состояния 0, то есть если мы будем сэмплировать, то должны получать ~ 0.5 . Проверим это с разным числом сэмплов:

```
print(f"5 сэмплов: {sim.sample(circuit, repetitions=5).mean()}")
print(f"\n100 сэмплов: {sim.sample(circuit, repetitions=100).mean()}")
print(f"\n1000 сэмплов: {sim.sample(circuit, repetitions=1000).mean()})")
```

```
5 сэмплов: (0, 0)      0.0
dtype: float64

100 сэмплов: (0, 0)     0.48
dtype: float64

1000 сэмплов: (0, 0)    0.481
dtype: float64
```

Note: Метод `sim.sample` возвращает хорошо знакомый всем специалистам в области Data Science объект `pandas.DataFrame`.

Также у нас есть опция запустить схему через метод `run`. Может показаться, что это то же самое, но на самом деле в отличие от `sample`, метод `run` возвращает результат в несколько ином виде; а еще он позволяет запускать программу на реальном квантовом компьютере Google или их новых квантовых симуляторах на TPU:

```
print(sim.run(circuit, repetitions=25))
```

```
(0, 0)=1111110111100111001011110
```

Тут мы просто видим последовательность наших измерений.

28.3 Квантовое машинное обучение

28.3.1 Импорты

Мы будем использовать Tensorflow и Tensorflow Quantum.

```
import os
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
import tensorflow as tf
import tensorflow_quantum as tfq
```

28.3.2 Задача

Давайте попробуем решить игрушечную задачку классификации простой гибридной квантово-классической нейронной сетью. У нас будет один квантовый слой и один классический слой. В качестве задачи сгенерируем простенький набор данных, используя рутины `scikit-learn`. Сразу переведем входящие признаки в диапазон от нуля до π .

```
from sklearn.datasets import make_classification
import numpy as np

x, y = make_classification(n_samples=50, n_features=2, n_informative=2, random_
state=42, n_redundant=0)
```

(continues on next page)

(continued from previous page)

```
def normalize(x):
    x_min = x.min()
    x_max = x.max()

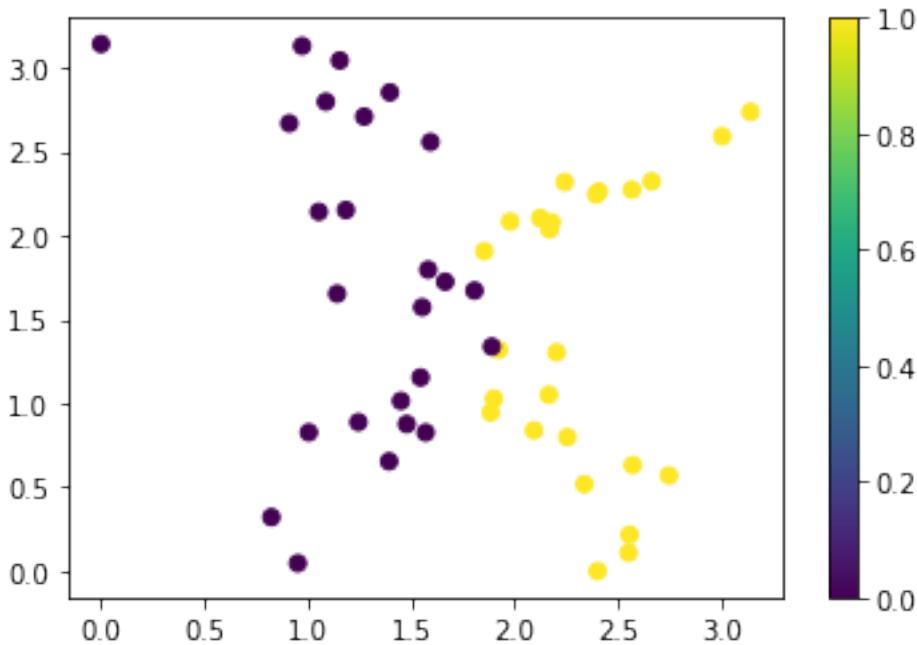
    return np.pi * (x - x_min) / (x_max - x_min)

x[:, 0] = normalize(x[:, 0])
x[:, 1] = normalize(x[:, 1])
```

Посмотрим на эти данные:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))
cb = plt.scatter(x[:, 0], x[:, 1], c=y)
plt.colorbar(cb)
plt.show()
```



28.3.3 Кубиты

Tensorflow Quantum позволяет “превращать” параметризованные схемы Cirq в слои нейронных сетей Tensorflow. Но для начала нам все равно потребуется схема. Давайте объявим пару кубитов.

```
qubits = [cirq.GridQubit(0, 0), cirq.GridQubit(0, 1)]
print(qubits)
```

```
[cirq.GridQubit(0, 0), cirq.GridQubit(0, 1)]
```

28.3.4 Входной слой нейронной сети

Определим входной слой, который будет кодировать наши классические данные в квантовые. Сразу закодируем данные. Так как Tensorflow работает с тензорами, то нам необходимо будет преобразовать схемы в тензор. Для этого есть специальная функция `convert_to_tensor`.

```
def data2circuit(x):
    input_circuit = cirq.Circuit()

    input_circuit.append(cirq.Ry(rads=x[0]).on(qubits[0]))
    input_circuit.append(cirq.Ry(rads=x[1]).on(qubits[1]))

    return input_circuit

x_input = tfq.convert_to_tensor([data2circuit(xi) for xi in x])
```

28.3.5 Слой из параметризованной схемы

Для создания параметризованных схем в Tensorflow Quantum используются символы из библиотеки символьных вычислений `sympy`. Давайте объявим несколько параметров и создадим схему:

```
from sympy import symbols

params = symbols("w1, w2, w3, w4")

trainable_circuit = cirq.Circuit()

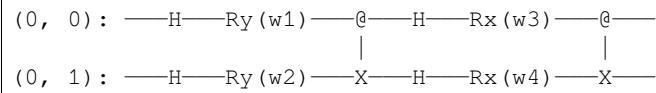
trainable_circuit.append(cirq.H.on(qubits[0]))
trainable_circuit.append(cirq.H.on(qubits[1]))
trainable_circuit.append(cirq.Ry(rads=params[0]).on(qubits[0]))
trainable_circuit.append(cirq.Ry(rads=params[1]).on(qubits[1]))

trainable_circuit.append(cirq.CNOT.on(qubits[0], qubits[1]))

trainable_circuit.append(cirq.H.on(qubits[0]))
trainable_circuit.append(cirq.H.on(qubits[1]))
trainable_circuit.append(cirq.Rx(rads=params[2]).on(qubits[0]))
trainable_circuit.append(cirq.Rx(rads=params[3]).on(qubits[1]))

trainable_circuit.append(cirq.CNOT.on(qubits[0], qubits[1]))

print(trainable_circuit)
```



28.3.6 Наблюдаемые

В качестве операторов, которые мы будем измерять, воспользуемся парой $\hat{X}\hat{Y}$ и $\hat{Y}\hat{X}$ для наших кубитов:

```
ops = [cirq.X.on(qubits[0]) * cirq.Y.on(qubits[1]), cirq.Y.on(qubits[0]) * cirq.X.
       ↪on(qubits[1])]
```

28.3.7 Гибридная нейронная сеть

Теперь воспользуемся классическим Tensorflow, чтобы объявить и скомпилировать нашу нейронную сеть, предварительно добавив в нее один классический слой.

- зафиксируем случайный генератор

```
tf.random.set_seed(42)
```

- входной тензор – это в нашем случае тензор типа `string`, так как это квантовые схемы

```
cirq_inputs = tf.keras.Input(shape=(), dtype=tf.dtypes.string)
```

- квантовый слой

```
quantum_layer = tfq.layers.PQC(trainable_circuit, ops)(cirq_inputs)
```

```
WARNING:tensorflow:From /home/runner/work/qmlcourse/qmlcourse/.venv/lib/python3.8/
         ↪site-packages/tensorflow/python/ops/array_ops.py:5043: calling gather (from
         ↪tensorflow.python.ops.array_ops) with validate_indices is deprecated and will be
         ↪removed in a future version.
Instructions for updating:
The `validate_indices` argument has no effect. Indices are always validated on CPU
         ↪and never validated on GPU.
```

- классический слой и выходной слой

```
dense_layer = tf.keras.layers.Dense(2, activation="relu")(quantum_layer)
output_layer = tf.keras.layers.Dense(1, activation="sigmoid")(dense_layer)
```

- компилируем модель и смотрим, что получилось. И сразу указываем метрики, которые хотим отслеживать

```
model = tf.keras.Model(inputs=cirq_inputs, outputs=output_layer)
model.compile(
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.1),
    loss=tf.keras.losses.BinaryCrossentropy(),
    metrics=[
        tf.keras.metrics.BinaryAccuracy(),
        tf.keras.metrics.BinaryCrossentropy(),
    ]
)
model.summary()
```

```
Model: "model"
```

Layer (type)	Output Shape	Param #
<hr/>		
input_1 (InputLayer)	[(None,)]	0

(continues on next page)

(continued from previous page)

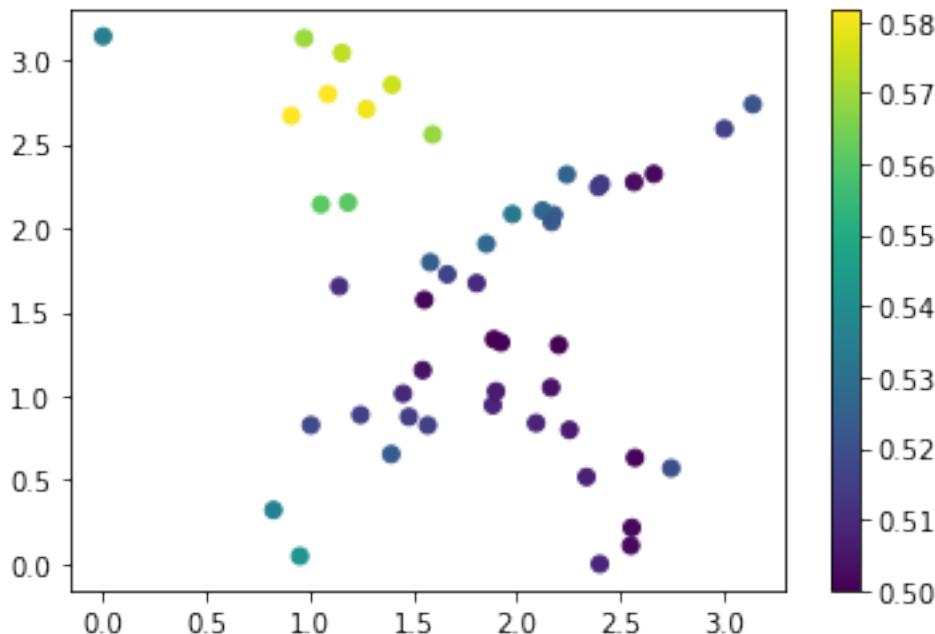
pqc (PQC)	(None, 2)	4
dense (Dense)	(None, 2)	6
dense_1 (Dense)	(None, 1)	3
<hr/>		
Total params: 13		
Trainable params: 13		
Non-trainable params: 0		

28.3.8 Предсказания со случайной инициализацией

Наша нейросеть имеет случайные начальные параметры. Давайте посмотрим, что она предсказывает до обучения:

```
preds = model(x_input).numpy()

plt.figure(figsize=(6, 4))
cb = plt.scatter(x[:, 0], x[:, 1], c=preds)
plt.colorbar(cb)
plt.show()
```



28.3.9 Обучение сети

- запустим обучение

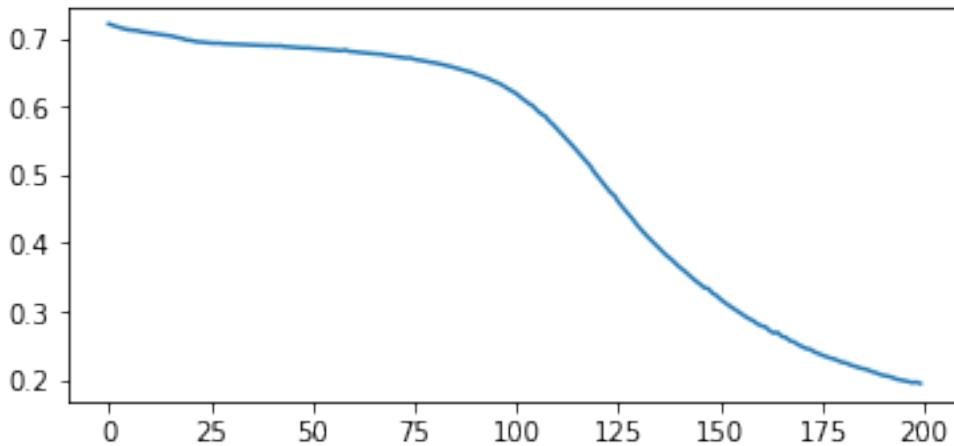
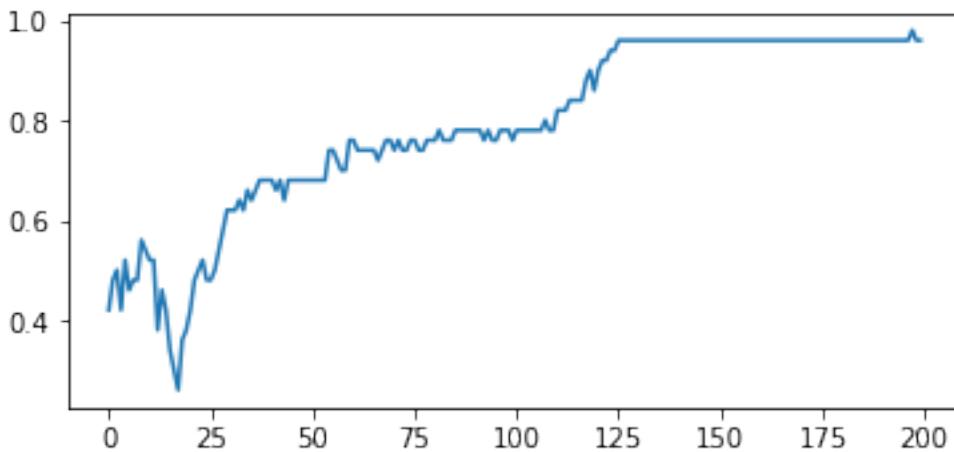
```
%%time
model.fit(x=x_input, y=y, epochs=200, verbose=0)
```

```
CPU times: user 4.18 s, sys: 40 ms, total: 4.22 s
Wall time: 3.85 s
```

```
<tensorflow.python.keras.callbacks.History at 0x7efe6ca18970>
```

- визуализируем логи обучения

```
f, ax = plt.subplots(2, figsize=(6, 6))
ax[0].plot(model.history.history["binary_accuracy"])
ax[1].plot(model.history.history["binary_crossentropy"])
plt.show()
```



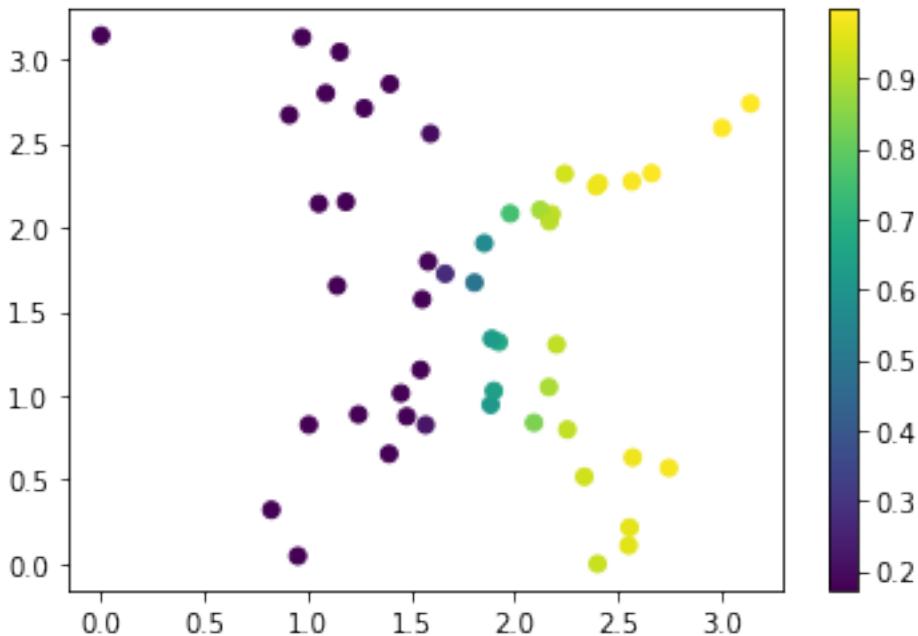
- визуализируем предсказания

```
preds_after_training = model(x_input).numpy()
plt.figure(figsize=(6, 4))
```

(continues on next page)

(continued from previous page)

```
cb = plt.scatter(x[:, 0], x[:, 1], c=preds_after_training)
plt.colorbar(cb)
plt.show()
```



28.4 Заключение

В данной лекции мы познакомились с фреймворком Tensorflow Quantum. Это достаточно мощный инструмент, особенно в связке с Tensorflow, так как позволяет использовать большое число готовых методов Tensorflow и различных расширений. Тем не менее, для целей обучения Tensorflow Quantum кажется не лучшим выбором, так как имеет много неочевидного синтаксиса и предполагает, как минимум, среднего знания Tensorflow. Во всех дальнейших лекциях мы будем использовать в основном библиотеку PennyLane.

В этой лекции мы посмотрим, как при помощи библиотеки Qiskit можно работать с кубитами и квантовыми схемами.

Warning: Во всех дальнейших лекциях мы будем, за редким исключением, использовать в основном библиотеку PennyLane, так что данная лекция исключительно обзорная и факультативная. В ней мы поверхностно познакомимся с концептами Qiskit и напишем пару примеров по работе с кубитами. Обзор экосистемы Qiskit, включая qiskit-machine-learning выходит за рамки этой лекции, но в конце мы дадим несколько ссылок для самостоятельного изучения.

29.1 Кубиты

29.1.1 Импорты

```
from qiskit import QuantumCircuit
from qiskit.providers.aer import QasmSimulator

from qiskit import version
print(f"Qiskit version: {version.get_version_info() }")
```

```
Qiskit version: 0.18.3
```

29.1.2 Симулятор

Для запуска мы будем использовать симулятор квантового компьютера QasmSimulator:

```
simulator = QasmSimulator()
```

В Qiskit поддерживается несколько симуляторов, например:

- UnitarySimulator – симулятор унитарной динамики идеальной схемы;
- StatevectorSimulator – симулятор волновой функции идеальной схемы;
- QasmSimulator – симулятор, который моделирует шумы;

Мы использовали лишь один из них. В целом, модуль `qiskit.providers` предоставляет базовые абстракции, которые позволяют запускать наш код на разных backend, включая реальные квантовые компьютеры, просто поменяв одну строчку в коде.

29.1.3 Квантовая схема

Ключевым элементом в Qiskit является квантовая схема – экземпляр класса `QuantumCircuit`. Создадим такую схему:

```
circuit = QuantumCircuit(2, 2)
```

В Qiskit необходимо явно указывать не только количество кубитов, которые нам необходимы, но также и количество классических битов, которые необходимы будут нам для измерений. В данном случае мы создали квантовую схему с двумя кубитами и двумя битами.

29.2 Квантовые гейты

Давайте попробуем воспроизвести функцию `make_entanglement` из лекции про `Pennylane`. Применим гейт Адамара к первому кубиту, а потом применим гейт **CNOT (CX)**:

```
circuit.h(0)
circuit.cx(0, 1)
```

```
<qiskit.circuit.instructionset.InstructionSet at 0x7f3e783f84f0>
```

И добавим шаг измерений:

```
circuit.measure([0, 1], [0, 1])
```

```
<qiskit.circuit.instructionset.InstructionSet at 0x7f3e783f86d0>
```

29.2.1 Компиляция и запуск схемы

Квантовые схемы Qiskit должны быть скомпилированы в инструкции квантового ассемблера. Для этого используется функция `transpile`:

```
from qiskit import transpile

compiled_circuit = transpile(circuit, simulator)
job = simulator.run(compiled_circuit, shots=1000)
result = job.result()
counts = result.get_counts(compiled_circuit)

print(f"Number of '00': {counts['00']}")
print(f"Number of '11': {counts['11']}")
```

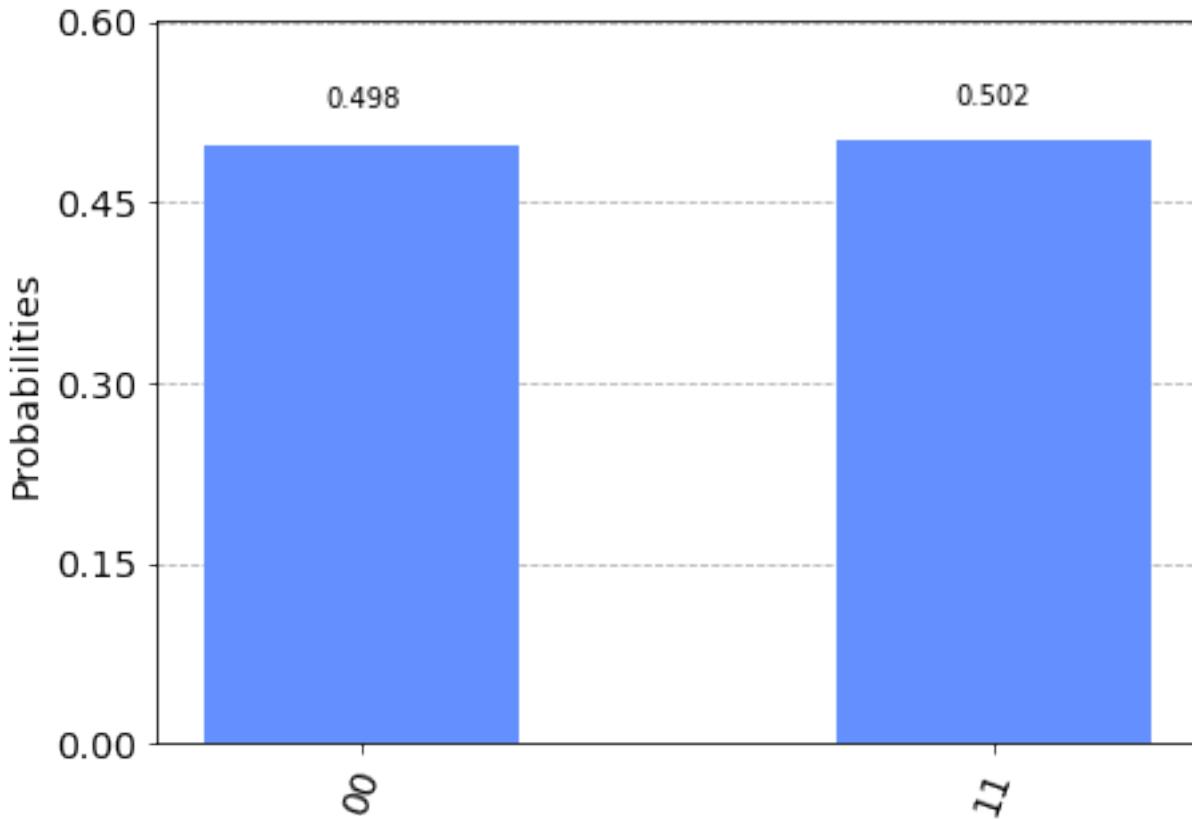
```
Number of '00': 498
Number of '11': 502
```

Мы получили результат, аналогичный полученному в лекции про `Pennylane`: вероятности получения результата 00 и 11 равны ~ 0.5 , а другие результаты мы получить не можем.

29.3 Визуализация

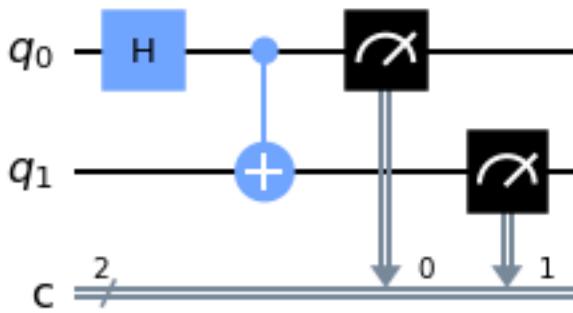
В Qiskit есть очень мощные средства визуализации как результатов измерений, так и самих квантовых схем. Например, мы можем посмотреть результаты измерения `counts` в виде графика:

```
from qiskit.visualization import plot_histogram
plot_histogram(counts)
```



А также визуализировать саму квантовую схему:

```
circuit.draw(output="mpl")
```



В данном случае мы использовали `MatplotlibDraw`, который позволяет делать очень красивые и легко читаемые визуализации квантовых схем – это одна из причин почему `Qiskit` часто используют в курсах по основам квантовых вычислений.

29.4 Экосистема Qiskit

Для более подробного изучения всей экосистемы `Qiskit` мы рекомендуем [онлайн книгу](#), которая также является прекрасным источником для изучения базовых и продвинутых квантовых алгоритмов. Для знакомства со средствами для квантового машинного обучения можно также изучить [документацию библиотеки `qiskit-machine-learning`](#), в которой представлены высокоуровневые API для таких вещей, как, например, квантовый `SVM`, или квантовая Generative Adversarial Network.

О БЛОКЕ “БАЗОВЫЕ КВАНТОВЫЕ АЛГОРИТМЫ”

Этот блок включает в себя:

- общий обзор наиболее известных квантовых алгоритмов;
- реализации алгоритмов Дойча и квантовой телепортации.

Продвинутые темы блока дополнительно рассказывают об алгоритмах:

- Саймона;
- Гровера.

ОБЗОР КВАНТОВЫХ АЛГОРИТМОВ

Квантовые вычисления открывают новые возможности решения задач, для которых ранее были известны только классические алгоритмы решения. С появлением идеи квантового компьютера стало понятно, что нахождение ответа для многих задач можно значительно ускорить. При этом некоторым сложным задачам, решить которые классическим способом в разумные сроки невозможно, квантовый компьютер дает реальный шанс быть решенными.

31.1 Классификация задач по временной сложности

Вообще, в соответствии с теорией алгоритмов, задачи можно разбить на классы по временной сложности их решения. Также часто используется классификация задач по объему необходимой памяти (пространственная сложность), но нас в первую очередь волнует, насколько быстро мы сможем найти правильный ответ, так что поговорим о временной сложности. Класс задач P – это те задачи, которые возможно решить на классическом компьютере за полиномиальное время, их сложность можно представить как $O(n^k)$. Соответственно, решение задач этого класса не является драматически затратным по времени (за исключением случаев, когда степень полинома высока, хотя такие алгоритмы и не являются типичными).

Другой класс задач – NP (расшифровывается как “недетерминировано полиномиальные”). Это класс задач, для которых неизвестно полиномиальное классическое решение. При этом проверка решения возможна за полиномиальное время. К примеру, нахождение простых множителей большого числа с помощью известных классических алгоритмов – субэкспоненциально сложная задача (работает за время, большее, чем полиномиальное, но меньшее, чем экспоненциальное). Проверить же найденное решение предельно просто: достаточно перемножить найденные простые числа.

Среди NP -задач есть наиболее сложные задачи, которые выделяют в специальную группу под названием NP -полные задачи (или NP -complete). Если найти для них быстрое решение, то этот способ решения также можно применить и к “обычным” NP -задачам.

Вообще, есть также задачи, которые, хотя и не относятся к классу NP , но несмотря на это, к ним все же можно свести задачи из NP -класса. В совокупности с NP -полными задачами они образуют класс NP -трудных задач (NP -hard).

К примеру, к NP -hard задачам относится задача коммивояжера, в которой требуется установить кратчайший путь. Как решение, так и его проверка в данном случае займут суперполиномиальное время, так что эта задача не входит в класс NP -complete. Если же ее немного упростить, так что решением будет являться путь не дольше заданного, то такая разновидность задачи является примером класса NP -complete, так проверка ее решения не требует времени, большего, чем полиномиальное.

Если дать возможность классическому компьютеру решать задачи с привнесением случайности, так что компьютер получает правильный ответ с высокой вероятностью (стандартно берут порог не менее $\frac{2}{3}$, хотя достаточно, чтобы вероятность была константой больше 0.5), то можно говорить о классе задач BPP (сокращение от англ. bounded-error, probabilistic, polynomial). Такие задачи получается решить за полиномиальное время, причем точность решения можно сколько угодно увеличивать, повторно запуская

алгоритм. Квантовый аналог таких задач – класс BQP (от англ. bounded error quantum polynomial time). Это задачи, которые получается решить на квантовом компьютере за полиномиальное время, обеспечивая точность решения повторным запуском алгоритма. Наиболее известный пример такой задачи – факторизация чисел, решаемая на квантовом компьютере с помощью алгоритма Шора.

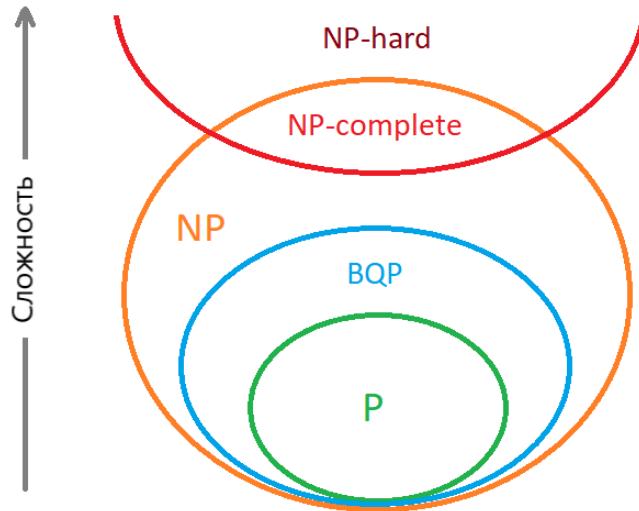


Fig. 31.1: Классы задач по временной сложности

На данный момент можно говорить о том, что класс BQP включает в себя в том числе и задачи, для которых неизвестно полиномиальное классическое решение, что позволяет относиться к квантовым вычислениям с оптимизмом.

31.2 Наиболее известные квантовые алгоритмы

Квантовые вычисления, несмотря на новые возможности, которые они предоставляют, все же не являются панацеей: не для всех классических “медленных” (то есть, не решаемых за полиномиальное время) алгоритмов пока удалось найти ускоренный квантовый аналог. Более того, многие даже более простые задачи в настоящий момент выгоднее решать на классических компьютерах. Тем не менее, уже найдены квантовые алгоритмы, работающие быстрее классических. Кратко расскажем о наиболее важных из них.

Алгоритм Шора – алгоритм, наделавший больше всего шума и привлекший внимание научно-популярных СМИ к квантовым вычислениям. Действительно, этот алгоритм дает повод для беспокойства, так как он позволяет узнавать содержание сообщений, зашифрованных с помощью алгоритма шифрования RSA. Для расшифровки требуется разложить большое число на два простых множителя. Для классического компьютера решение этой задачи может занять несколько тысяч лет, а для алгоритма Шора это дело считанных часов или даже минут. Такая скорость вычислений обусловлена тем, что на квантовом компьютере удается ускорить преобразование Фурье (как прямое так и обратное). Благодаря алгоритму Шора начала развиваться квантовая криптография – шифрование, неуязвимое для атак.

Еще один алгоритм, способный преобразить мир ИТ – **алгоритм Гровера**. Благодаря ему возможно ускорить поиск по базе данных. Если на классическом компьютере решить задачу поиска элемента в базе данных возможно только перебором всех элементов, то на квантовом компьютере можно получить квадратичное уменьшение сложности, так как за счет использования эффектов суперпозиции и квантовой запутанности

алгоритм Гровера “просматривает” одновременно все элементы, хотя и делает это много раз, постепенно выявляя правильное решение.

Некоторые квантовые алгоритмы пока не выглядят полезными с практической точки зрения, но даже и в таком случае они уже демонстрируют возможности, которых нет в классических вычислениях. К примеру, **алгоритм Дойча** и **алгоритм Саймона** не несут особой практической пользы в силу своей простоты, но даже такие простые примеры квантовых вычислений демонстрируют значительное ускорение (в данном случае экспоненциальное). Эти алгоритмы позволяют быстро установить свойства функций. Если алгоритм Дойча определяет, является ли функция сбалансированной, то с помощью алгоритма Саймона можно вычислить период некоторой функции.

31.2.1 Перспективы квантовых алгоритмов

С увеличением числа кубитов и уменьшением количества ошибок в квантовых компьютерах известные квантовые алгоритмы смогут показать себя в полной мере, но также станет возможным находить новые, более сложные и практически полезные квантовые алгоритмы. Заниматься их поиском в ближайшее время будут не только физики и математики, но и программисты, освоившие квантовые вычисления.

31.3 Алгоритмы в нашем курсе

Для более глубокого погружения в детали этих квантовых алгоритмов у нас в курсе есть отдельные лекции по:

- *Алгоритм Дойча (начальный уровень)*
- *Квантовая телепортация (начальный уровень) на Qiskit*
- *Алгоритм Гровера (продвинутый уровень)*

CHAPTER
THIRTYTWO

АЛГОРИТМ ДОЙЧА

Алгоритм Дойча (в английском варианте – **Deutsch's algorithm**) – это один из первых алгоритмов, показавших, что квантовый компьютер может решать задачи особым способом, отличающимся как от алгоритмов классического компьютера, так и от интуиции и здравого смысла человека. При этом такое решение может занимать меньшее количество шагов.

Нужно прежде всего сказать, что алгоритм Дойча не имеет практического применения в силу своей предельной простоты, зато является простейшим примером, с помощью которого можно понять, в чем состоит отличие квантовых алгоритмов от классических. Данный алгоритм был предложен в 1985 году, когда квантовых компьютеров еще не было, а практически он был реализован в 1998 году на 2-кубитном квантовом компьютере, работавшем на принципах ядерно-магнитного резонанса.

Дэвид Дойч

Помимо занятий теоретической физикой в Оксфордском университете, Дэвид Дойч – автор книг “Структура реальности” и “Начало бесконечности”, в которых он популярно излагает идеи квантовых вычислений с точки зрения многомировой интерпретации (сторонником которой является) и философствует о будущем науки и человечества. Так что можно сказать, что работа алгоритма, согласно замыслу создателя, производится в параллельных вселенных. Так это или нет, пока проверить невозможно, но вычисления работают, и это главное.

Итак, в чем состоит задача, которую решает алгоритм? Представьте, что у вас есть функция, которая представляет собой “черный ящик”, принимающий на вход число из множества $\{0, 1\}$. Функция неким образом обрабатывает входное значение и возвращает число из этого же множества, то есть либо 0, либо 1. Нам известно, что эта функция принадлежит либо к классу сбалансированных функций, либо к классу константных функций (которые мы также можем называть несбалансированными). Задача алгоритма – установить, к какому классу принадлежит функция.

Рассмотрим все варианты этих двух классов. Всего их четыре, то есть по две функции в каждом классе. Начнем с несбалансированных:

- $f_1(x) = 0$

Это функция, всегда возвращающая 0, независимо от входного значения. Для нее справедливы выражения:

$$f_1(0) = 0$$

$$f_1(1) = 0$$

- $f_2(x) = 1$

Такая функция всегда возвращает 1, то есть верно следующее:

$$f_2(0) = 1$$

$$f_2(1) = 1$$

Ну а теперь посмотрим на сбалансированные функции. Для них характерно то, что они могут возвращать как 0, так и 1. В этом и заключается “баланс”.

- $f_3(x) = x$

Это тождественная функция, которая ничего не делает с входным значением. Для нее справедливо следующее:

$$f_3(0) = 0$$

$$f_3(1) = 1$$

- $f_4(x) = \bar{x}$

А вот эта функция инвертирует входное значение, то есть возвращает не то число, которое было подано на вход, а другое:

$$f_4(0) = 1$$

$$f_4(1) = 0$$

Классический компьютер справляется с задачей за два шага. Например, нам дана некоторая функция-“черный ящик”, и мы должны установить, сбалансирована ли она. На первом шаге мы отправляем в функцию входное значение 0. Допустим, мы получили на выходе также 0. Мы можем сказать, что данная функция – либо f_1 (константная функция, всегда возвращающая 0), либо f_3 (сбалансированная функция, не меняющая входное значение). Для окончательного решения мы должны сделать еще один шаг – отправить в функцию значение 1. Если при этом мы получим опять 0, то это функция f_1 , а если получили на выходе 1, то искомая функция – f_3 .

Способа, с помощью которого на классическом компьютере можно за одно действие установить, сбалансирована функция или нет, не существует. И здесь свое преимущество показывает квантовый компьютер: он может установить класс функции за одно действие.

Для начала рассмотрим простейшую схему, с помощью которой можно отправлять число на вход и получать ответ от черного ящика:

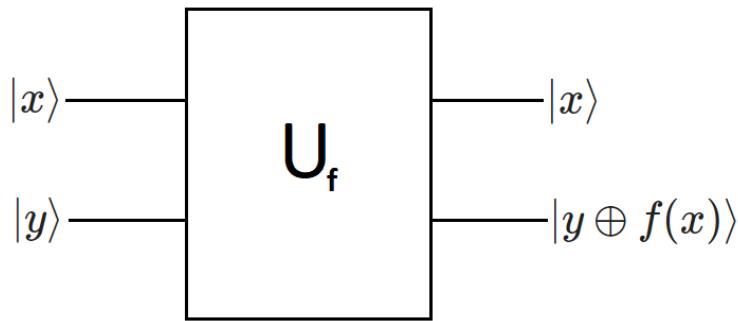


Fig. 32.1: Схема 1

U_f на данной схеме – это неизвестная функция (ее также часто называют “оракул”), являющаяся унитарным оператором. Обратите внимание, что в квантовой схеме используются два кубита. Это нужно для того, чтобы информация, с которой работает квантовый компьютер, не стиралась. В квантовом компьютере важно, чтобы все действия с кубитами (кроме операции измерения) были обратимыми, а для этого информация должна сохраняться. В верхнем кубите будет записано входное значение, а в нижнем – выходное значение функции. Таким образом, входное значение не будет перезаписано значением, которое вернет функция.

Но нам важно будет не только сохранить значение $|x\rangle$, но также и не разрушить $|y\rangle$. Так как кубит y очевидно имеет некоторое изначальное значение, мы не можем его просто перезаписать тем числом, которое выдаст

функция $f(x)$. Здесь на помощь приходит операция исключающее ИЛИ - XOR (также ее можно называть сложением по модулю 2), обозначенная на схеме как \oplus . В процессе работы черный ящик U_f не только находит значение $f(x)$, но и применяет исключающее ИЛИ к значениям y и $f(x)$.

Операции XOR соответствует такая таблица истинности:

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

Операция XOR хороша для нас тем, что она не разрушает значение $|y\rangle$, так как является обратимой. Убедиться в этом можно, проверив тождество:

$$(a \oplus b) \oplus b = a$$

Схема 1 пока что не дает преимущества по сравнению с классическим компьютером, но мы можем ее немного усовершенствовать:

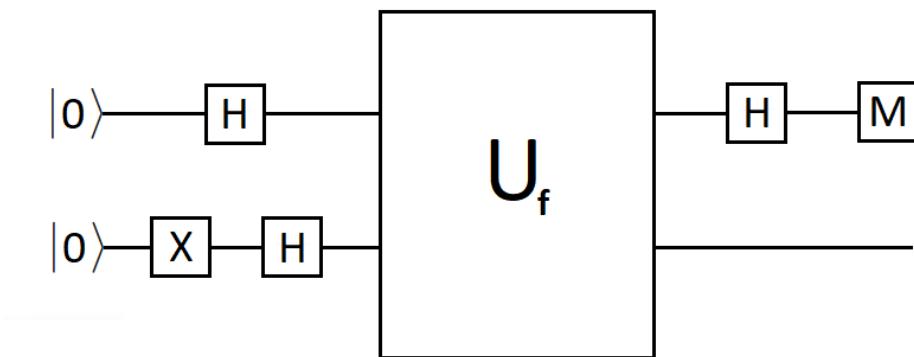


Fig. 32.2: Схема 2

В новой схеме оба кубита вначале будут находиться в состоянии $|0\rangle$. Затем мы применим к верхнему кубиту оператор Адамара, а к нижнему – гейт X , а затем так же, как и к верхнему, оператор Адамара. Тем самым мы приведем оба кубита в состояние суперпозиции перед тем, как они попадут на вход функции U_f . Верхний кубит будет находиться в такой суперпозиции:

$$|x\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle,$$

а нижний – в такой:

$$|y\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle.$$

Если рассмотреть это как систему $|\psi\rangle$, состоящую из двух кубитов, то она будет выглядеть так:

$$|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle)$$

Сразу после U_f система будет находиться в состоянии:

$$|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle \oplus f(x)) - |1\rangle \oplus f(x))$$

После того, как U_f отрабатывает, нижний кубит, как это ни странно, уже нас не интересует, так что к нему операции больше не применяются, и его измерение также не производится.

Дело в том, что ответ на вопрос о том, сбалансирована функция $f(x)$ или нет, будет нами получен из верхнего кубита после того, как на него подействует оператор Адамара и будет произведено измерение. В том случае, если функция сбалансирована, результат измерения верхнего кубита будет равен 1, а если несбалансированна – 0.

Разберемся подробнее, почему это происходит. Рассмотрим все возможные $f(x)$, которые могут находиться в черном ящике:

- $f(x) = f_1$

В этом случае $f(x)$ всегда принимает значение 0, и система кубитов будет выглядеть так:

$$|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle) = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)$$

- $f(x) = f_2$

$f(x)$ будет равно 1 независимо от аргумента. Используя таблицу истинности XOR, легко убедиться, что во второй скобке $|0\rangle$ и $|1\rangle$ поменяются местами, но если вынести минус за скобку, то мы можем его не учитывать, так общий фазовый множитель (-1 в данном случае) для системы не имеет значения:

$$|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|1\rangle - |0\rangle) = -\frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle) = -\frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)$$

Видно, что при применении функций f_1 и f_2 , являющихся несбалансированными, мы получаем фактически одно и тоже состояние. Если после этого применить к первому кубиту оператор Адамара, то после измерения мы получим значение 0.

Рассмотрим теперь сбалансированные функции f_3 и f_4 .

- $f(x) = f_3$

Здесь ситуация сложнее, так как $f(x)$ будет зависеть от состояния первого кубита. Поэтому мы раскроем скобки, а значения функции подставим позже:

$$\begin{aligned} |\psi\rangle &= \frac{1}{2}(|0\rangle + |1\rangle)(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle) = \frac{1}{2}(|0\rangle|0 \oplus f(x)\rangle - |0\rangle|1 \oplus f(x)\rangle + |1\rangle|0 \oplus f(x)\rangle - |1\rangle|1 \oplus f(x)\rangle) = \\ &= \frac{1}{2}(|0\rangle|0 \oplus 0\rangle - |0\rangle|1 \oplus 0\rangle + |1\rangle|0 \oplus 1\rangle - |1\rangle|1 \oplus 1\rangle) = \frac{1}{2}(|00\rangle - |01\rangle + |11\rangle - |10\rangle) = \\ &= \frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle) = \frac{1}{2}(|0\rangle - |1\rangle)(|0\rangle - |1\rangle) \end{aligned}$$

Видно, что первый кубит поменял свое состояние – теперь он в суперпозиции $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, так что далее к нему можно применить оператор Адамара, после которого он перейдет в состояние $|1\rangle$.

- $f(x) = f_4$

Здесь будет похожая ситуация:

$$\begin{aligned} |\psi\rangle &= \frac{1}{2}(|0\rangle + |1\rangle)(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle) = \frac{1}{2}(|0\rangle|0 \oplus f(x)\rangle - |0\rangle|1 \oplus f(x)\rangle + |1\rangle|0 \oplus f(x)\rangle - |1\rangle|1 \oplus f(x)\rangle) = \\ &= \frac{1}{2}(|0\rangle|0 \oplus 1\rangle - |0\rangle|1 \oplus 1\rangle + |1\rangle|0 \oplus 0\rangle - |1\rangle|1 \oplus 0\rangle) = \frac{1}{2}(|01\rangle - |00\rangle + |10\rangle - |11\rangle) = \\ &= -\frac{1}{2}(|00\rangle - |01\rangle + |11\rangle - |10\rangle) = -\frac{1}{2}(|0\rangle - |1\rangle)(|0\rangle - |1\rangle) \end{aligned}$$

Получили то же состояние $|\psi\rangle$, что и для f_3 , с точностью до фазового множителя. Соответственно, здесь первый кубит после применения оператора Адамара также будет измерен с результатом 1.

Теперь можно получить более компактную формулу, которая подходит для всех четырех функций:

$$|\psi\rangle = \frac{1}{2}((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle)(|0\rangle - |1\rangle)$$

Задание

С помощью квантовых операторов попробуйте создать U_f для всех четырех $f(x)$.

Задание рекомендуется сделать до прочтения программистской части по алгоритму Дойча, так как там содержится ответ.

32.1 Алгоритм Дойча в коде

Запрограммируем алгоритм с помощью библиотеки PennyLane. Предполагается, что функция, находящаяся в черном ящике, изначально присутствует, но для учебного примера создадим также и ее, точнее, все ее четыре варианта. Для того, чтобы нам не сразу было известно, какая из этих функций анализируется алгоритмом (иначе будет неинтересно), будем использовать случайный выбор функции.

Импортируем все необходимые библиотеки и модули, а также создадим квантовое устройство-симулятор, рассчитанное на схему из двух кубитов:

```
import pennylane as qml
from pennylane import numpy as np

dev = qml.device("default.qubit", shots=1, wires=2)
```

Теперь создадим функции для черного ящика. Обратите внимание, что здесь уже учтено сложение по модулю 2 результата функции с состоянием второго кубита:

```
def f1():
    pass

def f2():
    qml.PauliX(wires=[1])

def f3():
    qml.CNOT(wires=[0, 1])

def f4():
    qml.PauliX(wires=0)
    qml.CNOT(wires=[0, 1])
    qml.PauliX(wires=0)
```

Создадим словарь с функциями и их названиями:

```
black_boxes_dict = {"f1": f1, "f2": f2, "f3": f3, "f4": f4}
```

А вот таким образом мы будем случайно выбирать название функции для черного ящика:

```
def random_black_box(black_boxes_dict):
    black_boxes_dict_list_keys = list(black_boxes_dict.keys())
    n = np.random.randint(0, len(black_boxes_dict_list_keys))

    return black_boxes_dict_list_keys[n]
```

А теперь самое важное – сам алгоритм Дойча:

```
@qml.qnode(dev, interface=None)
def circuit(black_box_name):
    qml.Hadamard(wires=0)
    qml.PauliX(wires=1)
    qml.Hadamard(wires=1)

    black_boxes_dict[black_box_name]()
    qml.Hadamard(wires=0)

    return qml.sample(qml.PauliZ([0]))
```

Итак, подготовительные действия завершены, можно приступать к демонстрации работы алгоритма.

Выберем случайнym образом функцию:

```
black_box_name = random_black_box(black_boxes_dict)
```

А затем запустим алгоритм Дойча и выведем результат его работы. Собственное значение 1 оператора Z будет соответствовать состоянию $|0\rangle$ (функция несбалансированная), а собственное значение -1 – состоянию $|1\rangle$ (функция сбалансирована):

```
result = circuit(black_box_name)
print(result)
```

```
1
```

Проверим, насколько правильно сработал алгоритм. Для этого посмотрим на функцию из черного ящика:

```
print(black_box_name)
```

```
f1
```

Также посмотрим, как выглядит квантовая схема:

```
print(circuit.draw())
```

```
0: —H—H—| Sample[Z]
1: —X—H—|
```

На примере алгоритма Дойча мы видим, что уже двухкубитная схема дает прирост скорости в два раза. Если же увеличивать количество входных параметров (как в аналогичном алгоритме [Дойча-Йожи](#)), то ускорение будет экспоненциальным.

Для специалистов, занимающихся искусственным интеллектом, алгоритм может быть интересен тем, что не просто решает задачу нахождения некоторого значения, действуя как калькулятор, а дает возможность определить скрытую функцию. Это похоже на задачи машинного обучения, когда data scientist, производя математические манипуляции с данными, в итоге получает модель (фактически – функцию), описывающую связь признаков с целевой переменной. Таким образом, интерес специалистов ИИ к квантовым вычислениям, вполне понятен, как и перспективы квантовых вычислений в этой области.

КВАНТОВАЯ ТЕЛЕПОРТАЦИЯ

Пожалуй, пришло время познакомиться с квантовой телепортацией.

Часто вижу в новостях «телепортировали кубит», «телепортировали электрон». Но это лишь игра слов... Рассмотрим два запутанных электрона – первый находится у вас, а второй у вашего друга. И под телепортацией подразумевается вот что: вы, используя допустимую операцию, изменяете состояние своего электрона, и второй (запутанный с ним), также изменит свое состояние (в соответствии с примененной операцией). Вам остается лишь выбрать допустимую операцию.

Давайте сейчас посмотрим на [состояния Белла](#) (Bell states). Прошу:

$$|\beta_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \quad (33.1)$$

$$|\beta_{01}\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$

$$|\beta_{10}\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$$

$$|\beta_{11}\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$$

Это четыре всадника квантовой запутанности. И у них есть много преимуществ: используют всего два кубита, их легко получить... А что нам еще нужно? Дальше в примере мы воспользуемся одним из этих состояний.

Теперь работаем в терминах квантовой теории информации – это сильно поможет с пониманием остальных тем нашего с вами курса.

Итак, квантовая телепортация будет проводиться двумя персонами – Алисой и Бобом, из которых первые два кубита контролируются Алисой, а третий – Бобом. Алиса хочет передать Бобу закодированное сообщение (отождествляемое с квантовым состоянием): $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$

Изначально, вся система из трех кубитов будет иметь следующее состояние: $|\psi_0\rangle = |\psi\rangle|0\rangle|0\rangle$. Т.е. первый кубит находится в состоянии $|\psi\rangle$, которое затем будет передаваться с помощью второго кубита на третий.

Алиса и Боб

Первые два шага являются подготовительными. Поэтому Боб находится рядом и ждет, пока Алиса проведет необходимые операции.

1. Алиса применяет гейт Адамара ко второму кубиту:

$$|\psi_1\rangle = (I \otimes H \otimes I)|\psi_0\rangle = (I \otimes H \otimes I)|\psi\rangle|0\rangle|0\rangle = \frac{1}{\sqrt{2}}|\psi\rangle(|0\rangle + |1\rangle)|0\rangle$$

2. Далее Алиса использует гейт CNOT для того, чтобы запутать второй кубит с третьим:

$$\begin{aligned} |\psi_2\rangle &= (I \otimes CNOT(2,3))|\psi_1\rangle = \\ (I \otimes CNOT(2,3))\frac{1}{\sqrt{2}}|\psi\rangle(|0\rangle + |1\rangle)|0\rangle &= \frac{1}{\sqrt{2}}|\psi\rangle(|0\rangle|0\rangle + |1\rangle|1\rangle) \end{aligned}$$

Алиса и Боб

Теперь состояния готовы. Сейчас Боб, забрав третий кубит с собой, отправляется по своим делам. И Алиса, в случае необходимости, сможет передать ему послание.

3. Алиса применяет CNOT между первым и вторым кубитами. Вспоминая, что $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$:

$$\begin{aligned} |\psi_3\rangle &= (I \otimes CNOT(1,2))|\psi_2\rangle = \\ (I \otimes CNOT(1,2))\frac{1}{\sqrt{2}}(\alpha|0\rangle + \beta|1\rangle)(|00\rangle + |11\rangle) &= \\ &= \frac{1}{\sqrt{2}}[\alpha(|000\rangle + |011\rangle) + \beta(|110\rangle + |101\rangle)] \end{aligned}$$

4. Алиса применяет гейт Адамара на своем первом кубите:

$$\begin{aligned} |\psi_4\rangle &= (H \otimes I \otimes I)|\psi_3\rangle = \\ \frac{1}{2}[|00\rangle(\alpha|0\rangle + \beta|1\rangle) + & \\ |01\rangle(\alpha|1\rangle + \beta|0\rangle) + & \\ |10\rangle(\alpha|0\rangle - \beta|1\rangle) + & \\ |11\rangle(\alpha|1\rangle - \beta|0\rangle)] & \end{aligned}$$

Внимание

Видно, что в каждой строчке у нас получаются разные состояния, по первым двум кубитам мы можем определить, в каком будет третий. Обратите внимание на общий множитель $\frac{1}{2}$ в первой строчке, не пропустите.

5. Алисе нужно теперь измерить первые 2 кубита и станет ясно, в какое из четырех состояний перейдет кубит Боба:

00	$\alpha 0\rangle + \beta 1\rangle$
01	$\alpha 1\rangle + \beta 0\rangle$
10	$\alpha 0\rangle - \beta 1\rangle$
11	$\alpha 1\rangle - \beta 0\rangle$

6. В зависимости от полученных по классическому каналу данных, Боб должен применить одну из операций для того, чтобы восстановить исходное состояние:

00	I
01	X
10	Z
11	ZX

Одним из ограничений квантовой телепортации является необходимость передать результаты измерений первых двух кубитов от Алисы к Бобу по классическому каналу (5 шаг). Поэтому весь протокол выполняется не быстрее скорости света.

Приведем данную схему:

```
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister

# создаем необходимые регистры
qr = QuantumRegister(3, name="qr")
crz = ClassicalRegister(1, name="crz")
crx = ClassicalRegister(1, name="crx")

# создаем схему
qc = QuantumCircuit(qr, crz, crx)

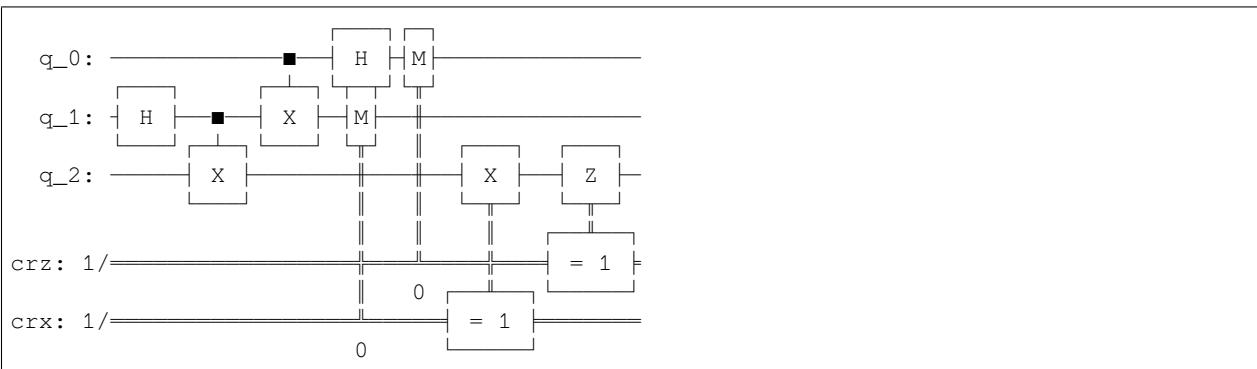
# Шаг 1
qc.h(qr[1])
# Шаг 2
qc.cx(qr[1], qr[2])

# Шаг 3
qc.cx(qr[0], qr[1])
# Шаг 4
qc.h(qr[0])

# Шаг 5 - измеряются 2 кубита Алисы, чтобы передать результат Бобу
qc.measure(qr[0], crz)
qc.measure(qr[1], crx)

# Шаг 6 - применяются гейт X и гейт Z в зависимости от того, какое из измерений дает результат 1.
qc.x(qr[2]).c_if(crx, 1)
qc.z(qr[2]).c_if(crz, 1)

qc.draw()
```



Лекция и реализация квантовой телепортации в Qiskit

CHAPTER
THIRTYFOUR

КВАНТОВОЕ СВЕРХПЛОТНОЕ КОДИРОВАНИЕ

Сверхплотное кодирование (Superdense coding) – простое, но при этом довольно неожиданное приложение квантовой механики. Оно комбинирует определенным нетривиальным образом большинство основных идей этой физической теории, и потому является отличным примером решения задач обработки информации с помощью квантовой механики.

В задаче о сверхплотном кодировании рассматриваются два действующих лица – Алиса и Боб. Они могут находиться, вообще говоря, далеко друг от друга. Цель – передать некоторое количество классической информации от Алисы к Бобу. Пусть у Алисы есть два классических бита информации, которые она хочет передать Бобу, но Алисе разрешено переслать Бобу только один кубит. Достижима ли эта цель?

Задача о сверхплотном кодировании в некотором смысле противоположна задаче о *квантовой телепортации*:

квантовая телепортация	сверхплотное кодирование
передача одного кубита с помощью двух классических бит	передача двух классических бит с помощью одного кубита

Ответ на вопрос выше – утвердительный. Пусть изначально у Алисы и Боба есть пара кубитов в запутанном состоянии

$$\psi = \beta_{00} = \frac{00 + 11}{\sqrt{2}}$$

(это первое состояние Белла (33.1) из четырёх, перечисленных в предыдущей лекции о квантовой телепортации). Вначале первый кубит находится у Алисы, второй – у Боба. Обратите внимание, что для приготовления состояния ψ Алисе не требовалось присыпать Бобу никаких кубитов. Некое третье действующее лицо могло приготовить запутанное состояние ψ заранее и переслать первый кубит Алисе, второй – Бобу.

Оказывается, Алиса может передать два бита классической информации Бобу, отправив ему свой кубит. Для этого Алиса выполняет следующую процедуру:

1. Если Алиса хочет передать Бобу битовую строку “00”, она ничего не делает со своим кубитом.
2. Если Алиса хочет передать битовую строку “01”, она применяет оператор X (гейт NOT) к своему кубиту.
3. Если Алиса хочет передать битовую строку “10”, она применяет оператор Z к своему кубиту.
4. Если Алиса хочет передать битовую строку “11”, она применяет оператор $ZX = iY$ к своему кубиту.

В результате получаются состояния двухкубитной системы справа от стрелок в следующих формулах (проверьте

это!):

$$\begin{aligned} 00 : \psi &\rightarrow \frac{00 + 11}{\sqrt{2}} = \beta_{00}, \\ 01 : \psi &\rightarrow \frac{10 + 01}{\sqrt{2}} = \beta_{01}, \\ 10 : \psi &\rightarrow \frac{00 - 11}{\sqrt{2}} = \beta_{10}, \\ 11 : \psi &\rightarrow \frac{01 - 10}{\sqrt{2}} = \beta_{11}, \end{aligned}$$

т.е. как раз четыре состояния Белла. Они образуют ортонормированный базис в пространстве состояний двухкубитной системы (убедитесь в этом самостоятельно). Следовательно, эти четыре состояния можно различить подходящим измерением для двухкубитной системы. Если Алиса отправит свой кубит Бобу, то Боб после измерения двухкубитной системы в базисе Белла сможет определить, которую из четырех возможных битовых строк Алиса хотела отправить.

Бобу для декодирования удобно было бы применить некий двухкубитный унитарный оператор, отображающий базис Белла в вычислительный базис (чтобы в итоге выполнить измерение в вычислительном базисе). Для этого можно применить оператор $CNOT$ (первый кубит – контрольный) и затем оператор Адамара H для первого кубита, как показано в пунктирном блоке в нижнем правом углу на общей схеме:

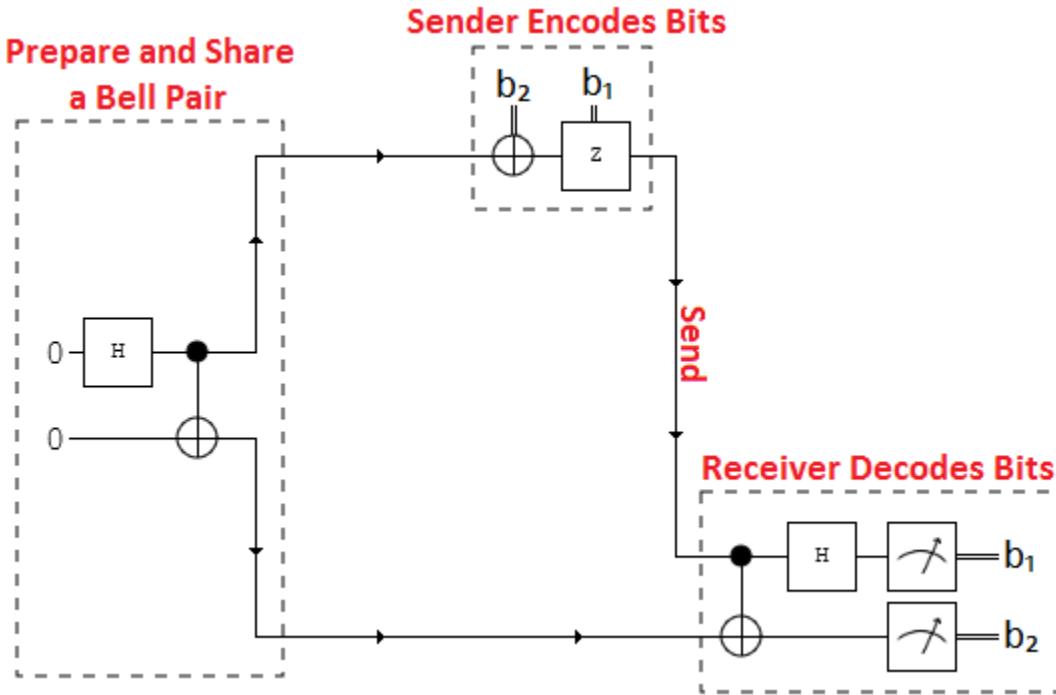


Fig. 34.1: Общая схема для решения задачи о сверхплотном кодировании

В качестве упражнения проверьте самостоятельно, что произведение операторов $(H \otimes I)CNOT$ действительно переводит базис $\{\beta_{00}, \beta_{01}, \beta_{10}, \beta_{11}\}$ в базис $\{00, 01, 10, 11\}$.

Приведем код на qiskit.

```
from qiskit import QuantumCircuit
```

(continues on next page)

(continued from previous page)

```

qc = QuantumCircuit(2)

# готовим состояние beta_{00} из стартового состояния |00>:
qc.h(0)
qc.cx(0, 1)

# визуальный разделитель при отрисовке схемы:
qc.barrier()

# выбираем какое-либо сообщение из набора в assert-выражении ниже:
message = '10'

assert message in ('00', '01', '10', '11')

# Алиса выполняет кодирование сообщения:
if message[1] == "1":
    qc.x(0)
if message[0] == "1":
    qc.z(0)

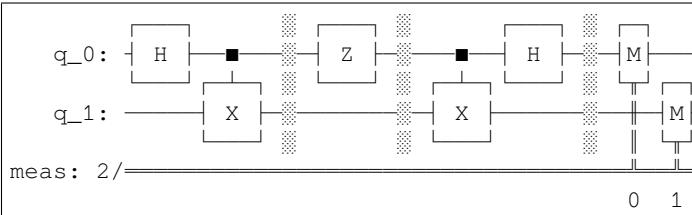
qc.barrier()

# Боб декодирует сообщение:
qc.cx(0, 1)
qc.h(0)

# Боб измеряет двухкубитную систему целиком (в вычислительном базисе):
qc.measure_all()

qc.draw()

```



Запускаем схему на симуляторе:

```

from qiskit import Aer

aer_sim = Aer.get_backend('aer_simulator')
result = aer_sim.run(qc).result()
counts = result.get_counts(qc)

print(
    f"message was '{message}' -> the measurement result is {counts}"
    " (<- NOTE: the keys are little-endian!)"
)

```

```
message was '10' -> the measurement result is {'01': 1024} (<- NOTE: the keys are little-endian!)
```

RTFM!

Обратите внимание на [документацию](#) к `qiskit.result.Result.get_counts!`. Ключи возвращаемого словаря – это битовые строки в **little-endian** формате, т.е. кубит с индексом [0] находится справа в битовой строке.

Инициализируем переменную `message` каждым из четырех значений и получаем следующие результаты:

```
message was '00' -> the measurement result is {'00': 1024}
message was '01' -> the measurement result is {'10': 1024}
message was '10' -> the measurement result is {'01': 1024}
message was '11' -> the measurement result is {'11': 1024}
```

Резюмируем. Алиса, взаимодействуя только с одним кубитом, смогла передать два классических бита информации Бобу. Конечно, в процедуре участвовало два кубита, но Алисе не требовалось взаимодействовать со вторым кубитом. Если бы Алиса пересыпала Бобу один классический бит вместо одного кубита, то передать таким способом два классических бита было бы невозможно, разумеется.

Существуют более впечатляющие, чем сверхплотное кодирование, примеры применения квантовой механики к задачам обработки информации. Но на примере сверхплотного кодирования виден важнейший принцип – **информация физична**. Нетривиальные физические теории вроде квантовой механики могут привести к неожиданным новым возможностям в обработке информации.

Напоследок рассмотрим вопрос безопасности коммуникации. Допустим, некий третий субъект (пусть с именем Ева) хочет “подслушать” сообщение, передаваемое Бобу. Ева перехватывает кубит, отправленный Алисой, и может попытаться извлечь какую-либо полезную информацию, выполнив какое-либо измерение над кубитом Алисы. Пусть Ева использует положительно-определенный оператор E для первого кубита и вычисляет величину $\xi|E \otimes I|\xi$, где ξ – одно из состояний $\{\beta_{00}, \beta_{01}, \beta_{10}, \beta_{11}\}$. Кубит Боба (второй кубит) Еве недоступен, поэтому мы поставили единичный оператор на второе место в составном операторе. Любое возможное состояние ξ можно записать в виде

$$\xi = \frac{\phi_1 + \phi_2}{\sqrt{2}},$$

где ϕ_1, ϕ_2 – двухкубитные состояния, соответствующие вычислительному базису (см. формулы для β_{ij} выше). Например, $\phi_1 = 01, \phi_2 = -10$ для случая $\xi = \beta_{11}$ и т.д.

$$2\xi|E \otimes I|\xi = \phi_1|E \otimes I|\phi_1 + \phi_2|E \otimes I|\phi_2 + \phi_1|E \otimes I|\phi_2 + \phi_2|E \otimes I|\phi_1.$$

В каждом из четырех возможных случаев $\phi_1|E \otimes I|\phi_2 = \phi_2|E \otimes I|\phi_1 = 0$, потому что второй кубит имеет противоположные значения в ϕ_1 и ϕ_2 . Вычислив $\phi_i|E \otimes I|\phi_i$ для $i = 1, 2$ (первые два слагаемых в правой части), легко убедиться, что

$$2\xi|E \otimes I|\xi = 0|E|0 + 1|E|1$$

во всех случаях. Таким образом, Ева всегда будет получать один и тот же результат своего измерения, не зависящий от сообщения, которое Алиса пыталась передать Бобу. Ева не сможет получить никакой полезной информации о сообщении от Алисы, перехватив её кубит.

[Лекция и реализация процедуры сверхплотного кодирования в Qiskit](#)

CHAPTER
THIRTYFIVE

АЛГОРИТМ ГРОВЕРА

Одно из самых востребованных действий в работе с данными – поиск по базе данных. При использовании классического компьютера такой поиск в худшем случае требует N операций, где N – количество строк в таблице. В среднем найти нужный элемент можно за $N/2$ операций.

Фактически, это означает, что если мы не знаем, где расположен нужный элемент в таблице, то придется перебирать все элементы, пока не найдем то, что нужно. Для классических вычислений это нормально, но что, если у нас есть квантовый компьютер?

Если наша база данных работает на основе квантовых вычислений, то мы можем применить алгоритм Гровера, и тогда такой поиск потребует всего порядка \sqrt{N} действий. Конечно же, такое ускорение не будет экспоненциальным, как при использовании некоторых других квантовых алгоритмов, но оно будет квадратичным, что также довольно неплохо.

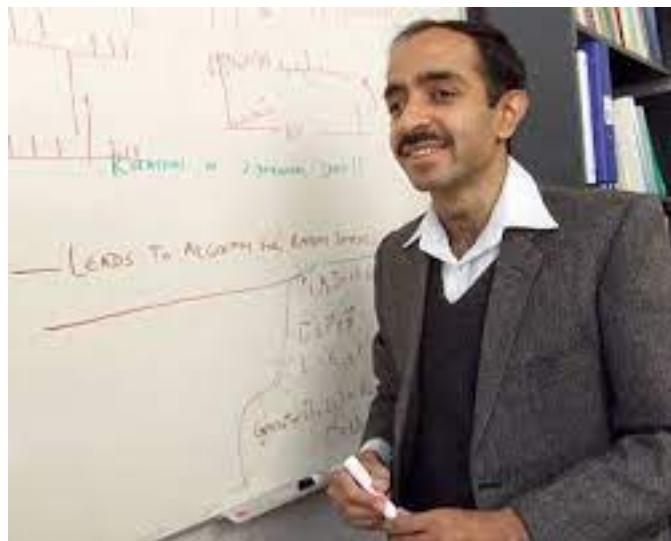


Fig. 35.1: Лов Гровер

Лов Гровер

Индо-американский ученый в сфере Computer Science Лов Кумар Гровер предложил квантовый алгоритм поиска по базе данных в 1996 году. Этот алгоритм считается вторым по значимости для квантовых вычислений после алгоритма Шора. Впервые он был реализован на простейшем квантовом компьютере в 1998 году, а в 2017 году алгоритм Гровера был впервые запущен для трехкубитной базы данных.

Итак, наша задача состоит в том, что мы должны найти идентификационный номер (*Id*) записи, которая

удовлетворяет определенным условиям. Функция-оракул находит такую запись (для простоты будем сначала считать, что такая запись одна) и помечает соответствующий ей Id . Отметка делается достаточно оригинальным способом: Id умножается на -1 .

Для полной ясности соотнесем количество Id с числом кубитов в квантовой схеме. Здесь все очень просто: имея n кубитов, можно закодировать $N = 2^n$ идентификаторов. К примеру, если в таблице базы данных 1024 записей, то закодировать все Id можно с помощью десяти кубитов.

Для того, чтобы не запутаться в квантовых операциях, рассмотрим пример поменьше: с помощью двух кубитов закодируем четыре идентификационных номера, один из которых будет помечен функцией-оракулом как искомый – он будет домножен на -1 . Все эти четыре числа могут существовать в квантовой схеме одновременно, если кубиты приведены в состояние суперпозиции.

Пусть искомый Id равен 11 (будем пользоваться двоичной системой и вести счет с нуля), тогда после работы функции-оракула мы будем иметь 4 состояния: $|00\rangle$, $|01\rangle$, $|10\rangle$, $-|11\rangle$. Проблема в том, что если измерить эту схему, то с равной вероятностью будет обнаружено одно из этих четырех значений, но узнать, какое из них функция-оракул пометила минусом, будет невозможно.

Получается, что одной функции-оракула недостаточно, нужно что-то дополнительное. На помощь приходит алгоритм Гровера. Правда, у него есть такая особенность – он является итерационным, то есть определенные операции (в том числе и применение функции-оракула) нужно повторить несколько раз (порядка \sqrt{N}). Причем, с количеством итераций нельзя ошибиться, иначе алгоритм даст неправильный ответ.

В идеале после всех итераций квантовую схему можно будет измерить и получить значение Id искомой записи в таблице базы данных.

Разберем операции, которые включает в себя каждая итерация, но перед этим добавим в схему еще один кубит, который мы будем называть вспомогательным. Он нужен для хранения метки искомого индекса. Звучит не совсем понятно, но ничего сложного в этом нет, все станет ясным после разбора работы функции-оракула. Итак, наша база данных двухкубитная, но сама схема состоит из трех кубитов.

Квантовая схема выглядит так:

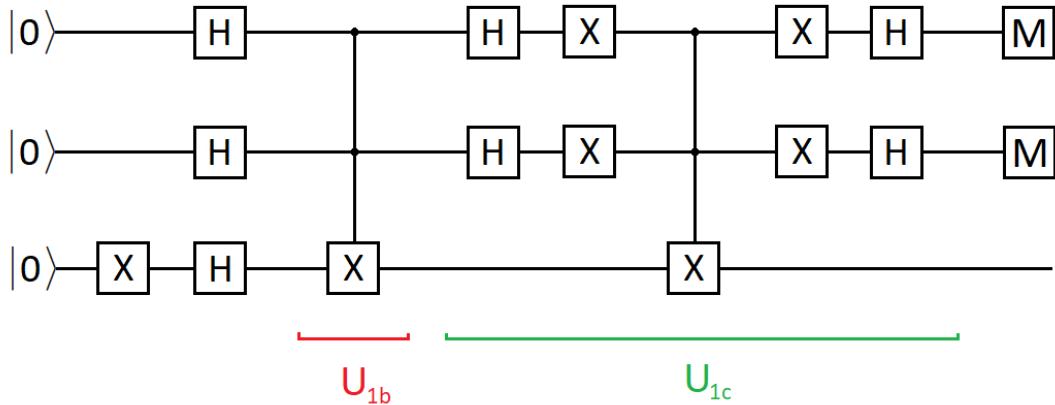


Fig. 35.2: Алгоритм Гровера для $n=2$ (искомый индекс 11).

В самом начале, еще до всех итераций, все кубиты (включая вспомогательный) должны быть приведены в состояние суперпозиции с помощью оператора Адамара. Причем начальное состояние всех кубитов должно быть равно 0, кроме вспомогательного кубита – до действия оператора Адамара он должен быть приведен в состояние 1.

Таким образом, вспомогательный кубит после применения оператора Адамара будет находиться в состоянии $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, тогда как остальные кубиты примут состояние $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

Далее начинаем итерации. Каждая итерация состоит из двух частей. Первая часть – это действие функции-оракула. Это некоторая функция, умеющая эффективно определять, какой индекс соответствует искомому объекту. Эта функция не может сообщить нам этот индекс напрямую, зато она может пометить его минусом.

Для разбора внутренней работы алгоритма нам потребуется задать функцию-оракул вручную и сделать ее достаточно простой, поэтому нужно знать, что в рабочих условиях она будет действовать похоже, но будет устроена, скорее всего, по-другому, так как предназначена для конкретной задачи выбора искомых данных. Мы не будем касаться вопроса конкретной реализации функции-оракула для выбора определенных данных, так как это уже другой вопрос, не влияющий на принцип алгоритма Гровера.

Для того чтобы понять алгоритм Гровера, мы должны будем понять, какие изменения происходят с состояниями кубитов до того момента, когда производится измерение, выдающее искомый индекс.

Мы договорились, что в нашей учебной задаче искомый Id равен 11, так что в результате измерения мы должны получить именно это значение. Смоделируем оракул, который будет помечать этот индекс. В качестве такого оракула подойдет гейт Тоффоли ($CCNOT$). При подаче на оба его управляющих входа значений 1, он будет применять к управляемому кубиту (это как раз будет вспомогательный кубит) гейт X .

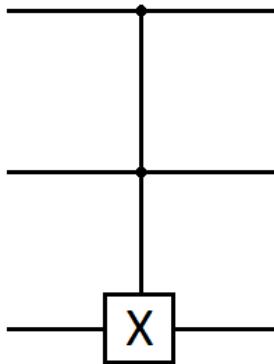


Fig. 35.3: Гейт Тоффоли.

На состояния верхних кубитов, кодирующих индексы 00, 01 и 10 гейт Тоффоли не будет реагировать, и вспомогательный кубит будет находиться в состоянии $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

Но при срабатывании гейта на индексе 11 к вспомогательному кубиту применится оператор X , так что вспомогательный кубит примет состояние $\frac{1}{\sqrt{2}}(|1\rangle - |0\rangle)$, или, если это состояние записать по-другому, за скобками появится минус: $-\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

Этот минус относится не только к вспомогательному кубиту, но и ко всему состоянию, соответствующему индексу 11, так что можно считать, что вспомогательный кубит остался неизменным, и отнести минус к состоянию 11 верхних кубитов. Таким образом, индекс 11 помечен минусом как искомый. Другими словами, функция-оракул перевела состояние $|11\rangle|q_{helper}\rangle$ в состояние $-|11\rangle|q_{helper}\rangle$, где $|q_{helper}\rangle$ – вспомогательный кубит.

Запишем состояние квантовой схемы после применения оракула (состояние вспомогательного кубита – скобка справа):

$$|\psi\rangle = \frac{1}{2\sqrt{2}}(|00\rangle + |01\rangle + |10\rangle - |11\rangle)(|0\rangle - |1\rangle)$$

Итак, первая часть первой итерации завершена. Искомый индекс помечен, но если измерить кубиты прямо сейчас, то это ничего не даст – минус не проявится при измерении. Да и сам индекс 11 будет получен только с вероятностью 0.25 – такой же, как и у других индексов.

Для того, чтобы лучше понять дальнейшие действия, представим первую половину работы алгоритма в виде рисунка, показывающего вектор текущего состояния. В качестве единичного вектора горизонтальной оси мы будем использовать все состояния из суперпозиции кроме того, который соответствует искомому индексу, а вертикальным единичным вектором будет искомый вектор.

Вектор c – состояние системы перед первой итерацией – является линейной комбинацией векторов, соответствующим горизонтальной и вертикальной осям.

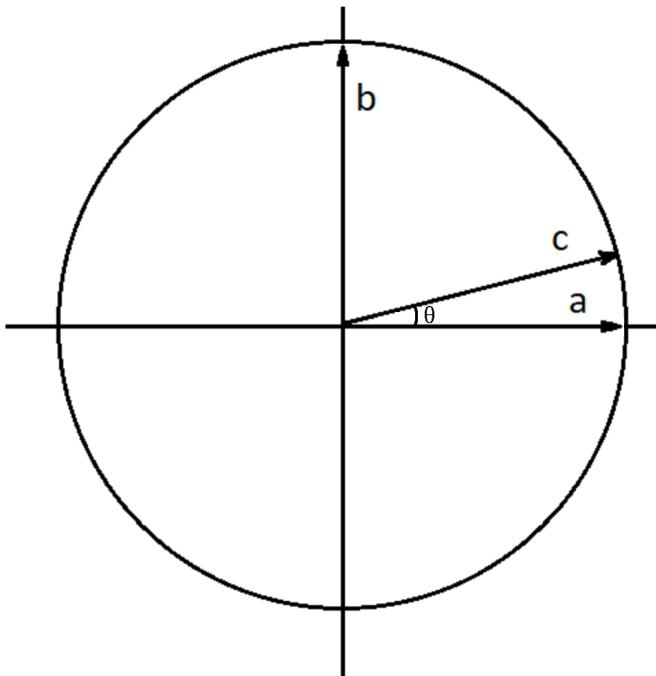


Fig. 35.4: Состояние системы перед первой итерацией.

Можно выразить вектор c для нашего случая (системы из двух кубитов с искомым индексом 11), обозначив его координаты за x и y :

$$\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = x \frac{1}{\sqrt{3}}(|00\rangle + |01\rangle + |10\rangle) + y|11\rangle$$

Из данного уравнения находим $x = \frac{\sqrt{3}}{2}$ и $y = \frac{1}{2}$.

По этим координатам можно понять, что угол между вектором c и горизонтальной осью (обозначим его как θ) равен $\frac{\pi}{6}$. Если забежать немного вперед, то можно сказать, что наша цель – добиться, чтобы текущее состояние дошло до $\frac{\pi}{2}$ (или хотя бы приблизительно), то есть почти или полностью равнялось искомому состоянию, так что после измерения можно было его и получить с высокой вероятностью.

Координаты текущего вектора состояния можно записать через угол θ :

$$x = \cos \theta$$

$$y = \sin \theta$$

На всякий случай нужно уточнить, что вспомогательный кубит не отражается на рисунке с окружностью, так как он не предназначен для обозначения индекса, а только хранит в себе его метку.

После применения функции-оракула текущий вектор отразится относительно горизонтальной оси. Объясняется это очень легко – его вертикальная компонента (вектор $|11\rangle$) становится отрицательной.

Вектор c_{1b} – это отражение вектора c на угол θ вниз относительно горизонтальной оси:

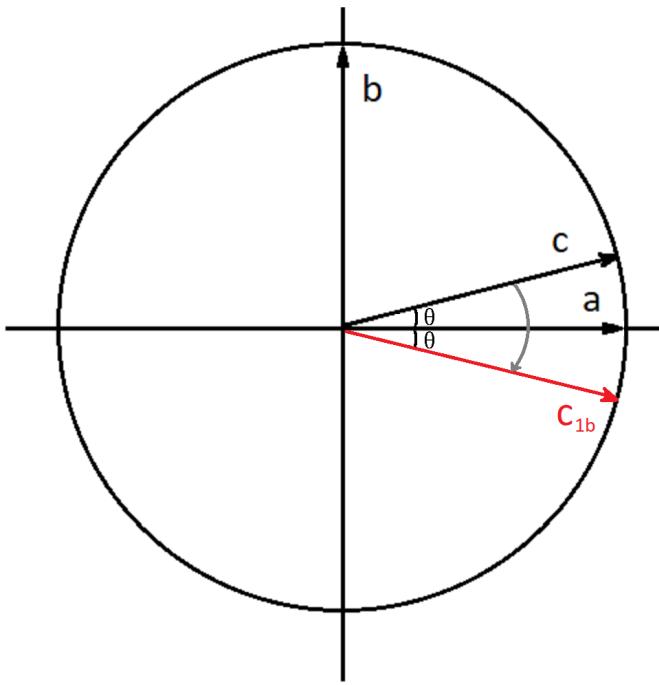


Fig. 35.5: Состояние системы после первой части первой итерации.

Такое отражение в нашем примере производится с помощью операции $CCNOT$, но в общем случае операция выглядит так:

$$U_{1b} = I - 2|b\rangle\langle b|$$

Функцию-оракул мы здесь обозначили как U_{1b} . Она меняет знак только для вертикальной составляющей вектора состояния, поэтому и происходит отражение.

Проверим формулу в действии, применив ее для нашего примера:

$$U_{1b}|c\rangle = (I - 2|11\rangle\langle 11|)\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle - 2|11\rangle) = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle)$$

И наконец приступаем к разбору второй части первой итерации. В ней будет происходить еще одно отражение вектора, но уже не относительно горизонтальной оси, а относительно вектора c . Нетрудно заметить, что при этом текущий вектор состояния станет равен $\cos 3\theta|a\rangle + \sin 3\theta|b\rangle$.

Операция для получения вектора c_{1c} будет выглядеть так:

$$U_{1c} = 2|c\rangle\langle c| - I$$

Посчитаем, чему равен вектор c_{1c} для нашего примера:

$$U_{1c}|c_{1b}\rangle = (2|c\rangle\langle c| - I)\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle) = |11\rangle$$

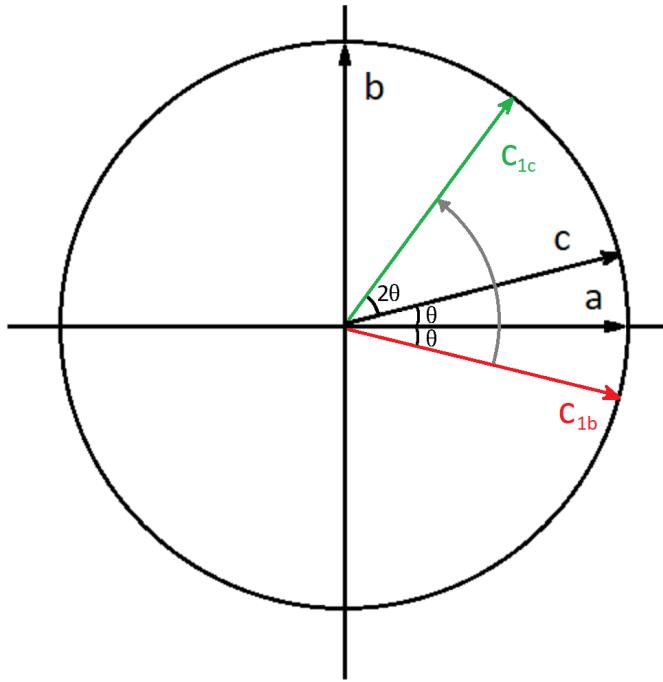


Fig. 35.6: Состояние системы после второй части первой итерации.

Произошло отражение текущего вектора состояния $|c_{1b}\rangle$ относительно вектора $|c\rangle$. Если представить $|c_{1b}\rangle$ как $k_1|c\rangle + k_2|c_\perp\rangle$, где $|c_\perp\rangle$ – вектор, перпендикулярный $|c\rangle$, а k_1 и k_2 – действительные коэффициенты, то тогда отраженный вектор будет равен $k_1|c\rangle - k_2|c_\perp\rangle$.

В нашей квантовой схеме эта часть итерации реализована таким образом:

Вначале применяется оператор Адамара для первого и второго кубитов. Это упрощает нашу задачу, так как теперь отразить вектор состояния нужно не относительно состояния суперпозиции $\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$, а относительно состояния $|00\rangle$.

Далее требуется отразить вектор, то есть всем состояниям кроме нулевого присвоить минус, но мы сделаем проще: присвоим минус нулевому состоянию $|00\rangle$, а остальные состояния, составляющие суперпозицию, оставим как есть. Сделаем это, последовательно применив гейты X и $CCNOT$. После этого вернемся в исходную “систему координат”, применив операции в обратном порядке: сначала X , а потом H .

Из-за применения такого лайфхака (присвоения минуса нулевому состоянию) мы в нашем двухкубитном примере получим результат с точностью до общей фазы: не $|11\rangle$, а $-|11\rangle$. Но это не страшно, так как после измерения мы все равно увидим искомое значение индекса.

По рисунку, изображающему на окружности состояние системы после второй части первой итерации, видно, что в общем случае каждая последующая итерация будет приближать текущий вектор к вертикальному. Но в нашем случае угол между вектором состояния и горизонтальной осью после окончания первой итерации равен 3θ , то есть это уже и есть желаемый угол $\frac{\pi}{2}$.

В общем случае этот угол равен $(2t + 1)\theta \approx \frac{\pi}{2}$, где t – номер произведенной итерации. Отсюда можно вывести число итераций, необходимое для работы алгоритма. При большом требуемом количестве итераций t и фиксированном $K = 1$ (для $K > 1$ вывод формулы аналогичный) угол θ будет близок к 0, так что его можно

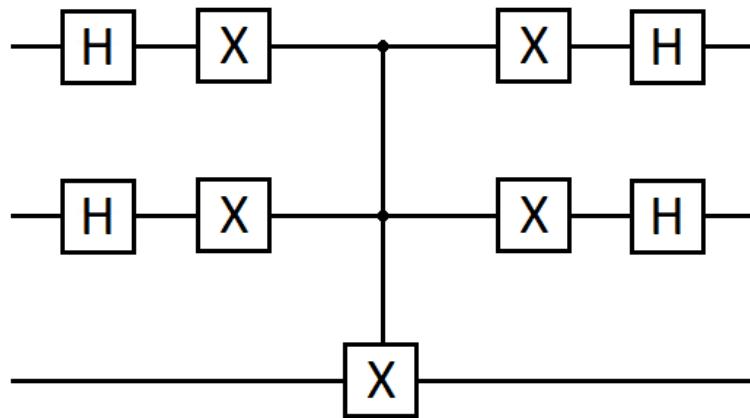


Fig. 35.7: Вторая часть итерации.

заменить на $\sin \theta$, который, в свою очередь, равен $\frac{1}{\sqrt{N}}$:

$$(2t + 1)\theta \rightarrow (2t + 1) \frac{1}{\sqrt{N}} \approx \frac{\pi}{2}$$

Если пренебречь единицей в скобках на основании того, что t – большое число, можно найти, что t приблизительно равно $\frac{\pi\sqrt{N}}{4}$.

Мы уже разобрались, что каждая итерация состоит из двух частей. Первая часть – отражение вниз относительно горизонтальной оси. Вторая часть – отражение вверх относительно изначального состояния, то есть вектора c . Вектор состояния всегда будет отражаться вверх на больший угол, чем в первой части итерации. Этим и будет обеспечиваться его постепенное приближение в вертикальной оси.

Мы разобрали случай, когда требуется найти один объект в таблице. Если же потребуется найти несколько объектов, то тогда, обозначив их количество за K , мы должны будем проделать около $\frac{\pi}{4} \sqrt{\frac{N}{K}}$ итераций. Таким образом, для успешной работы алгоритма Гровера необходимо знать число K , чтобы можно было найти через него угол θ , а затем число итераций.

35.1 Реализация алгоритма Гровера

Итак, мы разобрали общие принципы алгоритма Гровера, а также рассмотрели конкретный пример. Настало время написать для этого примера программу.

Для начала импортируем все необходимые библиотеки и создадим схему из трех кубитов:

```
import pennylane as qml
from pennylane import numpy as np

dev = qml.device('default.qubit', shots=1, wires=3)
```

Начальная функция, создающая суперпозицию для каждого кубита:

```
def U_start():
    qml.PauliX(wires=2)
    for i in range(3):
        qml.Hadamard(wires=i)
```

Создадим функцию, действующую аналогично оракулу (первая часть итерации). Эта функция помечает значение индекса 11:

```
def U_b():
    qml.Toffoli(wires=[0, 1, 2])
```

Вторая часть итерации:

```
def U_c():
    for i in range(2):
        qml.Hadamard(wires=i)
        qml.PauliX(wires=i)

    qml.Toffoli(wires=[0, 1, 2])
    for i in range(2):
        qml.PauliX(wires=i)
        qml.Hadamard(wires=i)
```

Объединим первую и вторую часть итерации в одну функцию:

```
def U_iteration():
    U_b()
    U_c()
```

Переходим к итоговой функции, содержащей все шаги, а также производящей измерение кубитов в конце. В аргументе N мы должны указать количество итераций:

```
@qml.qnode(dev)
def circuit(N: int):
    U_start()
    for t in range(N):
        U_iteration()
    return qml.probs(wires=[0, 1])
```

Запускаем функцию и выведем ее результат:

```
print(circuit(N=1))
```

```
[0. 0. 0. 1.]
```

Так как в качестве искомого индекса выступало значение 11, то в результате запуска функции мы должны получить массив, состоящий из вероятностей каждого индекса, в котором искомый индекс (в нашем примере он будет последним в массиве) должен иметь наибольшую вероятность. Параметр устройства `shots` при необходимости можно увеличивать, не забывая о том, что его увеличение будет кратно замедлять алгоритм. Таким образом, мы нашли с помощью алгоритма Гровера искомый индекс.

Алгоритм Гровера может применяться не только для задач простого поиска в базе данных, но и как дополнительное средство ускорения для поиска экстремума целочисленной функции, а также для поиска совпадающих строк в базе данных, так что этот алгоритм, как и его модификации, сможет быть полезным в разнообразных задачах Data Science.

35.2 Задание

1. Распишите операторы U_{1b} и U_{1c} из примера в виде матриц 4×4 и проведите расчеты для получения c_{1b} и c_{1c} в виде векторов-столбцов.
2. Модифицируйте приведенный выше код алгоритма Гровера для двухкубитной базы данных так, чтобы искомый индекс соответствовал состоянию $|00\rangle$.

АЛГОРИТМ САЙМОНА

36.1 Задача Саймона

Давайте начнём с того, что алгоритм Саймона решает задачу Саймона. Да, вот такой замечательный учёный - нашёл проблему, решил проблему.

По своей природе задача Саймона является разновидностью задачи о скрытой абелевой подгруппе [Lom04].

Пусть задана функция $f : \{0, 1\}^n \rightarrow \{0, 1\}^n$ и **неизвестная** строка $s \in \{0, 1\}^n$, для всех $x, y \in \{0, 1\}^n$ выполняется:

$$f(x) = f(y) \Leftrightarrow x \oplus y = s$$

То есть, если мы для двух различных строк x и y имеем одинаковое значение $f(x) = f(y)$, то $x \oplus y$ равняется некоторой неизвестной строке s . Функция $f(x)$ представляет собой чёрный ящик.

Задача состоит в том, чтобы **найти s , выполнив при этом как можно меньшее количество вызовов $f(x)$** .

36.2 Описание алгоритма

Сразу определим операцию на двух бинарных строках (они же бинарные векторы) $x = x_0x_1x_2\dots x_{2^n-1}$, $z = z_1z_2z_3\dots z_{2^n-1}$:

$$\langle x, z \rangle = \bigoplus_{i=0}^{2^n-1} x_i \wedge z_i = x_0 \wedge z_0 \oplus x_1 \wedge z_1 \oplus \dots \oplus x_{2^n-1} \wedge z_{2^n-1}$$

Принципиальная схема алгоритма Саймона:

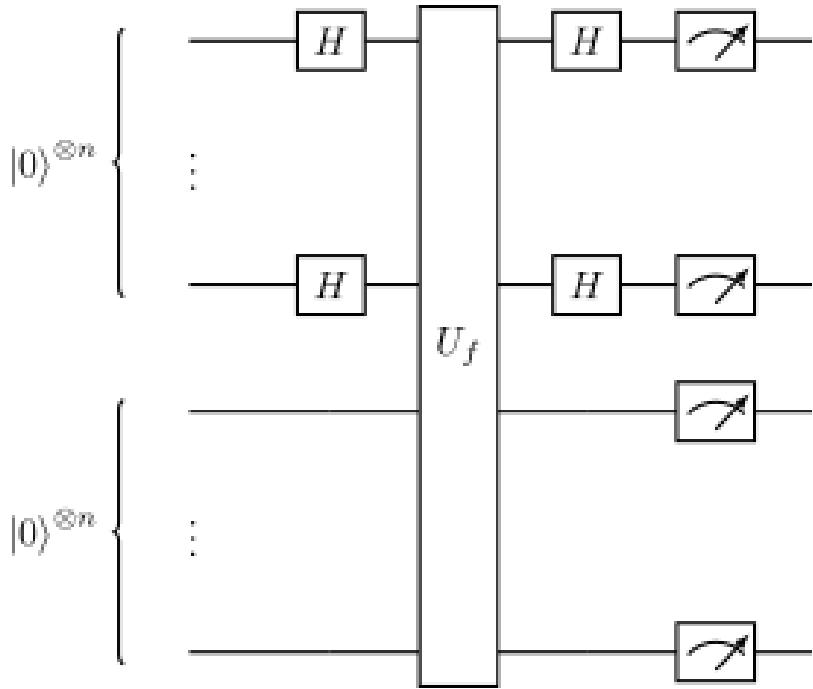
В принципе, есть небольшие вариации в реализации алгоритма, мы рассмотрим наиболее простую (все необходимые ссылки в конце приведены):

1. Сначала приготовления. Вначале мы приготовим 2 набора квантовых регистров (каждый размерности n) в следующем состоянии:

$$|\psi_0\rangle = |0\rangle|0\rangle$$

2. Применяем гейты Адамара на первом регистре:

$$(H^n \otimes I^n)|\psi_0\rangle = (H^n \otimes I^n)|0\rangle|0\rangle = |\psi_1\rangle = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle|0\rangle$$



3. Применяем оператор U_f , который выполняет следующее преобразование $U_f(|x\rangle|0\rangle) = |x\rangle|f(x)\rangle$, т.е. во второй регистр записывается значение функции $f(x)$:

$$U_f(|\psi_1\rangle) = U_f\left(\frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle|0\rangle\right) = |\psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle|f(x)\rangle$$

4. Производим измерение на втором регистре. Измеренное значение будет соответствовать либо x либо $y = x \oplus s$. А первый регистр примет значение:

$$|\psi\rangle_3 = \frac{1}{\sqrt{2}} (|x\rangle_1 + |y\rangle_1)$$

5. Снова применяем гейты Адамара на первом регистре:

$$(H^n \otimes I^n)|\psi_2\rangle = |\psi_3\rangle = \frac{1}{2^n} \sum_{z \in \{0,1\}^n} (-1^{\langle x, z \rangle} + -1^{\langle y, z \rangle})|z\rangle,$$

(операцию $\langle x, z \rangle$ мы уже определяли выше).

Note: Для чисел $x = 110111$ и $z = 010101$ получим

$$\langle x, z \rangle = 1 \wedge 0 \oplus 1 \wedge 1 \oplus 0 \wedge 0 \oplus 1 \wedge 1 \oplus 1 \wedge 0 \oplus 1 \wedge 1 = 1$$

$$(-1)^1 = -1$$

6. Производим измерение на первом регистре. И здесь возможны 2 варианта исхода:

1. $x \oplus y = s = 0^n$

Учитывая $x \oplus y = 0^n$ и то, что состояние всей системы описывается как

$$\sum_{z \in \{0,1\}^n} |z\rangle \otimes \frac{1}{2^n} \sum_{x \in \{0,1\}^n} (-1^{\langle x, z \rangle}) |f(x)\rangle$$

находим вероятность получить строку z на первом регистре

$$p_z = \left\| \frac{1}{2^n} \sum_{z \in \{0,1\}^n} ((-1)^{\langle z, x \rangle} |f(x)\rangle) \right\|^2 = \frac{1}{2^n}$$

т.е. в данном случае имеет место **равномерное распределение**.

2. $x \oplus y = s \neq 0^n$

Этот случай гораздо интереснее. Функция f преобразует два различных входных значения $x_1, x_2 \in \{0,1\}^n$ в одно $f(x_1) = f(x_2) = s \in \{0,1\}^n$. Также $x_1 \oplus x_2 = s$ эквивалентно $x_1 \oplus s = x_2$.

$$|\psi_3\rangle = \frac{1}{2^n} \sum_{z \in \{0,1\}^n} \sum_{x \in \{0,1\}^n} \frac{(-1)^{\langle z, x \rangle} (1 + (-1)^{\langle z, s \rangle})}{2} |z\rangle \oplus |f(x)\rangle =$$

$$\frac{1}{2^n} \sum_{z \in \{0,1\}^n} |z\rangle \otimes \sum_{x \in \{0,1\}^n} \frac{(-1)^{\langle z, x \rangle} (1 + (-1)^{\langle z, s \rangle})}{2} |f(x)\rangle$$

$$p_z = \left\| \frac{1}{2^n} \sum_{x \in \{0,1\}^n} ((-1)^{\langle x, z \rangle} |f(x)\rangle) \right\|^2 = \left\| \frac{1}{2^n} \sum_{z \in A} (((-1)^{\langle x_1, z \rangle} + (-1)^{\langle x_2, z \rangle}) |z\rangle) \right\|^2 \\ = \begin{cases} \frac{1}{2^{n-1}}, & \text{если } \langle z, s \rangle = 0 \\ 0, & \text{если } \langle z, s \rangle = 1 \end{cases}$$

Расчёты вероятностей можно найти в приложениях [Приложение 1](#).

Выполняем алгоритм n раз. После чего у нас будет система n линейно независимых уравнений. Теперь приступаем к вычислению строки s .

7. Постобработка.

Итак, для того, чтобы найти $s = (s_0, s_1, s_2, \dots, s_{n-1})^T$, нам потребуется n линейно независимых векторов \vec{z}_i , для которых выполняется $\langle \vec{z}_i, s \rangle = 0$.

После того как получена система из n линейно независимых уравнений, решение можно найти методом Гаусса.

36.3 Пример

Давайте возьмём $n = 3$, строку $s = 100$, и функцию f , которая соответствует критерию

$$f(x) = f(y) \Leftrightarrow x \oplus s = y$$

Обычно функция $f(x)$ задана наперёд. Ну а мы выберем её простейшей: $f(x) = x \oplus s$.

Давайте посмотрим на таблицу истинности всех нужных переменных.

x	$x \oplus s$	$f(x)$
000	100	000
001	101	001
010	110	010
011	111	011
100	000	000
101	001	001
110	010	010
111	011	011

Нарисуем схему на Qiskit, которая будет показывать одну итерацию алгоритма:

```
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister

# Работаем в пространстве размерности n = 3.
n = 3

# Создаём необходимые регистры
qr1 = QuantumRegister(n, name="q1")
qr2 = QuantumRegister(n, name="q2")
cr1 = ClassicalRegister(n, name="c1")

# Шаг 1. Создаём квантовую схему (инициализация состояния)
qc = QuantumCircuit(qr1, qr2, cr1)

# Шаг 2. Применяем гейт Адамара ко всем кубитам первого регистра
qc.h(range(n))

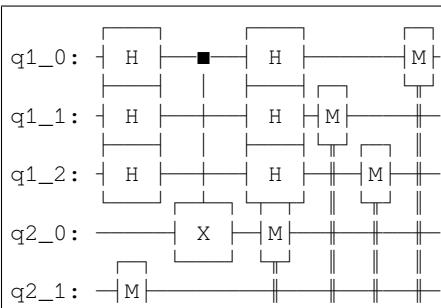
# Шаг 3. Применяем U_f
qc.cx(qr1[0], qr2[0])

# Шаг 4. Производим измерение первого регистра
qc.measure(qr2, cr1)

# Шаг 5. Ещё раз применяем гейт адамара к каждому из кубитов
qc.h(range(n))

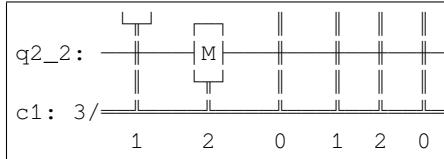
# Шаг 6. Производим измерение первого регистра
qc.measure(qr1, cr1)

# Рисуем схему
qc.draw()
```



(continues on next page)

(continued from previous page)



Теперь пройдём по всем шагам алгоритма:

- Инициализация всех регистров в 0 состоянии:

$$|\psi_0\rangle = |000\rangle_1|000\rangle_2$$

- Применяем Адамар к первому регистру:

$$\begin{aligned} (H^n \otimes I)(|\psi_0\rangle) = |\psi_1\rangle = \frac{1}{\sqrt{8}}(&|000\rangle + |001\rangle + |010\rangle + |011\rangle \\ &+ |100\rangle + |101\rangle + |110\rangle + |111\rangle)_1|000\rangle_2) \end{aligned}$$

- Применяем оракул U_f :

$$U_f(|\psi_1\rangle) = |\psi_2\rangle =$$

$$\begin{aligned} \frac{1}{\sqrt{8}}(&|000\rangle_1|0 \oplus 0, 0, 0\rangle_2 \\ &+ |001\rangle_1|0 \oplus 0, 0, 0\rangle_2 \\ &+ |010\rangle_1|0 \oplus 0, 0, 0\rangle_2 \\ &+ |011\rangle_1|0 \oplus 0, 0, 0\rangle_2 \\ &+ |100\rangle_1|0 \oplus 1, 0, 0\rangle_2 \\ &+ |101\rangle_1|0 \oplus 1, 0, 0\rangle_2 \\ &+ |110\rangle_1|0 \oplus 1, 0, 0\rangle_2 \\ &+ |111\rangle_1|0 \oplus 1, 0, 0\rangle_2) \end{aligned}$$

- Измеряем второй регистр. С вероятностью $\frac{1}{2}$ мы получим либо значение $|000\rangle$ либо $|100\rangle$. Допустим результат равен $|100\rangle$. Тогда состояние первого регистра примет вид: $|\psi_3\rangle = (|100\rangle_1 + |101\rangle_1 + |110\rangle_1 + |111\rangle_1)$.

- Ещё один раз применяем гейты Адамара на первый регистр:

$$\begin{aligned} |\psi_4\rangle = \frac{1}{4}[&(|0\rangle - |1\rangle) \otimes (|0\rangle + |1\rangle) \otimes (|0\rangle + |1\rangle)) \\ &+ (|0\rangle - |1\rangle) \otimes (|0\rangle + |1\rangle) \otimes (|0\rangle - |1\rangle)) \\ &+ (|0\rangle - |1\rangle) \otimes (|0\rangle - |1\rangle) \otimes (|0\rangle + |1\rangle)) \\ &+ (|0\rangle - |1\rangle) \otimes (|0\rangle - |1\rangle) \otimes (|0\rangle - |1\rangle))] \end{aligned}$$

- Производим измерение первого регистра, записывая результат.

Выполняем шаги 1-6 n раз.

- Здесь решаем систему уравнений.

36.4 Приложение

1. Расчёт вероятностей.

1. Случай, когда $x \oplus z = 0^n$:

$$\sum_{z \in \{0,1\}^n} |z\rangle \otimes \left(\frac{1}{2^n} \sum_{x \in \{0,1\}^n} ((-1)^{\langle x,z \rangle} |f(x)\rangle) \right)$$

Вероятность получить какую-либо из 2^n строк одинакова:

$$p_z = \left\| \frac{1}{2^n} \sum_{z \in \{0,1\}^n} ((-1)^{\langle z,x \rangle} |f(x)\rangle) \right\|^2 = \frac{1}{2^n}$$

Утверждение, данное выше, следует из того, что $f(x)$ отличается от x лишь порядком во всём множестве строк $\{0,1\}^n$:

$$\left\| \frac{1}{2^n} \sum_{z \in \{0,1\}^n} ((-1)^{\langle z,x \rangle} |f(x)\rangle) \right\|^2 = \left\| \frac{1}{2^n} \sum_{z \in \{0,1\}^n} ((-1)^{\langle z,x \rangle} |x\rangle) \right\|^2$$

2. Случай, когда $x \oplus z = s \neq 0^n$:

Определим $A = f(\{0,1\}^n)$ - образ функции f , $f(x_i) = \ell \in A$ - т.е. это какое-то значение функции f . Здесь у нас имеются два таких значения $x_1 \in \{0,1\}^n$, $x_2 \in \{0,1\}^n$, что для них выполняется $x_2 = s \oplus x_1$.

$$p_z = \left\| \frac{1}{2^n} \sum_{\ell \in A} (((-1)^{\langle z,x_1 \rangle} + (-1)^{\langle z,x_2 \rangle}) |\ell\rangle) \right\|^2 = \frac{1}{2^n}$$

Перепишем коэффициенты $(-1)^{\langle x_1, z \rangle} + (-1)^{\langle x_2, z \rangle}$:

$$(-1)^{\langle x_1, z \rangle} + (-1)^{\langle x_2, z \rangle} = (-1)^{\langle x_1, z \rangle} + (-1)^{\langle x_2 \oplus s, z \rangle}$$

Также, заметим, что выполняется $\langle x_1 \oplus s, z \rangle = \langle x_1, z \rangle \oplus \langle x_2, z \rangle$, тогда ещё раз перепишем:

$$(-1)^{\langle x_1, z \rangle} (1 + (-1)^{\langle z, s \rangle})$$

И собирая все полученные условия:

$$p_z = \left\| \frac{1}{2^n} \sum_{\ell \in A} ((-1)^{\langle x_1, z \rangle} (1 + (-1)^{\langle z, s \rangle}) |\ell\rangle) \right\|^2 = \frac{1}{2^n}$$

Если $\langle z, s \rangle = 1$, то $(-1)^{\langle z, s \rangle} = -1$ и следовательно

$$(-1)^{\langle x_1, z \rangle} (1 + (-1)^{\langle z, s \rangle}) = (-1)^{\langle x_1, z \rangle} (1 - 1) = 0$$

И вероятность в таком случае равняется 0

$$p_z = \left\| \frac{1}{2^n} \sum_{\ell \in A} ((-1)^{\langle x_1, z \rangle} (1 + (-1)^{\langle z, s \rangle}) |\ell\rangle) \right\|^2 = 0$$

Иначе $\langle z, s \rangle = 0$, и тогда $(-1)^{\langle z, s \rangle} = 1$:

$$(-1)^{\langle x_1, z \rangle} (1 + (-1)^{\langle z, s \rangle}) = 2(-1)^{\langle x_1, z \rangle}$$

Вычисление вероятности

$$\begin{aligned} p_z &= \left\| \frac{1}{2^n} \sum_{\ell \in A} (-1)^{\langle x_1, z \rangle} 2|\ell\rangle \right\|^2 = \\ &\left\| \frac{2}{2^n} \sum_{\ell \in A} (-1)^{\langle x_1, z \rangle} |\ell\rangle \right\|^2 = \left\| \frac{2}{2^n} \sum_{\ell \in A} (-1)^{\langle x_1, z \rangle} |\ell\rangle \right\|^2 = \left\| \frac{1}{2^{n-1}} \sum_{\ell \in A} (-1)^{\langle x_1, z \rangle} |\ell\rangle \right\|^2 \\ p_z &= \begin{cases} \frac{1}{2^{n-1}}, & \text{если } \langle z, s \rangle = 0 \\ 0, & \text{если } \langle z, s \rangle = 1 \end{cases} \end{aligned}$$

36.5 Ссылки

- Simon Algorithm
- Simon's problem
- Qiskit Simon algorithm

CHAPTER
THIRTYSEVEN

О БЛОКЕ “ЖЕЛЕЗО КВАНТОВЫХ КОМПЬЮТЕРОВ”

Этот блок включает в себя обзор различных подходов к физической реализации квантовых вычислений. Как и классические компьютеры, квантовые компьютеры вовсе не обязаны быть основаны на какой-то одном физическом принципе работы: информация может храниться в токах и напряжениях в электрической схеме, спинах атомов, механическом положении или движении микроскопических частиц. Квантовая механика проявляется для любых достаточно малых систем. В качестве критерия малости можно использовать постоянную Планка \hbar , которая имеет размерность фазового объема, или произведения координаты на импульс; это отражается и в принципе неопределенность Гейзенберга. Это значит, что если мы имеем систему, работающей на частоте ω , разность энергий уровней, в которых хранится информация, скорее всего будет порядка $\hbar\omega$. Из-за того, что $\hbar \sim 1.054 \times 10^{-34} \text{ J} \cdot \text{s}$ – очень маленькое число, два квантовых состояния очень слабо отличаются друг от друга.

Из-за того, что все платформы для квантовых вычислений очень разные, делать конкретные утверждения про все сразу практически невозможно. В этом блоке будет дана обзорная классификация разных платформ по разным признакам со ссылками на оригинальные работы, а затем более подробная лекция про сверхпроводниковые кубиты.

ОБЗОР И КЛАССИФИКАЦИЯ ФИЗИЧЕСКИХ ПЛАТФОРМ ДЛЯ КВАНТОВЫХ ВЫЧИСЛЕНИЙ

38.1 Радиочастотные и оптические кубиты

Первый признак, по которому можно классифицировать разные кубиты, это характер тех двух квантовых состояний, в которых хранится информация. Рассмотрим, например, фрагмент структуры уровней для иона Yb-171+ [Mon]. Этот ион часто используют для квантовых вычислений.

$^{171}\text{Yb}^+$ spin manipulation

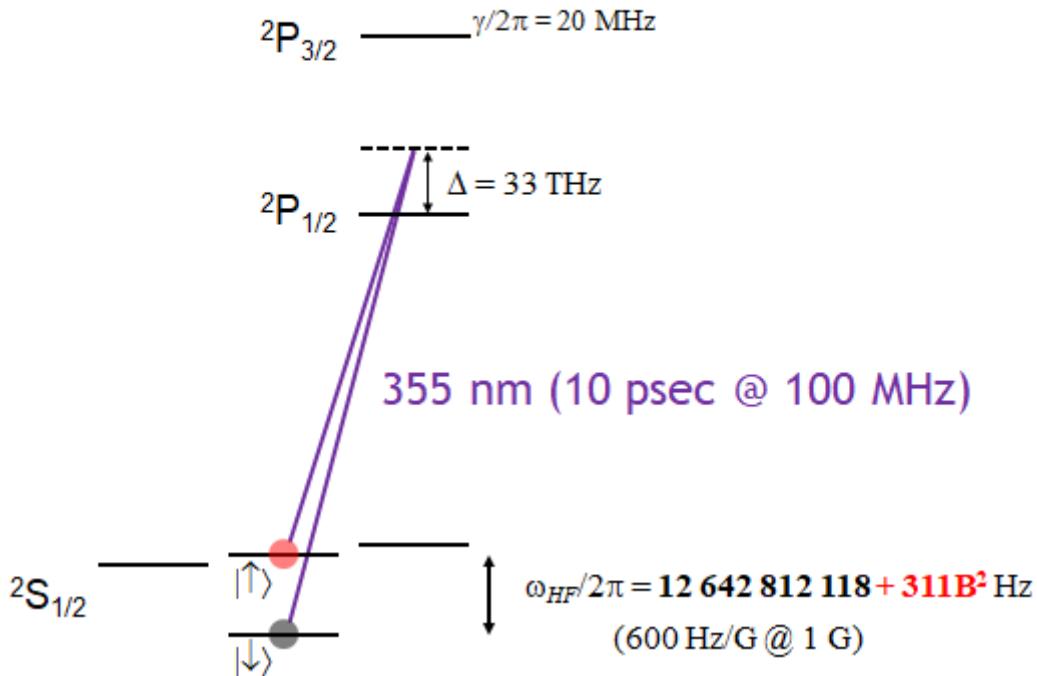


Fig. 38.1: Фрагмент структуры энергетических уровней иона Yb-171+. У типичного атома очень много степеней свободы: спин ядра, спин электронной подсистемы, радиус орбитали электрона и проекция момента орбитали электрона. Переходы между состояниями имеют очень разные частоты, времена когерентности и восприимчивости к частотам к внешним полям. Чтобы сделать из иона кубит, надо определиться, какой уровень будет “0”, а какой – “1”.

На сегодняшнем этапе все платформы для квантовых вычислений можно разделить на два класса: те, которые работают со сравнительно низкими $\omega/2\pi \sim 10^5 \dots 10^{10}$ Hz частотами (вплоть до микроволновых), и те, которые работают на высоких частотах $\omega/2\pi \sim 10^{13} \dots 10^{15}$ Hz. К первым относятся:

- сверхпроводниковые квантовые схемы;
- кубиты на тонком и сверхтонком расщеплении в нейтральных атомах;
- кубиты на сверхтонком расщеплении в ионах;
- кубиты спинах в квантовых точках;
- кубиты на экзотических квазичастицах, вроде Майорановских фермионов.

К второй группе относятся:

- оптические и телекоммуникационные фотонах;
- кубиты на оптических переходах в атомах и ионах.

Квантовые компьютеры первого типа надо сильно охлаждать, так как они самовозбуждаются под действием броуновского движения и теплового излучения тел, находящихся при комнатной температуре.

38.2 Природа степени свободы для кодирования информации

Можно классифицировать популярные платформы для квантовых вычислений и по другим признакам. Например, есть кубиты на естественных степенях свободы и кубиты на коллективных степенях свободы.

Естественные степени свободы – это:

- любые переходы в уединенных атомах и ионах;
- фотоны в открытом пространстве.

Кубиты на естественных степенях свободы автоматически получаются одинаковыми: вы берете одинаковые атомы, и у них обязательно будут одинаковые переходы. Впрочем, если поместить их в какую-нибудь ловушку, то из-за того, что ловушки немного влияют на частоты переходов в атомах, эту эквивалентность легко испортить.

Кубиты на искусственных степенях свободы получаются когда имеет место коллективная динамика большого числа частиц, например поляризация в материала оптоволокна, или коллективное движение электронов в сверхпроводнике. Примеры искусственных степеней свободы:

- сверхпроводниковые электрические цепи;
- фотоны в интегрально-оптической схеме;
- фотоны в оптоволокне;
- спины в квантовых точках;
- любые квазичастицы в твердотельных системах.

На самом деле граница между естественными степенями свободы и искусственными вовсе не такая очевидная, как может показаться на первый взгляд. Так, например, фотон, находящийся в открытом пространстве, можно поместить между двумя зеркалами, которые образуют резонатор. Но в таком случае в зеркалах, и в носителях заряда в зеркале будет храниться часть энергии фотона – получается так называемый “поляритон”, который уже не совсем естественная степень свободы. Абсолютно тождественными могут быть только те носители информации, которые ни с чем не связаны.

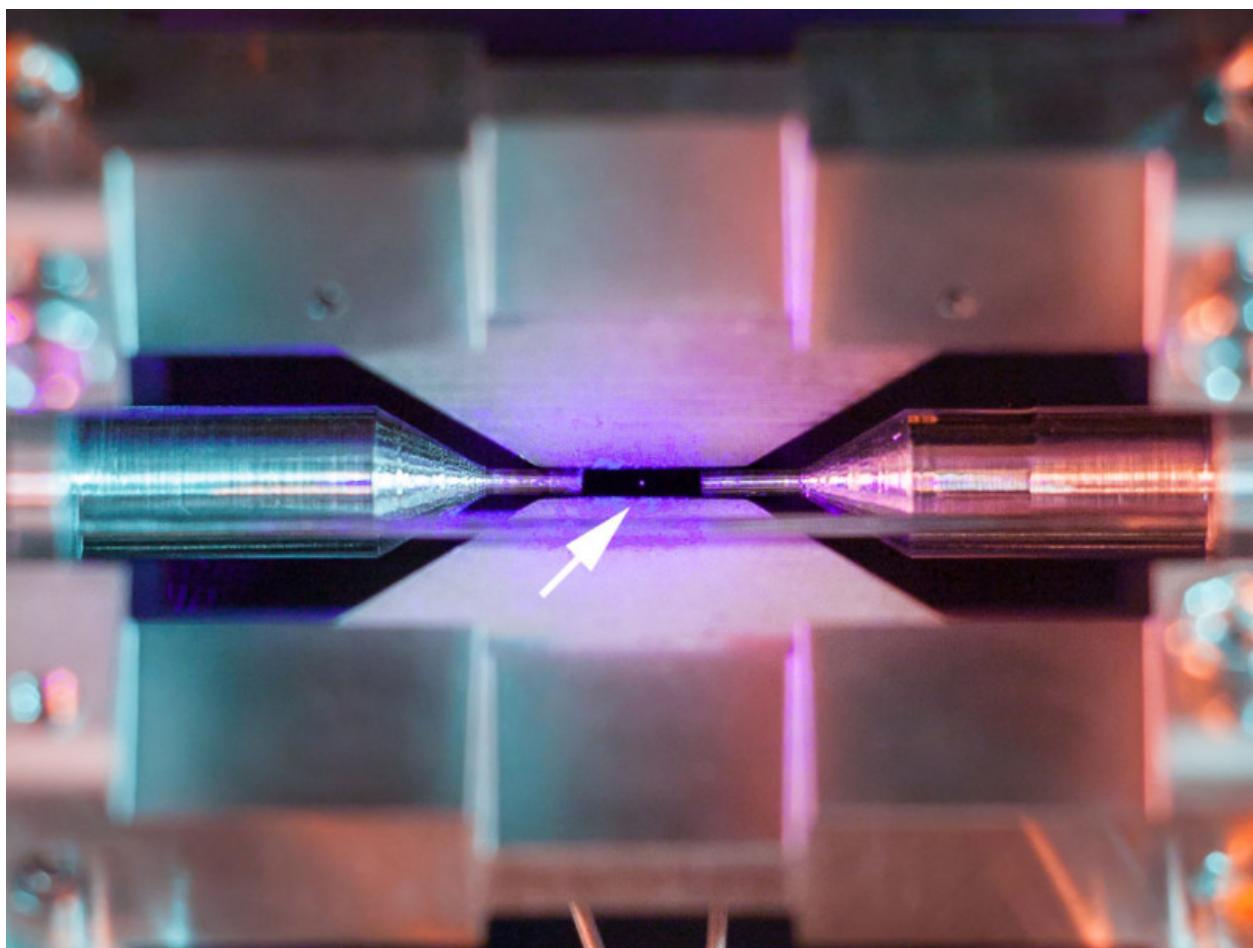


Fig. 38.2: Фотография уединенного атома в ловушке. Источник: [ion]

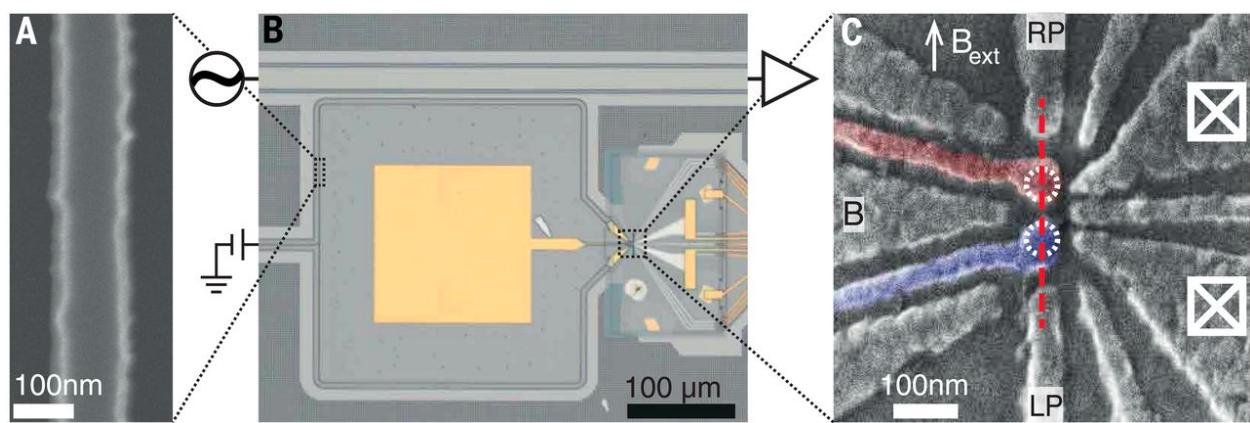


Fig. 38.3: Изображение кремниевой микросхемой с кубитом на квантовой точке. Источник: [SZK+18]

38.3 Способ кодирования информации

Только системы, где есть всего два уровня, а больше никакие не используются и не учитываются, строго говоря являются кубитами. Такими являются, например:

- потоковые сверхпроводниковые кубиты;
- логические кубиты, которые получаются выполнением кодов коррекции на большом числе физических кубитов любой природы.

Помимо “чистых” кубитов, у которых есть всего два дискретных состояния, существуют также “кудиты”, которые имеют d ($d > 2$) состояний. Таковыми являются, например:

- ионы со сложной структурой уровней и правилами отбора переходов между ними;
- сверхпроводниковые кубиты-трансмопы;
- атомы и ридберговские состояния в них.

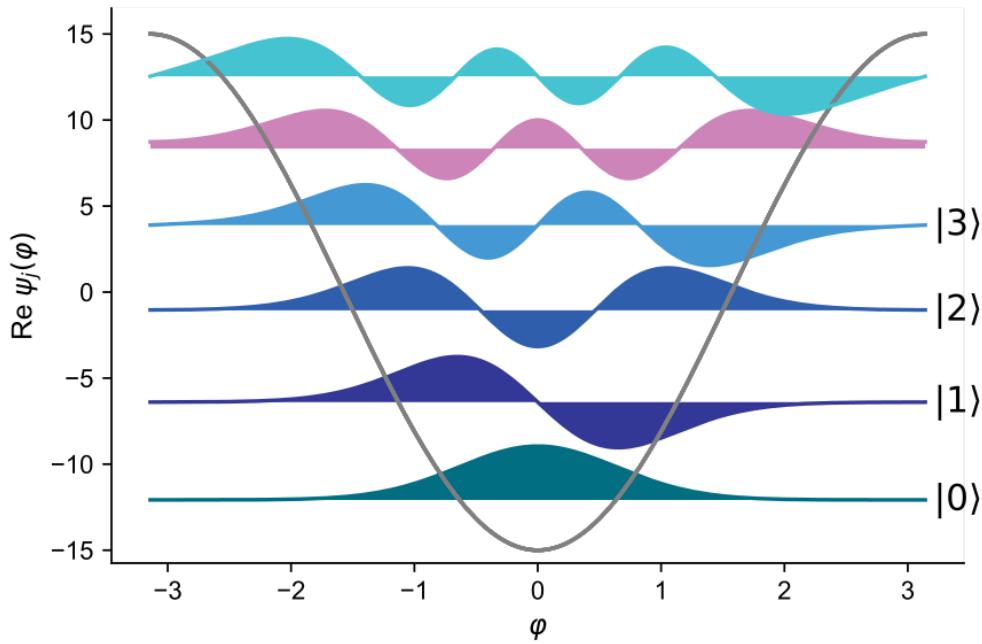


Fig. 38.4: Потенциал и уровни энергии в сверхпроводниковом кубите-трансмопе. Построено с помощью scqubits [GK21]. Как правило используются нижние два уровня, однако и остальные уровни можно использовать для алгоритмов

Бывают модели квантовых вычислений на непрерывных степенях свободы, например:

- оптические вычисления на непрерывных переменных;
- колебательное движение ионов в радиочастотных ловушках;
- состояния микроволнового поля в сверхпроводниковых резонаторах.

38.4 Способ выполнения операций

Модели вычислений в квантовых компьютерах можно классифицировать по признаку способа реализации операций в них. Самый простой и очевидный способ – воздействие на кубит (кубит, непрерывную переменную) при помощи короткого аналогового сигнала на резонансной частоте этой системы, причем амплитуда и фаза этого сигнала будут определять унитарную матрицу операции. Так работают

- сверхпроводниковые кубиты-трансмоны и потоковые кубиты;
- кубиты на ионах в радиочастотных ловушках;
- кубиты на нейтральных атомах в дипольных ловушках.

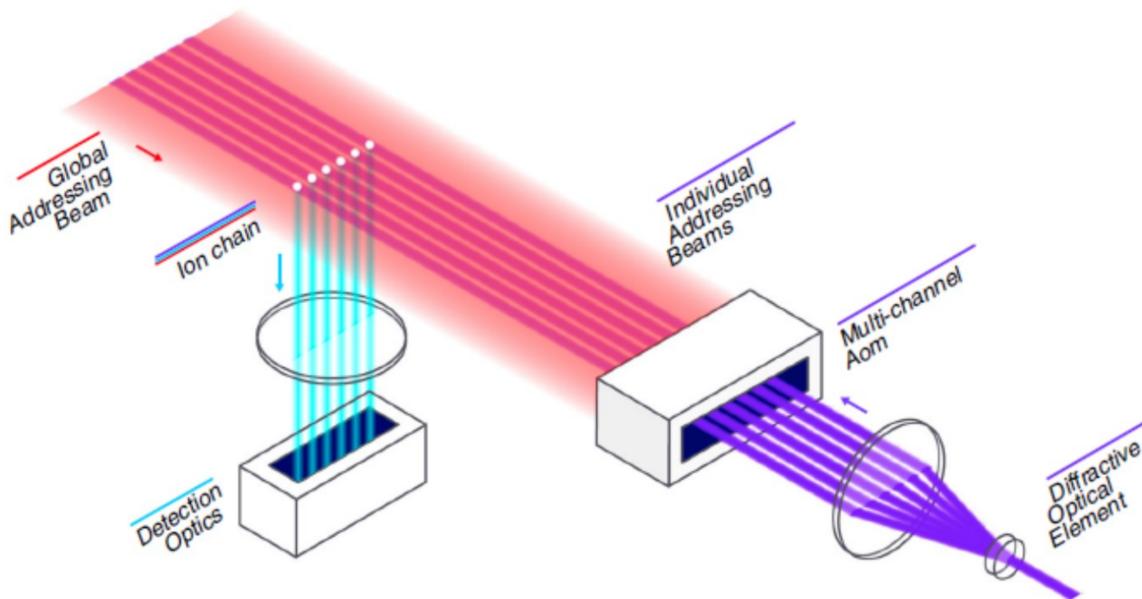


Fig. 38.5: Схема квантового компьютера на ионах. Источник: [WBD+19]. Кубиты расположены рядом друг с другом в вакуумной камере, а управляющие сигналы представляют собой лазерные лучи, направленные на каждый ион. Для однокубитных и двухкубитных операций используют разные лазеры, для считывания – еще лазер. Рассеянное на ионе излучение последнего лазера детектируется, чтобы определить состояние кубита. Похожим образом работают и кубиты на нейтральных атомах (там есть дополнительный оптический пинцет), и сверхпроводниковые кубиты (там вместо лазеров используются микроволновые кабели).

Бывают вычислительные модели, в которых вместо временной последовательности управляющих сигналов носитель информации движется по “конвейеру”, вдоль которого с ним выполняются операции. Так работают, например, вычисления на:

- фотонах в интегральной оптике;
- фотонах в открытом пространстве.

Некоторые модели вычислений вовсе не предполагают выполнения каких-то изолированных операций. Впрочем, для таких моделей вычислений и не выполняется большинство утверждений и теоретических предсказаний, связанных с квантовыми вычислениями. Таковыми моделями является, например, **квантовый аннилер D-Wave**. Важно отметить, что классические компьютеры работают совсем иначе, чем все вышеупомянутые квантовые компьютеры. В них операции осуществляются цифровыми сигналами, то есть от непосредственного значения уровня сигнала слабо зависит то, какая операция будет выполнена, главное, чтобы сигнал вовремя превысил некоторое пороговое значение. Цифровой характер классических компьютеров имеет очень важное значение для масштабирования алгоритмов, ведь именно он обеспечивает воспроизводимость

и возможность безошибочной работы квантового компьютера. К моделям квантовых вычислений, нечувствительным к аналоговым ошибкам управляющих сигналов, или с пониженной чувствительностью к ним, можно отнести:

- кубиты на майорановских фермионах;
- логические кубиты, которые получаются в результате работы алгоритмов квантовой коррекции ошибок на физических кубитах любой природы;
- сверхпроводниковые 0-пи кубиты, при условии, что гейты будут производиться специальным помехозащищенным образом, а не через промежуточные уровни.

38.5 Наборы инструкций и универсальность

Разные модели вычислений дают разные наборы инструкций для реализации квантовых алгоритмов. В зависимости от вероятности успешной реализации инструкций (квантовые компьютеры делают ошибки!) эти модели вычислений могут быть ближе, либо дальше от создания универсального масштабируемого квантового компьютера. По этому признаку:

- сверхпроводниковые кубиты;
- нейтральные атомы;
- ионы в радиочастотных ловушках

являются самыми очевидными. Набор инструкций в них содержит однокубитные и двухкубитные операции, инициализацию кубита в определенном состоянии и считывание. Чем выше точность операций – тем лучше, и все операции на сегодняшний день продемонстрированы с вероятностью положительного исхода выше 99% – этого все еще мало для многих применений, но это уже что-то.

Для вычислений на линейнооптических схемах используются источники единичных эквивалентных фотонов, либо источники запутанных фотонов, либо источники сжатого света. Вероятность успеха операции по созданию такого излучения (либо степень сжатия света) на сегодняшний день сильно далека от того, что нужно для универсальности квантовых вычислений; здесь идет речь о сжатии порядка 10 дБ (грубо переводя на язык “успешного выполнения операции” 0,9), и вероятности испускания фотона 0,9. Однако отсутствие универсальности не означает, что линейнооптические схемы должны быть для всех задач вычислительно слабее, чем другие платформы: напротив, самое сильное квантовое превосходство на сегодняшний день достигнуто именно на таких устройствах.

Причина, по которой линейнооптические системы выделяются среди других заключается в том, что для полноценного компьютера (что классического, что квантового) нужны нелинейные элементы. В классическом компьютере это транзисторы. На одних линейных индуктивностях и емкостях можно, в зависимости от способа кодирования информации, выполнять какой-то набор операций, но универсальным он не будет, потому что функция отклика линейной цепи является линейным. Алгоритмы, которые не сводятся к умножению матрицы на вектор в некотором представлении данных, на линейной системе выполнить не получится. Чтобы обойти это ограничение, линейно-оптические системы для квантовых вычислений принимают на часть своих входов, в дополнение к входным данным, сложные неклассические сигналы, а на выходе производится считывание числа фотонов, которое является квадратичным по отношению к амплитудам сигналов. Дополнение линейной оптики наборами нелинейных операций может сделать такие модели вычислений универсальными.

Наконец, кубиты с дискретными операциями, такие как логические кубиты и кубиты на майорановских фермионах, сами по себе не дают полный набор квантовых логических операций – в дополнение к ним нужны обычные кубиты. Впрочем, задача построения кубита на майорановских фермионах или на коррекции ошибок сама по себе настолько сложная, что необходимость связать с ним обычные дополнительные кубиты не кажется такой сложной.

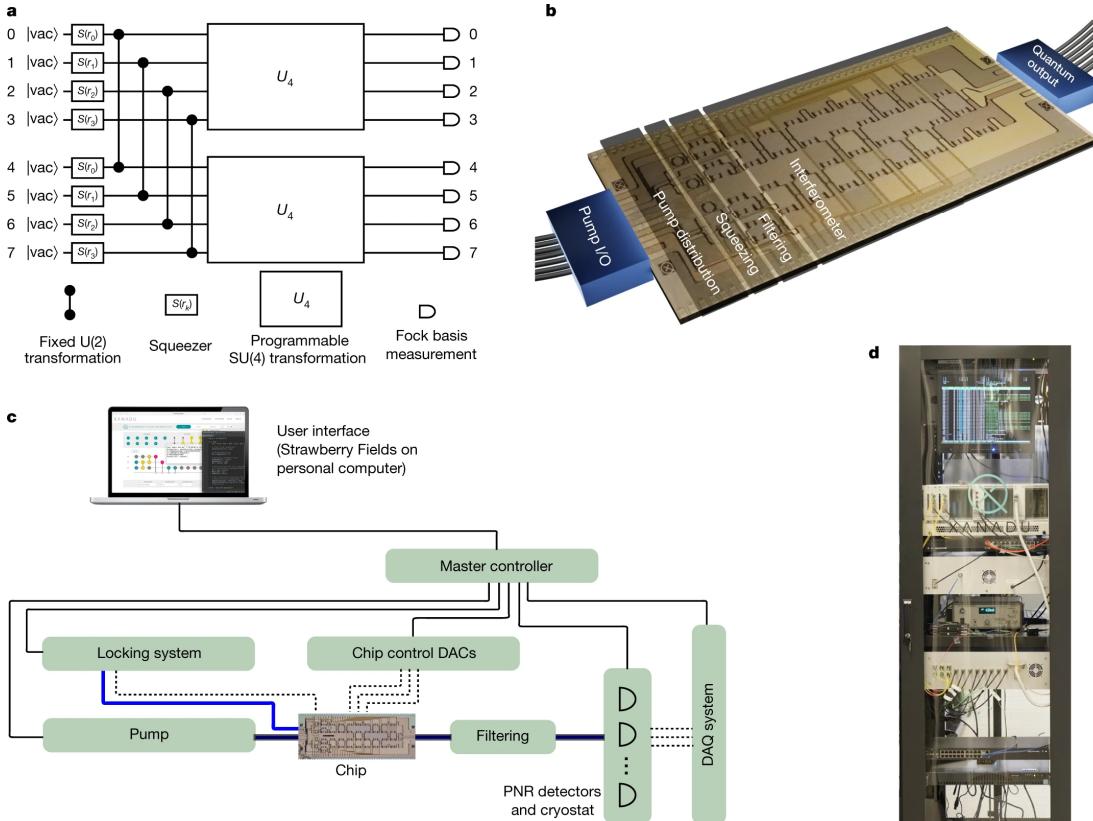


Fig. 38.6: Принципиальная схема работы оптического квантового чипа Xanadu. Взято из работы [ABBradler+21]. а – квантовый алгоритм, выполняемый схемой: 1) сжатие света до 8 модах и создание запутанности (формирование двухмодового сжатого состояния света, эквивалент однокубитных гейтов), 2) программируемые двухкубитные гейты в каждом четырехмодовом подпространстве, 3) считывание всех 8 мод в фоковском базисе; б – изображение чипа, на котором видны основные модули; в – схема управляющих сигналов и приборов; г – фотография всей системы (кроме управления детектором фотонов) в серверной стойке.

38.6 Успехи в практической реализации

Платформы для квантовых вычислений можно отсортировать по максимальному размеру гильбертова пространства и точности операций (не только унитарных гейтов, но и считыванию и инициализации), которое удавалось реализовать экспериментально. В этом рейтинге только три верхние платформы находятся относительно близко к режиму квантового превосходства в подтвержденных экспериментальных работах. Итак, рейтинг:

- линейная оптика (фотоны) в открытом пространстве;
- сверхпроводниковые кубиты-трансмоны;
- ионы в линейных радиочастотных ловушках;
- интегральнооптические фотони;
- нейтральные атомы в дипольных ловушках;
- спины в квантовых точках.

Другие платформы существуют на сегодняшний день в лучшем случае в виде двухкубитной системы, чаще – одного кубита, а лучше вообще в виде теоретической концепции – которая может оказаться вовсе нереализуемой на практике.

СВЕРХПРОВОДНИКОВЫЕ КУБИТЫ

Сверхпроводниковые кубиты относятся к наиболее успешным типам кубитов для квантовых вычислений и симуляции на сегодняшний день. Google впервые продемонстрировала квантовое превосходство на квантовом процессоре из 53 сверхпроводниковых кубитов-трансмонов [AAB+19]. Кроме Google, сверхпроводниковые кубиты используют IBM (в том числе для своего открытого облачного проекта IBM Quantum Experience), Rigetti Computing и другие крупные фирмы и лаборатории по всему миру.

Квантовый процессор на сверхпроводниковых кубитах – это электрическая цепь. Этую электрическую цепь можно изобразить в схемы с квазисосредоточенными элементами.

39.1 Квантование электрических цепей

Электрические цепи, как и любые другие физические объекты, могут быть описаны с помощью законов квантовой механики [VD17]. Для этого следует начать с гамильтонового описания, например с помощью функции Лагранжа. Энергия всей физической системы (электрической цепи) должна быть записана в виде суммы энергий, накопленных ее составными частями (сосредоточенных элементов), выраженной через динамические переменные системы и их производные. Самые простые электрические элементы, такие как конденсаторы и индукторы, запасают энергию E_C и E_L в электрическом и магнитном поле, а выражение для них можно записать либо через заряд на обкладках Q и поток через кольцо Φ , либо через напряжение V и ток I , соответственно:

$$E_C = \frac{Q^2}{2C} = \frac{CV^2}{2},$$

$$E_L = \frac{\Phi^2}{2L} = \frac{LI^2}{2}.$$

Для третьего самого простого линейного элемента электрической цепи, резистора, так сделать не получится. Резистор не накапливает в себе энергию, он ее диссирирует. Диссипативные системы не могут быть описаны исключительно при помощи функции Лагранжа. Это не значит, что резистор совершенно невозможно адекватно описать в гамильтоновом формализме: он может быть представлен как бесконечный резервуар энергии, так как с точки зрения остальной цепи не слишком важно, осталась ли энергия в резисторе или она была рассеяна, если она никогда больше не вернется в остальные элементы цепи. Резисторы не слишком полезны в квантовой обработке информации, так как любые потери энергии приводят к неунитарной эволюции, потере когерентности и в конечном счете – ошибкам вентиляй.

В электротехнике, как правило, используются переменные V и I : напряжение и ток можно померить, правильно подключив измерительные приборы к проводникам цепи, в то время как измерения Q и Φ затруднены, так как заряды и потоки локализованы внутри элементов. Но V и I не подходят в качестве переменных, от которых зависит функция Лагранжа, так как ни одна из них не является производной другой. Зато из закона сохранения заряда следует, что $\dot{Q} = -I$, а закон индукции Фарадея говорит, что $\dot{\Phi} = V$, а значит и Φ , и Q могут быть использованы в качестве динамических переменных системы.

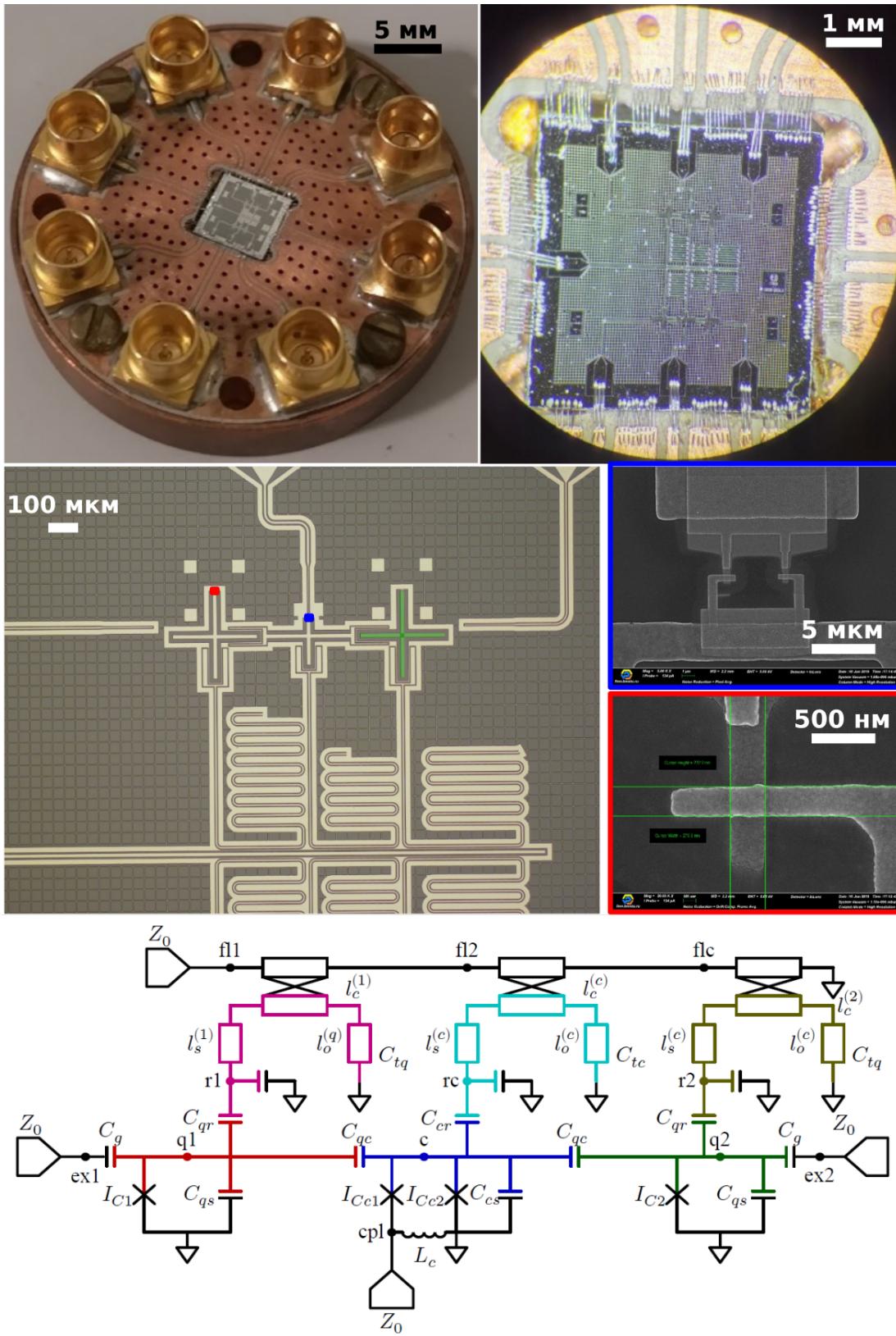


Fig. 39.1: Фотографии, изображения со сканирующего электронного микроскопа и эквивалентная электрическая схема микросхемы сверхпроводникового квантового процессора с двумя кубитами-трансмонами. Синим отмечен электрод одного из трансмонов. Микросхема была изготовлена в НОЦ ФМН в МГТУ им. Баумана, технология разработана в НИТУ МИСиС.

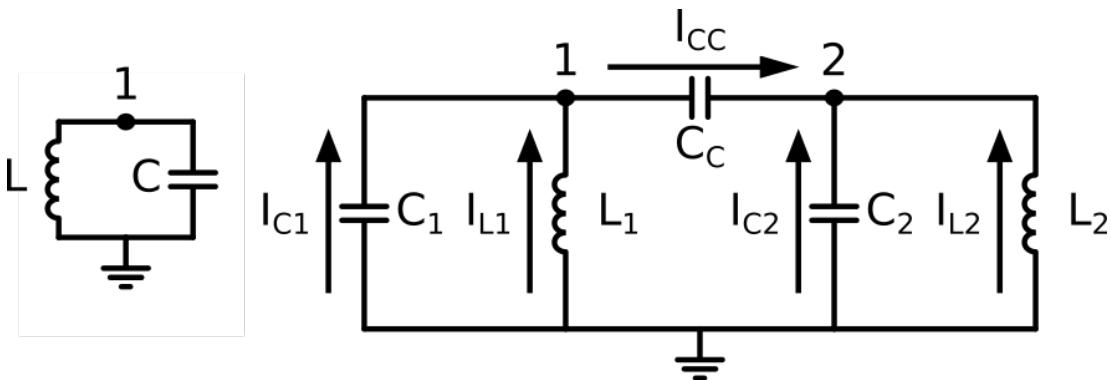


Fig. 39.2: Электрические схемы одного LC-контура и двух емкостно связанных LC-контуров.

В случае одной степени свободы определения для величин Q , Φ , V и I однозначны. В более сложных случаях в классической электротехнике принято использовать метод контурных токов, либо узловых напряжений. Рассмотрим метод узловых напряжений. Для лагранжевого описания введем узловой поток $\Phi_i(t)$ как первообразную напряжения на узле с номером i $V_i(t)$:

$$\Phi_i(t) = \int_0^t V_i(t) dt.$$

Функция Лагранжа LC-контура имеет вид

$$L(\Phi_1, \dot{\Phi}_1) = \frac{C\dot{\Phi}_1^2}{2} - \frac{\Phi_1^2}{2L}.$$

Канонически сопряженный импульс Q_i к обобщенной переменной Φ_i можно найти по формуле

$$Q_i = \frac{\partial L}{\partial \dot{\Phi}_i} = C\dot{\Phi}_i.$$

Физический смысл этого импульса – сумма зарядов на конденсаторах, примыкающих к узлу i , или узловой заряд.

Функция Гамильтона $H(\Phi_i, Q_i)$ представляет собой преобразование Лежандра от функции Лагранжа по парам $(\dot{\Phi}_i, Q_i)$, то есть

$$H(\Phi_i, Q_i) = \dot{\Phi}_i \dot{Q}_i - L = \frac{Q_i^2}{2C} + \frac{\Phi_i^2}{2L}.$$

Оператор Гамильтона может быть получен из функции Гамильтона заменой переменных Φ_1, Q_1 на соответствующие операторы:

$$H = \frac{\hat{Q}_1^2}{2C} + \frac{\hat{\Phi}_1^2}{2L}.$$

Здесь $[\hat{Q}_1, \hat{\Phi}_1] = -i\hbar$ в силу того, что Φ_1 и Q_1 – каноническая переменная и сопряженный к ней импульс. Волновая функция системы может быть записана в потоковом $\psi_\Phi(\Phi_1)$ или зарядовом $\psi_Q(Q_1)$ представлении. Переход от одного представления к другому, как в случае обыкновенных координат и импульсов, может быть произведен с помощью преобразования Фурье

$$\psi_\Phi(\Phi_1) = \frac{1}{\sqrt{2\pi}} \int e^{i\Phi_1 Q_1} \psi_Q(Q_1, Q_2) dQ_1.$$

Как и в случае механических осцилляторов, для LC-осцилляторов очень полезен формализм операторов вторичного квантования \hat{a} и \hat{a}^\dagger , которые можно определить как

$$\hat{Q}_1 = \sqrt{\frac{\hbar}{2Z}} (\hat{a}_1 + \hat{a}_1^\dagger),$$

$$\hat{\Phi}_1 = i\sqrt{\frac{\hbar Z}{2}}(\hat{a}_1 - \hat{a}_1^\dagger),$$

где $Z = \sqrt{L_1/C_1}$ – импеданс осциллятора. В этом формализме гамильтониан системы принимает вид

$$\hat{H} = \hbar\omega_1(\hat{a}_1^\dagger\hat{a}_1 + \frac{1}{2}),$$

где $\omega_1 = (L_1 C_1)^{-1/2}$ – частота осциллятора.

39.2 Джозефсоновский контакт

Линейные электрические элементы, такие как емкости и индуктивности, очень полезны при создании электронных приборов. Но из линейных элементов можно создать только линейную электрическую цепь. Это означает, что любые выходные сигналы $V_i^{\text{out}}(\omega)$ выражаются через входные сигналы $V_j^{\text{in}}(\omega)$ с помощью линейного преобразования

$$V_i^{\text{out}}(\omega) = \sum_j S_{ij}(\omega) V_j^{\text{in}}(\omega).$$

Это выражение остается справедливым и для квантового случая, где величина $V_i^{\text{out}}(\omega)$ является оператором. $S_{ij}(\omega)$ – это передаточная функция, или S-параметр системы. Для линейной системы S-параметр зависит лишь от частоты сигнала, но не зависит от состояния системы. По существу линейные системы не обладают состоянием, т.е. не могут играть роль памяти. В частности, такие системы не могут эффективно симулировать машину Тьюринга и тем более квантовую машину Тьюринга. Они могут реализовывать некоторое линейное преобразование над входными данными, однако это линейное преобразование жестко задано топологией и номиналами емкостей и индуктивностей цепи.

Для того, чтобы создать архитектуру компьютера, в которой есть память или управление, нужны нелинейные элементы. В классической электронике в качестве нелинейного элемента чаще всего используют транзистор. Наиболее популярный в интегральных микросхемах – это полевой транзистор. Полевой транзистор имеет три контакта, причем напряжение между затвором и стоком управляет сопротивлением между стоком и истоком. Для квантовой обработки информации управление сопротивлением – не самый practicalный вариант, в связи с тем, что протекание тока через конечные сопротивления приводит к ошибкам квантовых вентилей.

В качестве альтернативы полупроводниковым нелинейным элементам вроде транзистора и диода можно использовать сверхпроводниковые нелинейные элементы. Одним из таких элементов является джозефсоновский контакт сверхпроводник-изолятор-сверхпроводник (Superconductor-insulator-superconductor, SIS).

Для гамильтонового описания джозефсоновского контакта как элемента электрической цепи используется джозефсонская энергия, которая определяется соотношением

$$E_J = \frac{I_c \Phi_0}{2\pi} (1 - \cos \phi),$$

где I_c – критический ток контакта, $\Phi_0 = h/2e$ – квант магнитного потока в сверхпроводнике, $\phi = 2\pi\Phi/\Phi_0$ – обезразмеренная на квант потока разность узловых потоков на берегах контакта. Эта величина одновременно является фазой конденсата куперовских пар в сверхпроводнике. Для понимания принципов работы сверхпроводниковых кубитов совершенно необходимо понимать физику самой сверхпроводимости, что такое куперовская пара, что подразумевается под словом конденсат и что значит его фаза. С точки зрения схемотехники кубитов, все знания о физике сверхпроводников могут быть сведены к определению фазы и соотношению для энергии контакта в зависимости от фазы. Эти соотношения однозначно определяют поведение потоков и зарядов в цепи. Мы будем пользоваться фазой вместо узлового потока исключительно из-за того, что эта величина является безразмерной, и это позволит упростить все соотношения. Также заряд будет измеряться в единицах заряда куперовской пары, то есть $2e$.

Дифференцируя соотношение для джозефсоновской энергии по потоку, можно получить выражение для индуктивности джозефсоновского контакта

$$L_J = \frac{1}{\frac{\partial^2 E}{\partial \Phi^2}} = \frac{2\pi\Phi_0^2}{I_c\Phi_0(2\pi)^2 \cos \phi} = \frac{\Phi_0}{I_c 2\pi \cos \phi}.$$

39.3 Трансмон

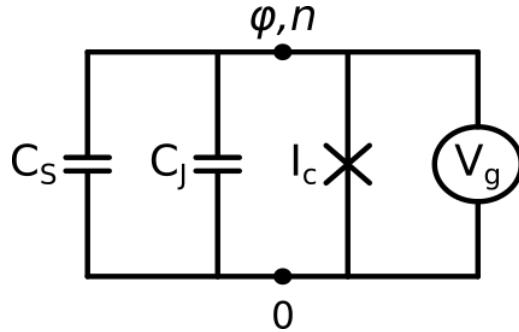


Fig. 39.3: Электрическая схема трансмона. I_c – критический ток контакта, C_S – шунтирующая емкость, C_J – емкость контакта

Самый простой, и в то же время один самых успешных видов сверхпроводниковых кубитов – это трансмон[KYG+07]. Трансмон представляет собой ангармонический осциллятор на основе наноразмерного джозефсоновского контакта, шунтированного емкостью. Такая цепь хорошо известна в сверхпроводниковой электронике, а ее резонансная частота называется плазменной частотой:

$$\omega_p = \frac{1}{\sqrt{L_J(C_J + C_S)}}.$$

В квантовом случае, свойства этого осциллятора могут зависеть от постоянного напряжения V_g . Наступление квантового режима определяется произведением критического тока и емкости; фактически, в зависимости от этого произведения, такая схема может быть классическим осциллятором, трансмоном или зарядовым кубитом.

Гамильтониан трансмона (а также зарядового кубита) имеет вид

$$\hat{H} = \frac{(2e)^2(\hat{n} - n_g)^2}{2(C_J + C_S)} + \frac{I_c\Phi_0}{2\pi}(1 - \cos \hat{\phi}),$$

где $\hat{n} = \hat{Q}/2e$, $\hat{\phi} = \frac{2e}{\hbar}\hat{\Phi}$ – безразмерные операторы числа и фазы куперовских пар на электроде, а n_g – средний наведенный на острове внешними источниками напряжения заряд. Гамильтониан можно упростить, заменив произведения размерных константы на параметры

$$E_C = \frac{e^2}{2(C_J + C_S)}$$

$$E_J = \frac{I_c\Phi_0}{2\pi},$$

которые называют емкостной и джозефсоновской энергией трансмона, соответственно. В этих обозначениях гамильтониан принимает вид

$$\hat{H} = 4E_C(\hat{n} - n_g)^2 + E_J(1 - \cos \hat{\phi}).$$

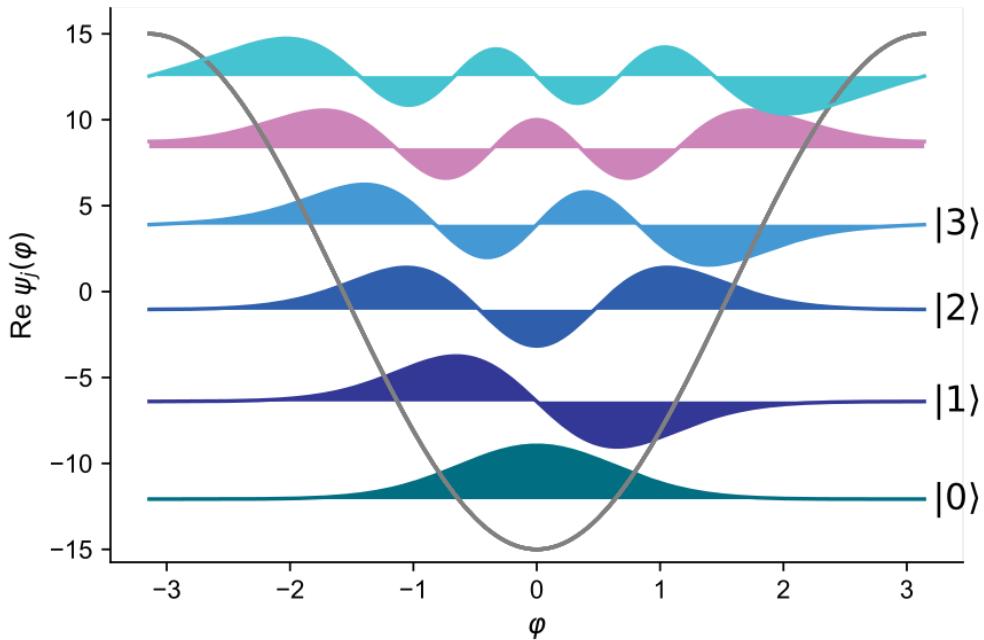
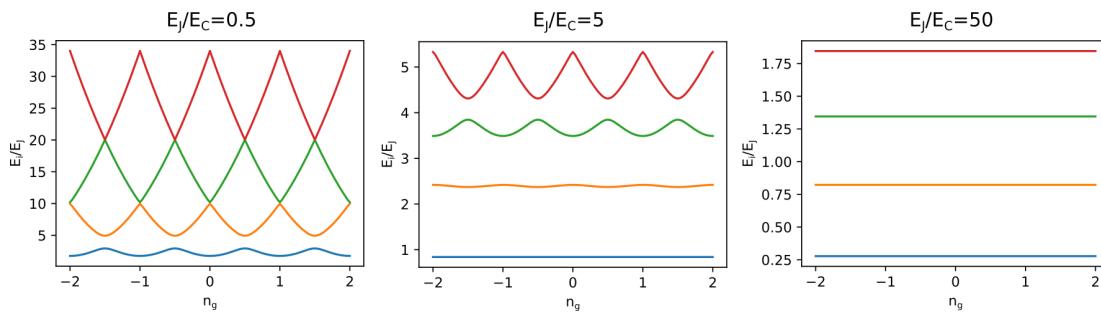


Fig. 39.4: Уровни энергии в зарядовом кубите и в трансмоне

Уровни энергии стационарных состояний в такой системе будут зависеть от приложенного напряжения (вернее, от наведенного внешним напряжением V_g зарядом n_g). Однако эта зависимость ослабевает по мере того, как увеличивается соотношение E_J/E_C .

Трансмоном называется режим этой электрической цепи, когда $E_J/E_C \sim 50$; при таком соотношении между константами зависимость от заряда практически отсутствует. Это очень хорошо, потому что наведенный заряд очень сложно контролировать на практике. Космические частицы и события радиоактивного распада в подложке приводят к запиранию зарядов в случайных местах в подложке, из-за чего зарядово-чувствительные кубиты могут менять частоту перехода между уровнями. Уровни энергии в трансмоне не чувствительны к таким воздействиям, что значительно улучшает их времена когерентности по сравнению с зарядовыми кубитами. Обратной стороной такой схемы является невысокий ангармонизм. Ангармонизмом δ называется разность частот между переходами $|0\rangle \leftrightarrow |1\rangle$ и $|1\rangle \leftrightarrow |2\rangle$. У трансмона $\delta = -E_C$. По существу, трансмон является слабо ангармоничным осциллятором. Однако расстояния между частотами переходов в трансмоне достаточно большие, чтобы можно было попасть в один переход и при этом не попасть в соседний.

Fig. 39.5: Уровни энергии и волновые функции стационарных состояний трансмона с $E_J/E_C = 50$

39.4 Считывание

Считывание состояния кубита в сверхпроводящих цепях, как правило, осуществляется методом дисперсионного считывания [WSB+05]. В основе дисперсионного считывания состояния сверхпроводниковых кубитов стоит архитектура квантовой электродинамики в электрической цепи, в которой сверхпроводниковый кубит связан с микроволновым резонатором. Концепция квантовой электродинамики в электрической цепи возникла по аналогии с квантовой электродинамикой в полости, где уединенный атом помещается в оптическую полость, которая выступает в роли резонатора. Взаимодействие такого атома с излучением сильно отличается от того, что происходит с таким атомом, помещенным в вакуум: из-за отражений от стенок полости, излучение не может эффективно покинуть атом. Это увеличивает время жизни атома, если резонансная частота полости находится вдали от спектральной линии кубита. Напротив, если атом находится в резонансе с полостью, то это стимулирует излучение.

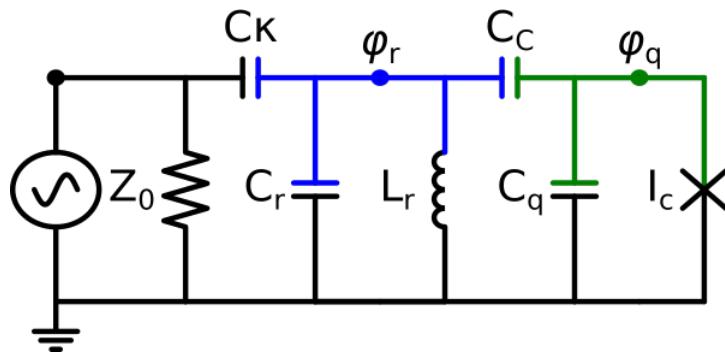


Fig. 39.6: Эквивалентная электрическая схема трансмона, связанного с резонатором. Синий — электрод резонатора, зеленый — электрод трансмона

Для многокубитных схем, как правило, в качестве микроволнового резонатора используется участок копланарной передающей линии на микросхеме. Микроволновый резонатор, в свою очередь, подключается к микроволновой линии.

Пропускание микроволновых сигналов через линию приводит к изменению квантового состояния электромагнитного излучения в резонаторе; это излучение взаимодействует с кубитом. Преимуществом данной схемы является то, что кубит может быть очень сильно связан с резонатором, но при этом сохраняет когерентность. Если пренебречь влиянием потерь в управляемую линию, то система описывается гамильтонианом вида

$$\hat{H}/\hbar = \left(\omega_q + \frac{1}{2} \delta_q (\hat{a}^\dagger \hat{a} - 1) \right) \hat{a}^\dagger \hat{a} + \left(\omega_r + \frac{1}{2} \delta_r (\hat{b}^\dagger \hat{b} - 1) \right) \hat{b}^\dagger \hat{b} + \chi \left(\hat{a}^\dagger \hat{a} + \frac{1}{2} \right) \left(\hat{b}^\dagger \hat{b} + \frac{1}{2} \right),$$

где ω_q, ω_r — частоты кубита и резонатора, соответственно; δ_q, δ_r — ангармонизмы мод трансмона и резонатора; χ — дисперсионный сдвиг, или кросс-керровская нелинейность между трансмоном и резонатором, \hat{a}, \hat{a}^\dagger — одночастичные операторы уничтожения и рождения, действующие на моду трансмона, \hat{b}, \hat{b}^\dagger — одночастичные операторы рождения и уничтожения, действующие на моду резонатора, $\Omega(t)$ — амплитуда возбуждения со стороны микроволновой линии.

Если емкость C_C между трансмоном и резонатором маленькая, а их частоты сильно отличаются, то взаимодействие между ними будет нерезонансным, и тогда ангармонизмом резонатора можно пренебречь. А вот наличие дисперсионного сдвига между модой кубита и резонатора приводит к тому, что частота эффективная резонатора, то есть сумма всех чисел перед множителем $\hat{b}^\dagger \hat{b}$, зависит от населенности трансмона. В таком случае, измеряя отклик от резонатора на микроволновый импульс вблизи его резонансной частоты, мы можем получить информацию о состоянии кубита.

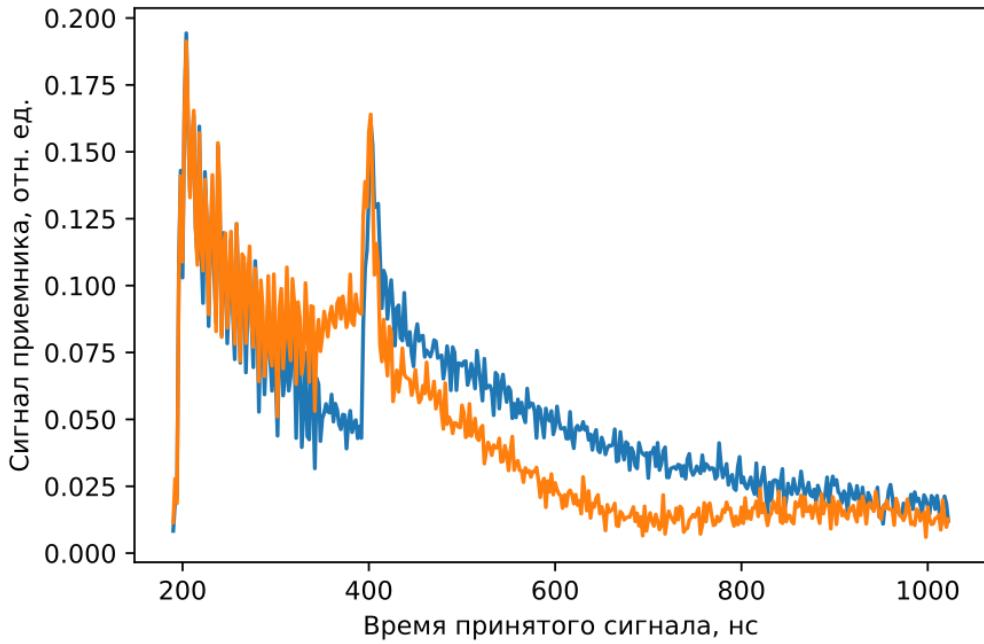


Fig. 39.7: Микроволновый прямоугольный импульс, отраженный от резонатора, при различных состояниях трансмона. Усреднено по 10000 повторениям.

39.5 Микроволновые однокубитные операции

Логические операции над кубитами выполняют микроволновыми сигналами на резонансной частоте кубита. Поведение трансмона по отношению к внешнему излучению можно представить в виде гамильтониана

$$\hat{V} = V_{\text{exl}}(t) \frac{C_g}{C_g + C_{qs}} \hat{Q}$$

для схемы с возбуждением напряжением $V_{\text{exl}}(t)$, подаваемым через маленький конденсатор. Пусть

$$V_{\text{exl}}(t) = V_I(t) \cos(\omega_q t + \phi(t)) + V_Q(t) \sin(\omega_q t + \phi(t)),$$

где $V_I(t), V_Q(t)$ – медленно изменяющиеся огибающие напряжения, а $\phi(t)$ – медленно изменяющаяся фаза. “Медленно изменяющаяся” здесь означает, что за переходы между уровнями будет отвечать только быстро осциллирующий множитель. Обозначим

$$\Omega(t) = I(t) - iQ(t) = \frac{C_g}{C_g + C_{qs}} \sqrt{\frac{\hbar}{2Z}} (V_I - iV_Q).$$

Полный гамильтониан трансмона с возбуждающим излучением будет иметь вид

$$\hat{H}_{\text{dr}}/\hbar = \hat{H}/\hbar + \hat{V}/\hbar = \left(\omega_q + \frac{1}{2} \delta_q (\hat{a}^\dagger \hat{a} - 1) \right) \hat{a}^\dagger \hat{a} + \frac{1}{2} \left(\Omega(t) e^{i(\omega_q t + \phi(t))} + \Omega^*(t) e^{-i(\omega_q t + \phi(t))} \right) (\hat{a} + \hat{a}^\dagger).$$

Если трансмон использовать как кубит (то есть не возбуждать никаких уровней, кроме основного и первого возбужденного), можно перейти от операторов повышения и понижения к операторам Паули с помощью

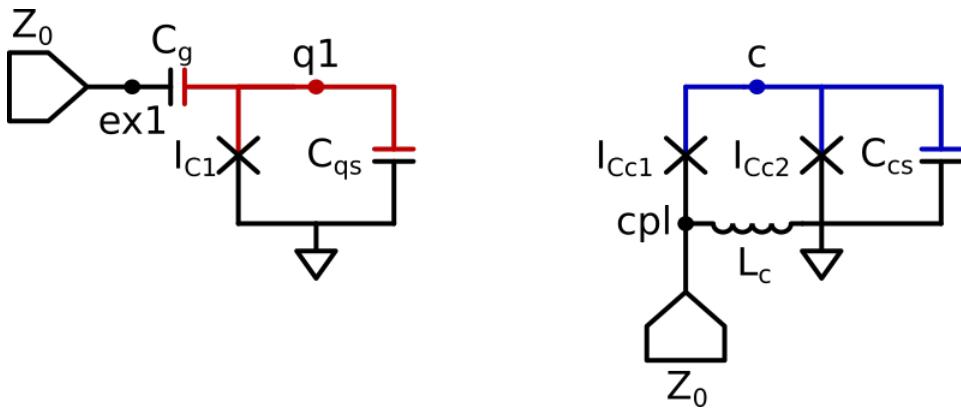


Fig. 39.8: Трансмон с линиями для микроволновых однокубитных операций. Слева – емкостно связанные линии (через конденсатор), справа – индуктивно связанные линии (через индуктивность).

соотношений

$$\begin{aligned}\hat{a}^\dagger \hat{a} &= -\frac{1}{2} \hat{\sigma}_z \\ \hat{a} + \hat{a}^\dagger &= \hat{\sigma}_x \\ i(\hat{a} - \hat{a}^\dagger) &= \hat{\sigma}_y.\end{aligned}$$

Тогда мы получим гамильтониан вида

$$\hat{H}_{\text{dr}}/\hbar = \frac{1}{2} (\Omega(t)e^{i(\omega_q t + \phi(t))} + \Omega^*(t)e^{-i(\omega_q t + \phi(t))}) \sigma_x.$$

Такое возбуждение отвечает модели Раби для двухуровневой системы. Модель Раби хорошо известна в квантовой оптике, так как она описывает когерентное взаимодействие одного атома с лазерным излучением. Здесь мы остановимся поподробнее, так как модель Раби имеет центральное значение для однокубитных операций на очень широком классе различных кубитов.

Пусть $|\psi(t)\rangle = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix}$ – решение уравнения Шредингера. Для того, чтобы решить уравнение Шредингера, которое является системой двух обыкновенных линейных дифференциальных уравнений относительно $c_0(t)$ и $c_1(t)$, нужно сделать замену

$$\begin{aligned}c_0(t) &= d_0(t)e^{-i\omega_q t/2}, \\ c_1(t) &= d_1(t)e^{+i\omega_q t/2}.\end{aligned}$$

Эта замена называется переходом во вращающуюся систему отсчета. В данном случае, система отсчета вращается вместе с быстро осциллирующей частью микроволнового сигнала. Эта замена может быть записана в виде произведения матрицы на вектор:

$$\begin{pmatrix} c_0(t) \\ c_1(t) \end{pmatrix} = \begin{pmatrix} e^{i(\omega_q t + \phi(t))/2} & 0 \\ 0 & e^{-i(\omega_q t + \phi(t))/2} \end{pmatrix} \begin{pmatrix} d_0(t) \\ d_1(t) \end{pmatrix}.$$

Матрица этого линейного преобразования $\hat{U} = e^{i(\omega_q t + \phi(t))/2\hat{\sigma}_z}$ является унитарной – ее собственные числа по модулю равны единице. Если подставить эту замену в уравнение Шредингера

$$i\hbar \partial |\psi\rangle / \partial t = \hat{H}_{\text{dr}} |\psi\rangle,$$

то получим систему

$$\begin{aligned} i\dot{d}_0 e^{i(\omega_q t + \phi(t))/2} + \frac{\omega_q + \dot{\phi}}{2} d_0 e^{i(\omega_q t + \phi(t))/2} = \\ -\frac{\omega_q}{2} d_0 e^{i(\omega_q t + \phi(t))/2} + d_1 e^{i(\omega_q t + \phi(t))/2} (\Omega(t) e^{i(\omega_q t + \phi(t))} + \Omega^*(t) e^{-i(\omega_q t + \phi(t))}), \\ i\dot{d}_1 e^{-i(\omega_q t + \phi(t))/2} - \frac{\omega_q + \dot{\phi}}{2} d_1 e^{-i(\omega_q t + \phi(t))/2} = \\ \frac{\omega_q}{2} d_1 e^{-i(\omega_q t + \phi(t))/2} + d_0 e^{i(\omega_q t + \phi(t))/2} (\Omega(t) e^{i(\omega_q t + \phi(t))} + \Omega^*(t) e^{-i(\omega_q t + \phi(t))}). \end{aligned}$$

Некитрыми алгебраическими преобразованиями можно упростить гамильтониан до вида

$$\begin{aligned} i\dot{d}_0 + \frac{\dot{\phi}(t)}{2} d_0 = \\ (\Omega(t) e^{i2(\omega_d t + \phi(t))} + \Omega^*(t)) d_1, \\ i\dot{d}_1 - \frac{\dot{\phi}(t)}{2} d_1 = \\ (\Omega(t) + \Omega^*(t) e^{-i2(\omega_d t + \phi(t))}) d_0. \end{aligned}$$

Далее используют “приближение вращающейся волны”, смысл которого в том, чтобы отбросить быстро осциллирующие слагаемые в гамильтониане с множителями $e^{-2i\omega_d t}$ и $e^{2i\omega_d t}$. При не очень больших амплитудах возбуждающих сигналов $\Omega \ll \omega$, эти слагаемые слишком быстро меняются, чтобы оказать существенное влияние на более медленное изменение $d_0(t)$ и $d_1(t)$.

Получается система уравнений уже с постоянными коэффициентами

$$\begin{aligned} i\dot{d}_0 + \frac{\dot{\phi}(t)}{2} d_0 = \\ \Omega^*(t) d_1, \\ i\dot{d}_1 - \frac{\dot{\phi}(t)}{2} d_1 = \\ \Omega(t) d_0, \end{aligned}$$

которую можно записать через операторы Паули в виде

$$i|\tilde{\psi}(t)\rangle = \left(I(t)\hat{\sigma}_x + Q(t)\hat{\sigma}_y - \frac{\dot{\phi}(t)}{2}\hat{\sigma}_z \right) |\tilde{\psi}(t)\rangle,$$

$$\text{где } |\tilde{\psi}(t)\rangle = \begin{pmatrix} d_0(t) \\ d_1(t) \end{pmatrix}.$$

Такой гамильтониан отвечает вращению вектора состояния кубита в представлении сферы Блоха вокруг оси $\vec{\Omega} = \{I(t), Q(t), \dot{\phi}(t)/2\}$ с угловой скоростью $|\vec{\Omega}|$.

Наиболее часто используются микроволновые импульсы, отвечающие вращениям на углы $\pi/2$ и π вокруг осей x или y , однако реально любая однокубитная операция может быть реализована правильным подбором управляющих сигналов $I(t)$, $Q(t)$, или $\phi(t)$.

39.6 Декогеренция в сверхпроводниковых кубитах

В отличие от других типов кубитов, сверхпроводниковые кубиты имеют сравнительно низкие времена когерентности. Это означает, что информация, хранящаяся в сверхпроводниковом кубите, может испортиться. Для сверхпроводниковых кубитов декогеренцию можно понимать как неконтролируемое воздействие на кубит со стороны флюктуаций напряжения или тока. Природа флюктуаций может быть очень разной. Наиболее проблематичными являются флюктуации микроскопических зарядов и магнитных моментов внутри самой микросхемы.

Как правило различают шумы и наводки, которые могут привести самопроизвольному возбуждению или релаксации кубита, а также к переходу в недопустимые состояния типа уровней $|2\rangle$ и выше, и шумы, которые влияют на частоту переходов между уровнями кубита.

В самой простой, марковской модели модели шума, события декогеренции независимы во времени и пространстве. Флюктуационный сигнал $S_f(t)$, связывающийся с трансмоном посредством члена гамильтониана типа $\hat{V} = S_f(t)\hat{a}^\dagger\hat{a}$, будет случайным образом менять частоту кубита. Это приведет к набегу фазы

$$\Delta\phi(t) = \int_0^t S_f(t')dt'$$

на кубите. Если разложить флюктуационный сигнал по спектральным компонентам (то есть представить в виде Фурье-компонент), то получим

$$\Delta\phi(t) = \int_{-\infty}^{+\infty} \frac{S_f(\omega)}{\omega} \cos\left(\frac{\omega t}{2}\right)d\omega.$$

Таким образом, наиболее существенным с точки зрения случайного набега фазы (то есть дефазировки) является низкочастотный шум. К таким флюктуационным сигналам может относиться, например, тепловая населенность считывающего резонатора.

Другой тип флюктуационного сигнала связывается с трансмоном посредством дополнительного члена в гамильтониане, имеющего вид $\hat{V} = i\Delta\Omega_\downarrow(\hat{a} - \hat{a}^\dagger)$. Такой флюктуационный сигнал приводит к самопроизвольному возбуждению и релаксации кубита под действием внешней силы. Наиболее существенными являются шумы, частота которых отвечает частоте кубита.

Помимо внешних воздействий, которые можно представить как случайное слагаемое в гамильтониане, существуют также и воздействия, которые нельзя представить в таком виде, например обыкновенную релаксацию кубита без сопутствующего теплового возбуждения. Такой канал декогеренции отвечает случайному действию оператора уничтожения \hat{a} . Таковым является, например, релаксация трансмона в управляющую линию, через которую выполняются однокубитные операции.

В общем случае, для всех типов марковского шума можно ввести оператор коллапса \hat{c} и константу декогеренции γ . В случае релаксации, $\hat{c} = \hat{a}$, $\gamma = 1/T_1$. Для дефазировки $\hat{c} = \hat{\sigma}_z$, $\gamma = 1/2T_\phi$.

Для того, чтобы измерить эффекты декогеренции на сверхпроводниковые кубиты, используют два основных типа эксперимента. Первый типа эксперимента заключается в возбуждении кубита и считывании его состояния спустя время t . Эксперимент повторяется много раз с различным временем задержки t ; результат можно зафитовать экспонентой. Время затухания экспоненты в e будет временем релаксации T_1 .

Такое измерение нечувствительно к дефазировке кубита. Чтобы измерить влияние дефазировки, кубит помещают в состояние суперпозиции при помощи вращения вокруг оси x на угол $\pi/2$, ждут время t , делают вращение вокруг оси z на угол, пропорциональный длительности паузы t , а затем еще поворачивают на угол $\pi/2$ вокруг оси x . В случае идеальной двухкубитной системы зависимость результата измерений от длительности паузы t будет синусоидальной; при наличии декогеренции в марковской модели осцилляции будут затухать экспоненциально, причем константа затухания T_2 может быть выражена как $\frac{1}{T_2} = \frac{1}{2T_1} + \frac{1}{T_\phi}$.

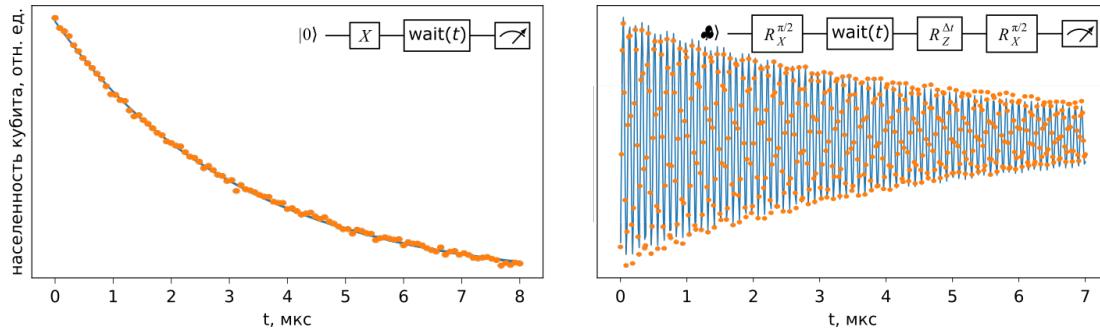


Fig. 39.9: Эксперименты по измерению времени когерентности кубита

Процессы декогеренции ограничивают длину алгоритмов, которые могут быть выполнены на сверхпроводниковом квантовом процессоре. Если хотя бы одно событие декогеренции произойдет в течение алгоритма, то результат выполнения может оказаться неправильным. Важно, что считывание в квантовом компьютере имеет принципиально дискретный характер: из-за этого в принципе возможно говорить о дискретных ошибках, а не о непрерывных ошибках, как в аналоговых компьютерах. Кроме того, как и в обычных цифровых компьютерах, в квантовых компьютерах возможна коррекция ошибок. Для того, чтобы измерить вероятность ошибки напрямую, используют рандомизированное тестирование (randomized benchmarking) квантовых вентилей.

Идея рандомизированного тестирования заключается в том, чтобы выполнять случайную последовательность операций, а зачем сравнивать результат ее выполнения с теоретическим. Существует ряд различных вариаций рандомизированного тестирования, которые отличаются тем, из какой совокупности берутся случайные последовательности. В самом простом случае используют вентили из группы Клиффорда – это такое подмножество логических операций, которые не образуют универсальный набор, но зато алгоритмы целиком состоящие только из таких вентилей можно быстро просимулировать на классическом компьютере даже при большом количестве кубитов. Кроме того, часто используют только такие последовательности вентилей, в конце которых система оказывается не в состоянии суперпозиции, а в определенном состоянии, например в состоянии $|0 \dots 0\rangle$. Ошибки приводят к тому, что вероятность $\langle F_n \rangle$ получить состояние $|0 \dots 0\rangle$ экспоненциально убывает с увеличением длины последовательности вентилей N :

$$\langle F_n \rangle_{\text{seq}}(N) \sim e^{-F_1 n}$$

39.7 Двухкубитные вентили

Для трансмонов существует целый ряд способов реализации двухкубитных операций. Как правило, для этого трансмоны связывают емкостным способом, напрямую или через элемент связи. Мы будем рассматривать двухкубитную операцию iSWAP, реализуемую при помощи параметрического изменения эффективной константы связи между кубитами.

Так же, как в случае кубита, связанного с резонатором, гамильтониан двух связанных трансмонов описывается в первом приближении соотношением

$$\hat{H}/\hbar = \left(\omega_1 + \frac{1}{2}\delta_q (\hat{a}^\dagger \hat{a} - 1)\right) \hat{a}^\dagger \hat{a} + \left(\omega_2 + \frac{1}{2}\delta_r (\hat{b}^\dagger \hat{b} - 1)\right) \hat{b}^\dagger \hat{b} + \chi \left(\hat{a}^\dagger \hat{a} + \frac{1}{2}\right) \left(\hat{b}^\dagger \hat{b} + \frac{1}{2}\right),$$

где \hat{a}, \hat{a}^\dagger – операторы уничтожения и рождения одной степени свободы (трансмиона), а \hat{b}, \hat{b}^\dagger – операторы другой степени свободы (трансмиона).

Принцип работы элемента связи в том, что вместо одного джозефсоновского контакта, в нем есть два джозефсоновских контакта, соединенных параллельно. Такая цепь называется ПТ-СКВИД, или просто СКВИД

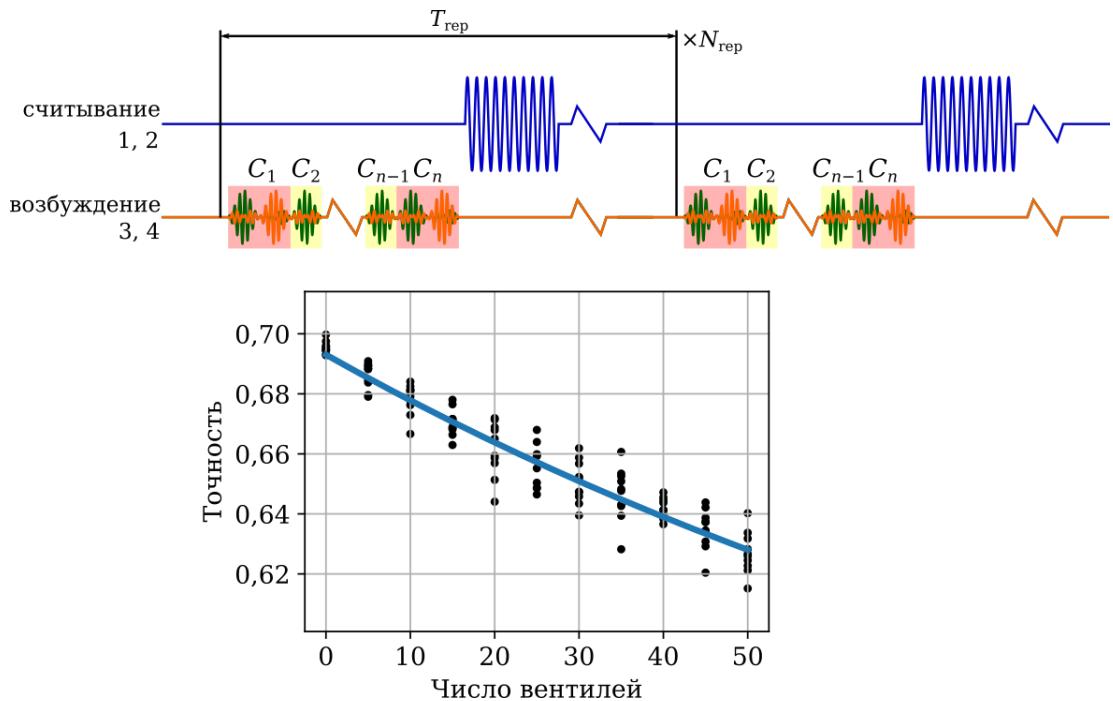


Fig. 39.10: Схема микроволновых импульсов и пример результатов рандомизированного тестирования вентиляй из группы Клиффорда. Каналы 1 и 2 генератора сигналов произвольной формы подключены к линии считывания, каналы 3 и 4 – к линии управления кубитом. Измерена населенность кубита после случайных последовательностей вентиляй разных длин; константа экспоненциального затухания населенности дает точность в расчете на один вентиль $F_1 = 0.993$.

(Superconducting Quantum Interference Device). Если у нас есть два джозефсоновских контакта в кольце, то такая структура будет вести себя по отношению к внешней цепи так же, как один контакт, но с управляемым критическим током. Управление критическим током можно осуществлять при помощи магнитного потока Φ_x , который проходит через кольцо СКВИДа. Соотношение для критического тока выглядит следующим образом:

$$I_{\text{CSQUID}}(\Phi_x) = \sqrt{(I_{\text{Cc}1} + I_{\text{Cc}2})^2 \cos^2(\pi\Phi_x/\Phi_0) + (I_{\text{Cc}1} - I_{\text{Cc}2})^2 \sin^2(\pi\Phi_x/\Phi_0)}.$$

Таким образом, элемент связи представляет собой слабо нелинейный LC-контур с перестраиваемой индуктивностью, а значит и частотой. В зависимости от его частоты будет менять эффективная константа связи между кубитами χ , а также – в меньшей мере – и остальные параметры системы.

Когда нет двухкубитной операции, желательно, чтобы кубиты не взаимодействовали, то есть $\chi = 0$. Все константы этого гамильтониана зависят от внешнего параметра – потока в СКВИДе элемента связи Φ_x . При быстром изменении этого параметра появляется дополнительное (диабатическое) слагаемое

$$\hat{V} = i\hbar\hat{U} \frac{\partial\hat{U}^\dagger}{\partial\Phi_x} \frac{\partial\Phi_x}{\partial t} \sim \Omega (\hat{a}^\dagger + \hat{a}) (\hat{b}^\dagger + \hat{b}).$$

Мы не будем здесь подробно рассматривать природу этого слагаемого; подробнее об этом можно почитать, например, в работе [RGM+17]. Ограничимся тем, что это взаимодействие является поперечным, то есть xx -взаимодействием. Если модулировать поток в СКВИДе в резонансе с разностью частот между кубитами ($n\omega_m = \omega_1 - \omega_2$, где n – номер гармоники), то возбуждение с одного кубита будет переходить на другой, и обратно.

Унитарная матрица этого вентиля имеет вид

$$\hat{U}(t) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\Omega_g t & i\sin\Omega_g t & 0 \\ 0 & i\sin\Omega_g t & \cos\Omega_g t & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Помимо вентиля iSWAP, на трансмонах делают CPHASE, который физически можно реализовать импульсным увеличением константы χ , или, например, как IBM Quantum Experience – при помощи кросс-резонансных вентилей.

39.8 Что почитать для дальнейшего изучения вопроса?

- Introduction to Quantum Computing and Quantum Hardware
- A Quantum Engineer's Guide to Superconducting Qubits [KKY+19]

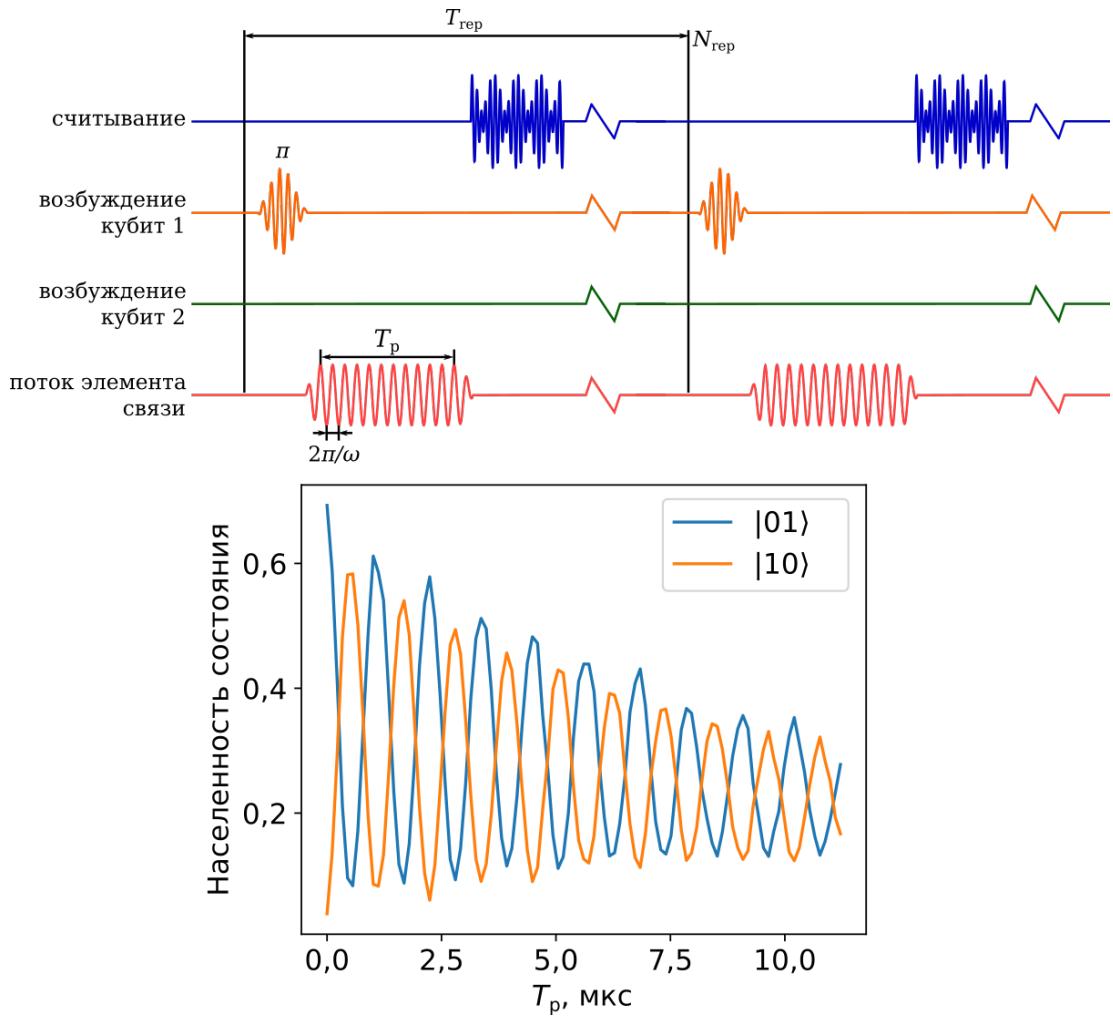


Fig. 39.11: Последовательность сигналов, которую нужно подать, чтобы наблюдать осцилляции населенности между кубитами, и результат из наблюдения. iSWAP получается при длительности вентиля $T_p = 500$ ns

О БЛОКЕ “КВАНТОВОЕ МАШИННОЕ ОБУЧЕНИЕ”

Этот блок включает в себя:

- общий рассказ о видах квантового машинного обучения;

Продвинутые темы блока дополнительно рассказывают:

- об алгоритме HHL;
- о квантовом алгоритме ближайшего соседа;

О КВАНТОВОМ МАШИННОМ ОБУЧЕНИИ

Квантовое машинное обучение – это на самом деле огромная область, и наш курс охватывает лишь небольшую ее часть. В этой обзорной лекции мы попробуем взглянуть на картину в целом и посмотреть, какие есть направления в этой интересной и динамично развивающейся области.

Note: Эта статья является исключительно обзорной – в ней будет много красивых картинок, но мы не будем вдаваться в формулы, которые могут эти картинки объяснить. Цель лекции – познакомить читателя с тем, как машинное обучение может быть применено в разных задачах, связанных с квантовыми вычислениями и квантовой физикой.

41.1 Виды квантового машинного обучения

Как мы помним из [лекции по машинному обучению](#), в таких задачах у нас есть две важных составляющих – данные и алгоритм. В случае квантового машинного обучения у нас появляется некоторая “вариативность”:

- квантовые данные
- квантовый алгоритм
- классические данные
- классический алгоритм

На базе этого можно построить классификацию машинного обучения.

Когда мы имеем классические данные и классический алгоритм, то это как раз обычное машинное обучение, рассмотренное во вводной лекции курса. А вот остальные виды мы рассмотрим далее.

41.2 Квантовые данные и классический алгоритм

Классическое машинное обучение применительно к квантовым данным – это очень активно развивающаяся область квантовой физики и квантовой химии. Главный вопрос в данном случае – что именно мы понимаем под “квантовыми данными”. Но мы будем придерживаться максимально широкой трактовки этого понятия. Далее рассмотрим примеры возможного применения ML в задачах квантовой физики и химии.

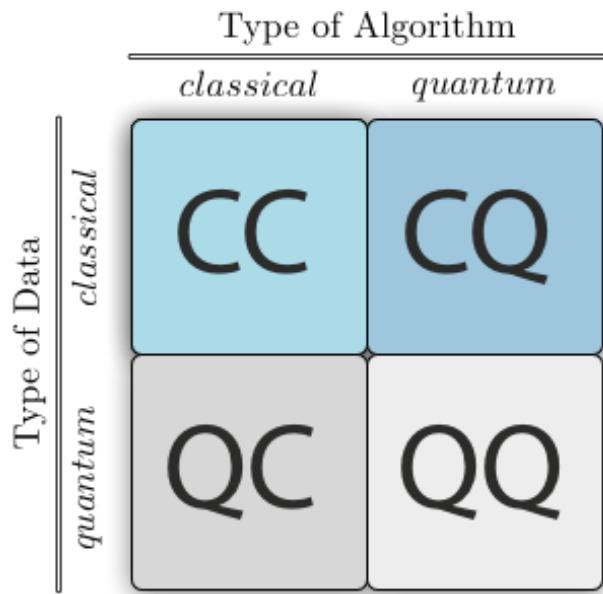


Fig. 41.1: Виды машинного обучения в зависимости от типа данных и алгоритма.

41.2.1 NQS

Neural Quantum States это очень перспективное улучшение квантового Монте-Карло – популярного метода приближенного решения задач квантовой физики. Мы будем подробно рассматривать проблему нахождения энергетических спектров квантomeханических систем, а также моделирования квантовой динамики в блоке о проблемах квантового машинного обучения. Пока нам достаточно знания того, что эти задачи являются вычислительно трудными и их прямое численное решение очень быстро становится невозможным по мере увеличения размера системы и числа взаимодействующих частиц. В подходе **NQS**, впервые опубликованном в журнале *Science* [CT17], предложено моделировать волновую функцию Ψ физической системы при помощи глубокой нейронной сети. Это дает ряд преимуществ над обычным квантовым Монте-Карло в плане точности моделирования, а также позволяет явно симулировать *time-dependent* уравнение Шредингера. Относительно другого популярного метода, который называется *Matrix Product State* **NQS** дает лучшую масштабируемость и меньшую аппроксимацию сложность, которая оказывается линейной по числу частиц в физической системе.

41.2.2 CNN над потенциалами

Другим интересным подходом является применение *Convolutional Neural Networks* (*CNN*) – специального класса глубоких нейронных сетей, разработанных для обработки изображений, к физическим потенциалам. Дело в том, что трехмерный энергетический потенциал можно представить как 2D изображение. Например, можно взять периодическую потенциальную энергию, созданную атомами в кристаллической решетке и рассмотреть, как будет вести себя электрон в таком потенциале.

Существует ограниченный набор частных случаев, когда эта задача может быть решена аналитически. Используя такие точные решения, мы составляем обучающую выборку, тренируем *CNN*, а потом можем применять ее к другим, более сложным потенциалам.

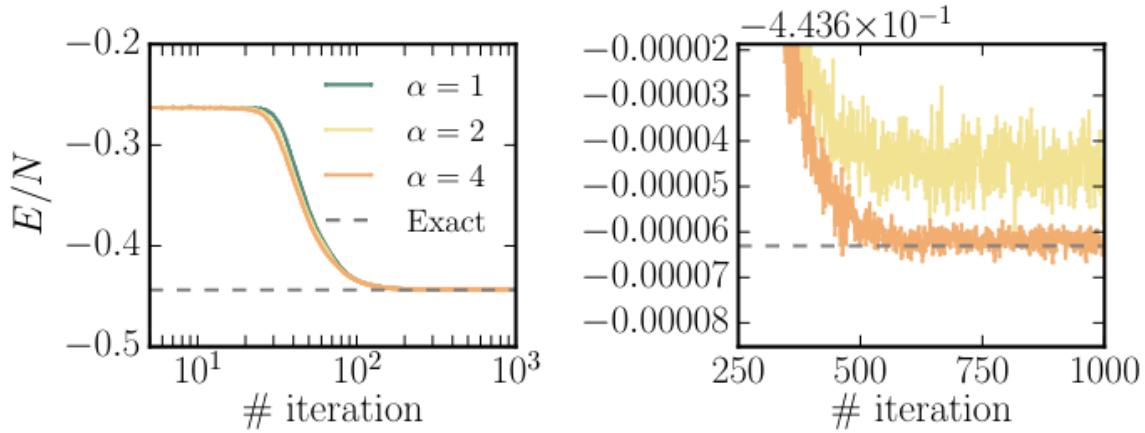


Fig. 41.2: Сходимость NQS по энергии в зависимости от числа эпох обучения. По вертикали значение энергии на один спин, по горизонтали эпохи обучения. Пунктиром отмечено точное значение, полученное прямой диагонализацией гамильтониана. Слева общий вид, справа – сверхтонкая сходимость вблизи энергии основного состояния. Источник [CT17].

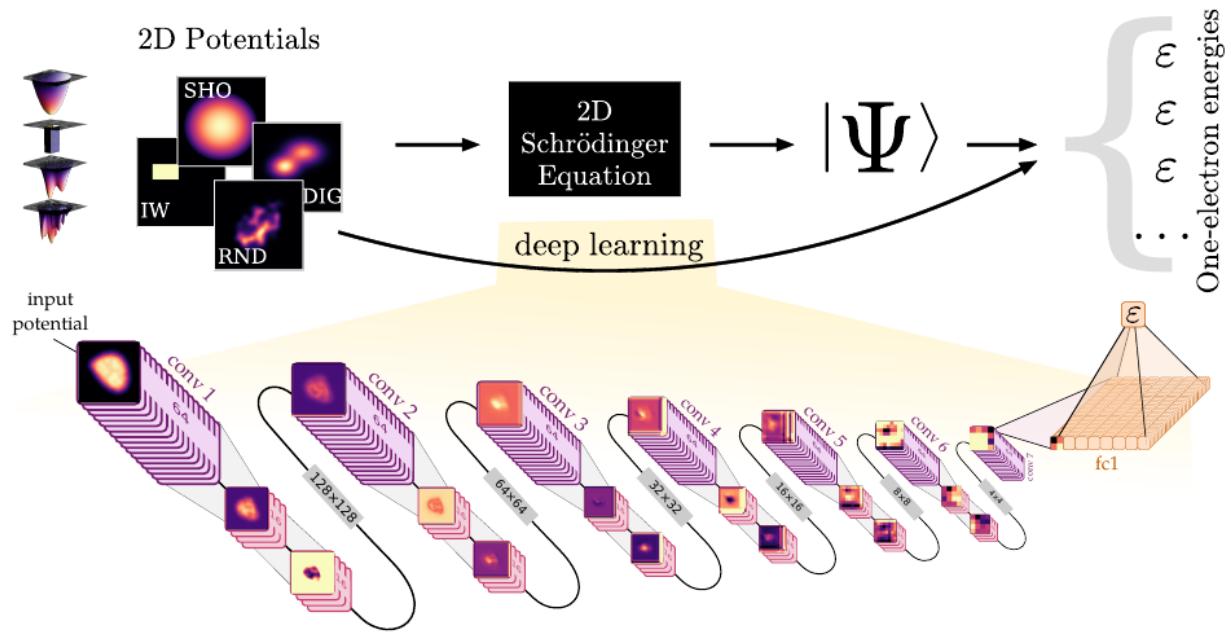


Fig. 41.3: Иллюстрация применения CNN на энергетических потенциалах. Источник [MST17].

41.2.3 Решение уравнения Шредингера

Другим примером решения задачи об энергетических уровнях является применение глубокой нейронной сети, которая на вход принимает *Slater determinants*, *Jastrow factor* и другие известные способы приближенного описания квантомеханических систем. А на выходе из такой глубокой нейронной сети получаем волновую функцию системы. Таким образом, получается объединить все преимущества хорошо изученных физических представлений с высокой экспрессивностью глубоких нейронных сетей [HSchatzleNoe20].

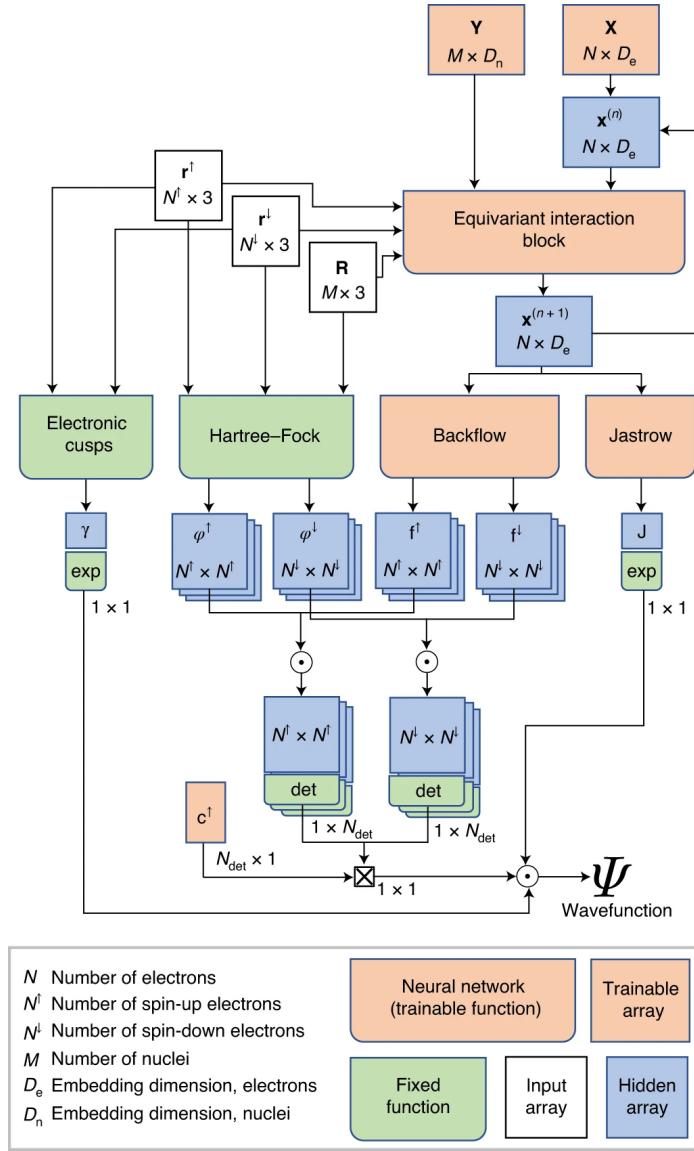


Fig. 41.4: Архитектура подхода *PauliNet*. Источник [HSchatzleNoe20].

Такой подход дает фантастическую точность по сравнению с другими подходами и выглядит очень перспективно, так как дополнительное позволяет получить очень хорошую масштабируемость из-за возможности снизить число *slater determinants*.

41.2.4 RNN для моделирования волновых функций

Другой похожий на *NQS* подход – это использовать для аппроксимации волновой функции рекуррентные нейронные сети вместо машин Больцмана.

В данном подходе мы строим “последовательную” волновую функцию многочастичной квантовой системы, “подавая” туда по одной каждую частицу. А так как нейронная сеть – рекуррентная, мы по сути последовательно строим волновую функцию многочастичной системы.

41.2.5 Квантовая томография

Мы немного рассказывали о задаче квантовой томографии в [лекции про смешанные состояния](#). Правда, там привели пример использования метода максимизации правдоподобия для решения задачи восстановления состояния по измерениям. Оказывается, что для этих целей можно использовать глубокие нейронные сети. Один из таких подходов описан в [AMunozNK21]. Суть подхода в том, что используются генеративные сети с условием – *Conditional GAN* (C-GAN) для восстановления состояния. Для создания матрицы плотности используется генератор, а для вычисления ошибки относительно реальных результатов измерений – дискриминатор.

41.2.6 Заключение к подразделу

В целом, применение классических алгоритмов машинного обучения к задачам квантовой физики и квантовой химии – очень перспективная область. Мы еще немного коснемся этой темы, когда будем рассматривать решение задачи о собственных значениях гамильтонианов. Однако, эта тема выходит за рамки нашего курса.

41.3 Классические данные и квантовый алгоритм

Теперь перейдем к ситуации, когда мы рассматриваем классические данные (задачи классификации и регрессии) и применяем к ним машинное обучение, реализуемое на квантовом компьютере как набор операций над кубитами.

41.3.1 HHL

Один из самых известных алгоритмов квантового машинного обучения – это алгоритм **HHL** [HHL09], которому в нашем курсе [посвящена отдельная продвинутая лекция](#). Это алгоритм решения системы линейных уравнений за $O(N \log N)$ операций. Так как к решению системы уравнений можно свести огромное число задач реального мира, этот алгоритм обладает огромным потенциалом. К сожалению, сегодня его практическое применение сильно ограничено несколькими вещами:

- необходимость эффективной генерации начального состояния – без этого достижения превосходства **HHL** невозможно;
- высокие требования к точности операций, а также необходимость в большом числе кубитов.

Этот алгоритм является примером чисто квантового алгоритма, где абсолютно все операции выполняются на квантовом компьютере.

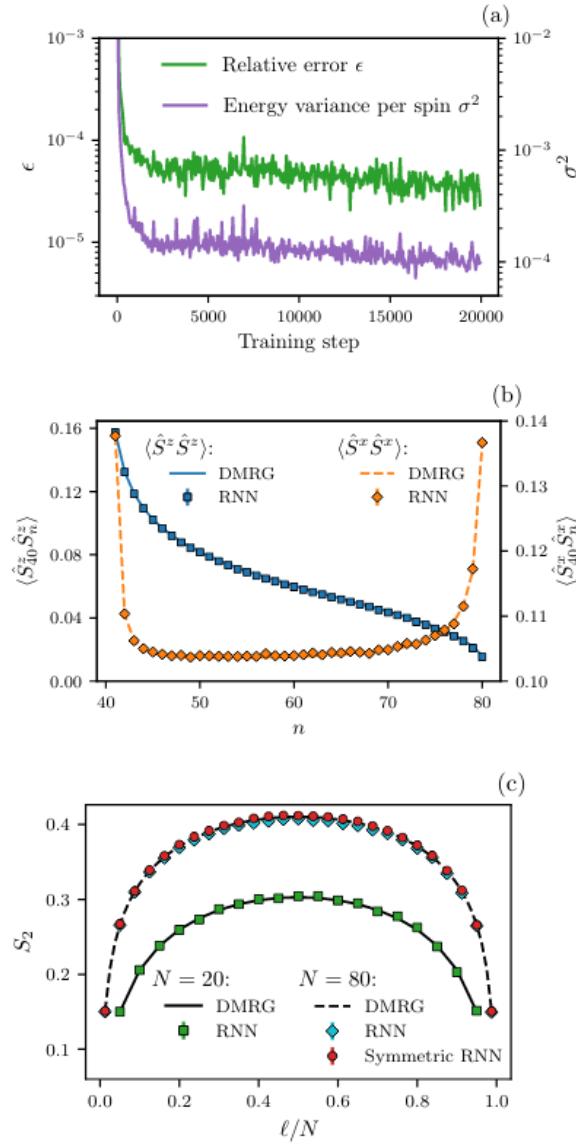
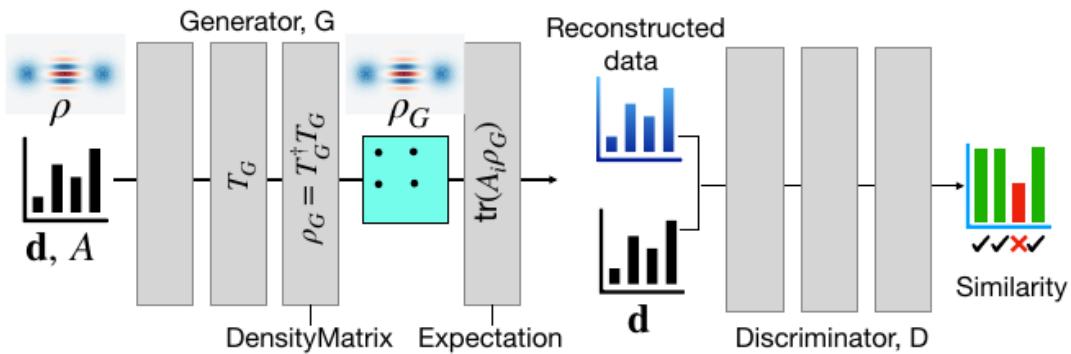


Fig. 41.5: Результаты RNN-модели для аппроксимации волновых функций. Можно получать не только значение энергии, но и ожидаемые значения других операторов, например, операторов спина или операторов спиновых корреляций. Источник [HAGH+20]. На графике (а) приведена зависимость энергии от эпохи обучения. На графике (б) приведены зависимости собственного значения операторов спиновых корреляций 40-го (центрального) спина с n -ым от номера n . На графике (с) приведено распределение значений оператора спина по длине цепочки спинов.

Fig. 41.6: Иллюстрация подхода $C\text{-GAN}$.

41.3.2 Quantum k-NN

Другой пример полностью квантового алгоритма – это модификация классического алгоритма ближайшего соседа, с той лишь разницей, что для вычисления расстояний между точками в N -мерном пространстве мы используем квантовый компьютер. Этот алгоритм называется *Quantum k-NN* (*Quantum k Nearest Neighbours*). Сегодня существует довольно много потенциально эффективных реализаций этого алгоритма, которые отличаются в основном тем, какая используется метрика расстояния в гильбертовом пространстве. Одна из реализаций [рассмотрена в продвинутой лекции нашего курса](#). К сожалению, практическое использование таких алгоритмов сегодня ограничено тем, что для них требуется эффективная квантовая память – *quantum Random Access Memory*.

41.3.3 Заключение к подразделу

Сегодня мы живем в так называемой *NISQ* (*Noisy Intermediate-Scale Quantum*) эпохе, то есть когда у нас есть квантовые компьютеры лишь ограниченного размера и с относительно большим уровнем шумов. А чисто квантовые алгоритмы квантового машинного обучения очень требовательны именно к точности вычислений. Также в таких алгоритмах большой проблемой является постоянная необходимость сложной операции перевода классических данных в квантовые. В теории, эта проблема исчезнет с появлением *qRAM*, однако сегодня такая “квантовая память” еще не существует и даже пока нет однозначного понимания того, как именно ее делать. Все это сильно ограничивает потенциал полностью квантовых подходов. Далее мы рассмотрим более перспективный в *NISQ*-эпоху подход – гибридное квантово-классическое машинное обучение.

41.4 Гибридное обучение

Большая часть из оставшихся лекций нашего курса будет посвящена как раз гибридным методам квантового машинного обучения. В этих методах часть алгоритма реализуется в виде квантовых вентилей, а часть выполняется на классическом компьютере.

41.4.1 VQC

Variational Quantum Circuits, или просто вариационные схемы, – это одно из центральных понятий в гибридном квантово-классическом обучении. Основная идея заключается в том, что мы используем квантовую операцию, которая задается каким-то классическим параметром. Обычно это одна или несколько операций “вращений” на сфере Блоха, про которые мы говорили в лекции про квантовые гейты. При этом вариация классического параметра осуществляется на классическом компьютере, например, при помощи градиентного спуска.

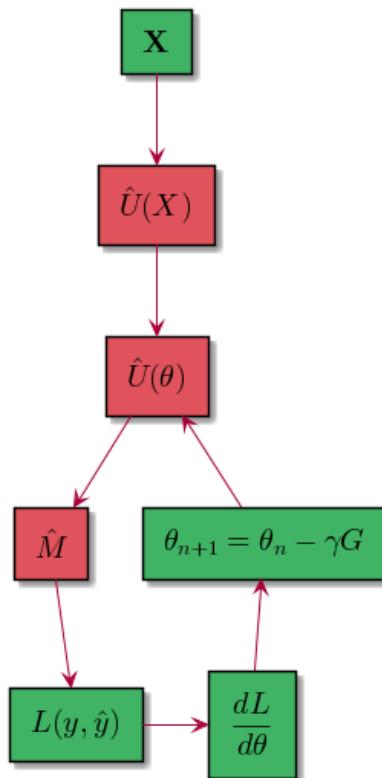


Fig. 41.7: Схема работы вариационной квантовой схемы.

В нашем курсе теме вариационных схем будет посвящено несколько лекций:

- *вариационные квантовые схемы*
- *градиенты VQC*
- *продвинутая лекция по градиентам VQC*

Также мы часто будем возвращаться понятию VQC в теме квантовых нейросетей, а также в блоке про решение задачи собственных значений при помощи алгоритма *Variational Quantum Eigensolver*.

41.4.2 Квантовые нейросети

Квантовые “нейронные сети” – это как раз пример, когда мы комбинируем вариационные слои вместе с обычными слоями нейронных сетей.

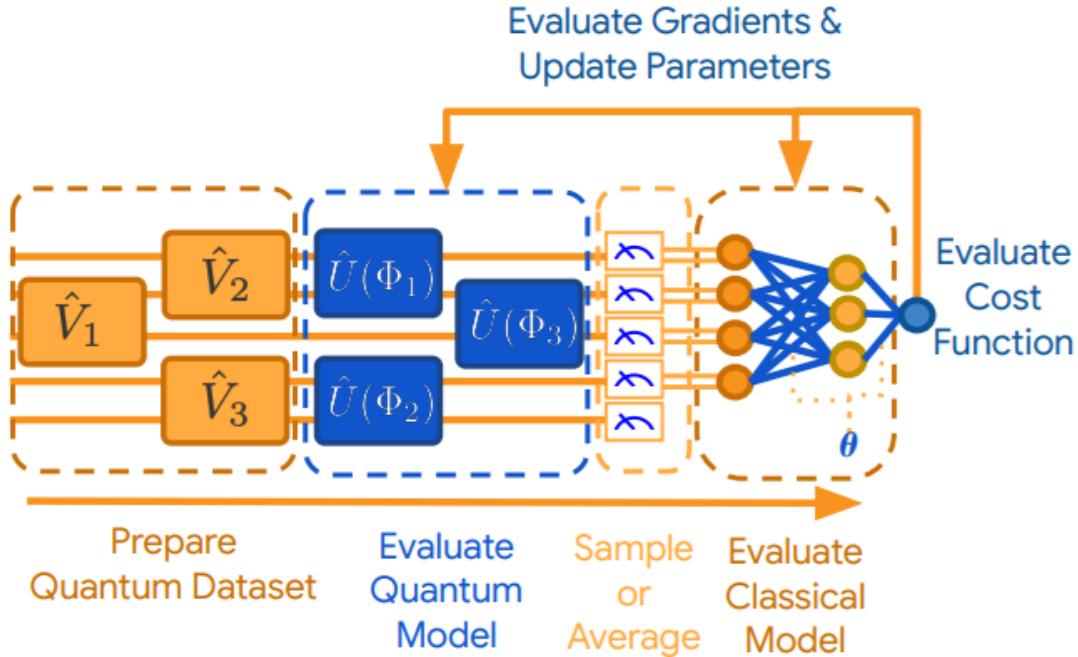


Fig. 41.8: Пример гибридной квантовой нейросети. Источник [BVM+21].

На картинке выше приведен пример комбинации вариационных квантовых схем и классических слоев обычных глубоких нейронных сетей. При этом оптимизация параметров выполняется при помощи единого процесса *back-propagation*, с той лишь разницей, что для классических и квантовых слоев немного по-разному вычисляется градиент для параметров конкретного слоя. Теме квантовых нейронных сетей в нашем курсе будет посвящен целый блок, где будет, в том числе, рассказано еще и об аналоге сверточного слоя с использованием VQC.

41.4.3 Квантовые ядра

Квантовые ядра и квантовый алгоритм *Support Vector Machine (SVM)* – это другой пример, как можно объединить квантовые гейты с классическими алгоритмами. Математике “под капотом” классического *SVM* у нас будет посвящена отдельная лекция, но если кратко, то суть там в том, что решение задачи об оптимальной разделяющей гиперплоскости (другими словами, об оптимальной классификации) можно выразить через скалярные произведения точек обучающей выборки. Причем не обязательно в исходном пространстве, а в любом гильбертовом пространстве. А как мы помним, квантовые гейты как раз представляют собой операции в гильбертовом пространстве, причем экспоненциально большого размера. Так и появляется идея гибридного *SVM* – мы переписываем скалярные произведения точек как результат измерения параметризованных квантовых схем, где параметры это и есть компоненты каждой из точек. А дальше уже применяем классические методы решения оптимизационной задачи. Этой теме у нас будет посвящена отдельная лекция.

41.4.4 Оптимизация энергии

Еще один интересный и перспективный класс задач, которые можно решать гибридными методами – это оптимизация энергии гамильтониана. Может показаться, что это еще что-то из области квантовой физики, но на самом деле в блоке о проблемах квантового машинного обучения мы покажем, как к задаче поиска основного состояния системы, описанной квантовым оператором – гамильтонианом, можно свести огромное число задач реального мира. Например, это задача Коммивояжера, очень важная в области логистики, или задача о нахождении энергий электронов, которая играет важную роль в разработке лекарств или создания новых материалов.

41.5 Заключение

В этой лекции мы познакомились с тем, каким может быть квантовое машинное обучение. Мы узнали, что бывают:

- классические алгоритмы над квантовыми данными;
- квантовые алгоритмы над классическими данными;
- гибридные алгоритмы.

Именно гибридные алгоритмы наиболее перспективны в *NISQ*-эпоху, и именно им будет посвящена большая часть оставшихся лекций курса.

АЛГОРИТМ ННЛ

Сегодня пришла пора поговорить о знаменитом алгоритме Харроу, Хассидима и Ллойда, более известном как HHL-алгоритме, способном решать системы линейных уравнений.

Очень надеюсь, что к данному занятию у вас уже есть представление об алгоритме фазовой оценки (QPE), использующем обратное квантовое преобразование Фурье, на котором и базируется HHL. Глубокое понимание всех тонкостей этого алгоритма потребует от вас уверенного владения математическим аппаратом. За детальным описанием вы всегда можете обратиться к статьям [DHM+18], [HHL09], и [HZL+17]. Приготовьтесь потратить время и умственные ресурсы, если алгоритм вас зацепит и вы решите в нем как следует покопаться. Мы же поможем вам заинтересоваться, рассмотрим основные принципы и небольшой пример.

Note: Именно HHL-алгоритм произвел настоящую революцию в области квантового машинного обучения. Ведь решение систем линейных уравнений так или иначе находится “под капотом” почти любого известного алгоритма машинного обучения. И действительно:

- классические линейная и логистическая регрессия сводятся именно к этой задаче;
- задача SVM может быть переформулирована в терминах решений систем линейных уравнений;
- задача нахождения обратной матрицы (часто используется в глубоком обучении) внутри обычно решается через решение линейной системы;

И это только малая часть примеров!

Так что знакомство с QML не будет полным без ознакомления с этим прекрасным, но очень сложным алгоритмом!

42.1 Задача

Представим обычную систему линейных уравнений:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1 \\ a_{21}x_1 + a_{22}x_2 = b_2 \end{cases}$$

Что в операторной форме можно переписать как:

$$A\vec{x} = \vec{b},$$

где A – эрмитова матрица.

Мы будем решать задачу на квантовом компьютере, то нам нужно перейти к квантовым состояниям:

$$A|x\rangle = |b\rangle$$



Fig. 42.1: Сет Ллойд, профессор МИТ и один из создателей HHL-алгоритма

Чтобы найти искомый вектор $|x\rangle$, все что нам по сути нужно сделать – это найти обратный к A оператор (обозначаемый A^{-1}), который находится из равенства:

$$AA^{-1} = A^{-1}A = I$$

Распишем применительно к нашей задаче поиска вектора $|b\rangle$:

$$\begin{aligned} A|x\rangle &= |b\rangle \\ A^{-1}A|x\rangle &= A^{-1}|b\rangle \\ |x\rangle &= A^{-1}|b\rangle \end{aligned}$$

Оказывается, что и это все можно провернуть с помощью известных квантовых преобразований. Принципиальный вид нашей схемы представлен следующим образом:

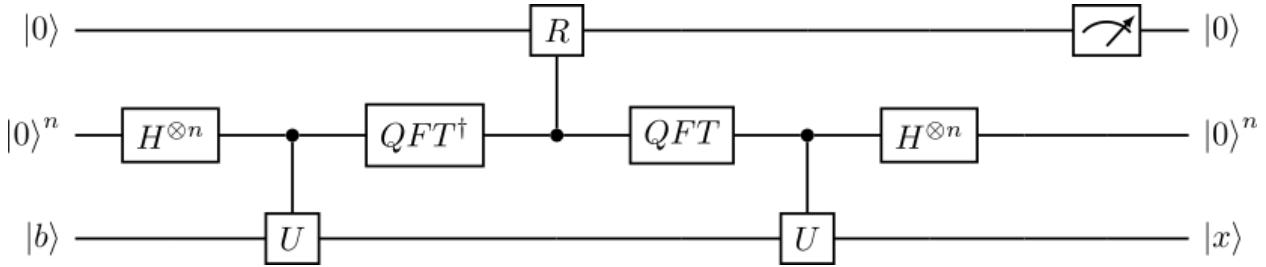


Fig. 42.2: Квантовая схема, реализующая алгоритм HHL

В нижний регистр загружается вектор $|b\rangle$, средний и нижний регистры участвуют в фазовой оценке, а верхний дополнительный кубит нужен для так называемого вращения, обусловленного собственными значениями. Давайте разбираться.

42.2 Реализация HHL

Для начала мы должны подготовить наши регистры по всем квантовым законам: $|b\rangle$ и $|x\rangle$ должны быть пронормированы, а оператор A должен быть эрмитовым. Надеемся, что Вы помните про ортонормированный базис, сферу Блоха, комплексное представление векторов $|0\rangle$ и $|1\rangle\dots$. Если нет, то обратитесь к предыдущим разделам курса.

Мы будем использовать оператор $U = e^{iAt}$, и нужно, чтобы он был обратим — для этого должна быть эрмитовой.

Note: В случае, когда A не является эрмитовой, нужно перейти к эрмитовой матрице C :

$$C = \begin{pmatrix} 0 & A \\ A^\dagger & 0 \end{pmatrix}$$

И рассматривается задача $C\vec{y} = \begin{pmatrix} \vec{b} \\ 0 \end{pmatrix}$ для того, чтобы найти решение $y = \begin{pmatrix} 0 \\ \vec{x} \end{pmatrix}$

Вспомним, что эрмитову матрицу A можно представить в виде суммы собственных векторов, умноженных на собственные значения, т.е. в виде спектрального разложения:

$$\begin{aligned} A &= \sum_{j=0}^{N-1} \lambda_j |u_j\rangle \langle u_j| \\ A^{-1} &= \sum_{j=0}^{N-1} \lambda_j^{-1} |u_j\rangle \langle u_j| \end{aligned}$$

Тогда вектор $|b\rangle$ можно представить через собственные векторы A :

$$|b\rangle = \sum_{j=0}^{N-1} b_j |u_j\rangle$$

Чтобы понять, почему это тоже ключевой момент, давайте вспомним, что значит собственный вектор и собственное значение матрицы.

Note: Собственным вектором $|u\rangle$ оператора A называется такой ненулевой вектор, для которого выполняется:

$$A|u\rangle = \lambda|u\rangle$$

λ – собственное значение оператора A .

Таким образом, искомый вектор $|x\rangle$ – не что иное, как:

$$|x\rangle = A^{-1}|b\rangle = \sum_{j=0}^{N-1} \lambda_j^{-1} b_j |u_j\rangle$$

Итак, фазовая оценка. Мы применяем к кубитам второго регистра матрицы Адамара, тем самым приводим их в суперпозицию. Следом запускаем оператор U :

$$U = e^{iAt} = \sum_{j=0}^{N-1} e^{i\lambda_j t} |u_j\rangle\langle u_j|$$

Для того, чтобы узнать собственное значение оператора U , получения фазы (Quantum Phase Estimation – QPE), результатом которого получится следующее состояние:

$$\begin{aligned} QPE(U, |0\rangle|u\rangle) &= \\ \frac{1}{2^{m/2}} (|0\rangle + e^{2\pi i 2^{m-1}\psi} |1\rangle) \otimes (|0\rangle + e^{2\pi i 2^{m-2}\psi} |1\rangle) \otimes \dots \otimes (|0\rangle + e^{2\pi i 2^0\psi} |1\rangle) \otimes |u\rangle &= \\ \frac{1}{2^{m/2}} \sum_{j=0}^{2^m-1} e^{2\pi i \psi j} |j\rangle |u\rangle &= |\psi_u\rangle |u\rangle \end{aligned}$$

Параметр t это нормировочная константа в случае $U = e^{iAt}$:

$$e^{2\pi i \psi} = e^{i\lambda_j t}$$

$$\psi = \frac{\lambda_j t}{2\pi}$$

Параметр t подбирается с учетом того, что на выходе алгоритма QPE собственные значения λ_j нормализуются к виду $0 \leq \lambda_j \leq 1$ и обычно мы располагаем ограниченным числом кубитов, которое можно использовать для аппроксимации.

Алгоритм обратного квантового Фурье переводит фазу в конкретный вектор.

Принципиальная схема QPE выглядит следующим образом:

Итак, мы подготовились, вспомнили много хорошего, теперь пошагово распишем наш алгоритм.

Стартуем мы со следующим состоянием:

$$|0\rangle_a |0\rangle_r |b\rangle_m$$

Т.е. наше состояние будет храниться в трех регистрах, в каждом из которых содержится столько кубитов, сколько нужно для решения задачи.

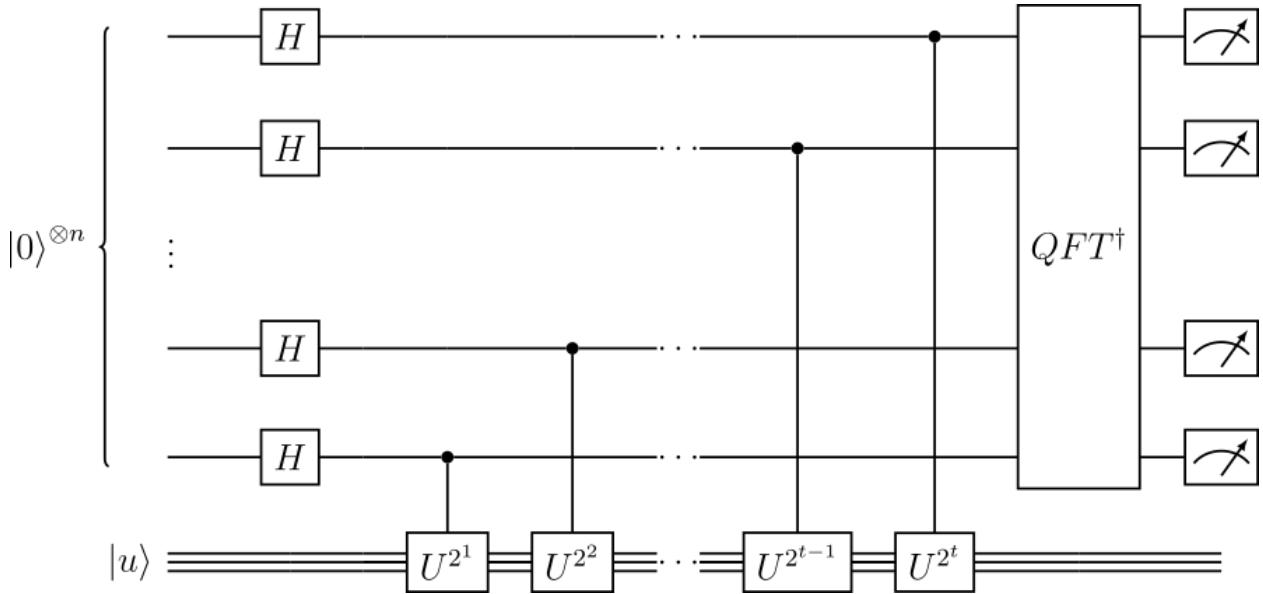


Fig. 42.3: Схема алгоритма QPE

1. Применение QPE с использованием преобразования e^{iAt} , после чего мы получим собственное значение оператора A во втором регистре:

$$|0\rangle_a |0\rangle_r |b\rangle_m \rightarrow \sum_{j=0}^{N-1} b_j |0\rangle_a |\lambda_j\rangle_r |u_j\rangle_m$$

2. Поворачиваем первый кубит (с индексом a), используя специальный оператор вращения R :

$$R|0\rangle_a = \sum_{j=0}^{N-1} \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle_a + \frac{C}{\lambda_j} |1\rangle_a \right),$$

где C – константа, которая должна быть меньше минимального из лямбда: $|C| < \lambda_{min}$ [Почему?].

Переводим первый кубит $|0\rangle_a$:

$$\begin{aligned} \sum_{j=0}^{N-1} b_j |0\rangle_a |\lambda_j\rangle_r |u_j\rangle_m &\rightarrow \\ \sum_{j=0}^{N-1} \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle + \frac{C}{\lambda_j} |1\rangle \right) b_j |\lambda_j\rangle_n |u_j\rangle_m \end{aligned}$$

3. Применяем QPE^\dagger (т.е. обратное получение фазы) и получаем следующее состояние:

$$\sum_{j=0}^{N-1} \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle + \frac{C}{\lambda_j} |1\rangle \right) b_j |0\rangle_n |u_j\rangle_m$$

В конце мы измеряем верхний кубит и если получаем единицу, то знаем, что в нижнем регистре хранится искомый $|x\rangle$ с учетом нормировки:

$$|x\rangle \approx \sum_{j=0}^{N-1} C \left(\frac{b_j}{\lambda_j} \right) |u_j\rangle$$

42.3 Пример

Рассмотрим небольшой, но удобный пример. Удобный в том отношении, что, вообще говоря, алгоритм ННЛ имеет определенное приближение. Если собственные значения не представимы в бинарной форме, то о 100% точности говорить не приходится. Мы также опустим ряд вопросов, связанных с подбором параметров и количества кубитов второго регистра. Главное сейчас – понять, что происходит, и для этого наша матрица эрмитова, все условия подобраны, а преобразования точны. Стоит помнить, что знать заранее значение собственных векторов и собственных значений нам совершенно не обязательно – это нужно лишь для наглядности.

Итак, пусть задача выглядит так :

$$A = \begin{pmatrix} 1 & \frac{3}{5} \\ \frac{3}{5} & 1 \end{pmatrix}$$

$$|b\rangle = |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

$$\begin{cases} x_1 + \frac{3}{5}x_2 = 0 \\ \frac{3}{5}x_1 + x_2 = 1 \end{cases}$$

Собственные значения и соответствующие собственные векторы:

$$\lambda_0 = \frac{2}{5}, |u_0\rangle = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

$$\lambda_1 = \frac{8}{5}, |u_1\rangle = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Зададим параметр t и проанализируем фазу:

$$t = 2\pi \frac{5}{16}$$

$$e^{2\pi i \psi} = e^{i\lambda_j t}$$

$$\psi = \frac{\lambda_j t}{2\pi}$$

$$\frac{\lambda_0 t}{2\pi} = \frac{1}{8}, \quad \frac{\lambda_1 t}{2\pi} = \frac{1}{2}$$

Как мы видим, для перевода угла ψ в векторную форму, нам понадобятся три кубита. После преобразования QPE мы имеем следующее состояние:

$$QPE(|0\rangle_a |0\rangle_r |b\rangle_m) = \sum_{j=0}^1 \frac{1}{\sqrt{2}} |0\rangle_a |\lambda_j\rangle_r |u_j\rangle = \frac{1}{\sqrt{2}} (|0\rangle_a |001\rangle_r |u_0\rangle + |0\rangle_a |100\rangle_r |u_1\rangle)$$

Подберем константу C (как мы помним, она должна быть меньше наименьшего из собственных чисел) и произведем вращение:

$$\begin{aligned} & \frac{1}{\sqrt{2}} \left(\sqrt{1 - \frac{(1/16)^2}{(1/8)^2}} |0\rangle + \frac{1/16}{1/8} |1\rangle \right) |001\rangle_r |u_0\rangle_m + \\ & + \frac{1}{\sqrt{2}} \left(\sqrt{1 - \frac{(1/16)^2}{(1/2)^2}} |0\rangle_a + \frac{1/16}{1/2} |1\rangle_a \right) |100\rangle_r |u_1\rangle_m \end{aligned}$$

В конце мы производим измерение верхнего кубита (с индексом a) и при получении единицы можем быть уверены, что нижний регистр содержит искомое решение с учетом нормировки:

$$\frac{1}{2\sqrt{2}}|1\rangle_a|000\rangle_r|u_0\rangle_m + \frac{1}{8\sqrt{2}}|1\rangle_a|000\rangle_r|u_1\rangle_m$$

$$|x\rangle \approx \frac{1}{2\sqrt{2}}/\left(\sqrt{\frac{17}{128}}\right)|u_0\rangle + \frac{1}{8\sqrt{2}}/\left(\sqrt{\frac{17}{128}}\right)|u_1\rangle$$

QUANTUM K-NEAREST NEIGHBOR

43.1 Введение

Если вы занимались машинным обучением, то, скорее всего, знакомы с классическим алгоритмом k ближайших соседей. Он относительно прост, применяется как в задачах классификации, так и в регрессии. Кстати, классический knn можно вспомнить обратившись к [лекции](#) от ODS по классическому машинному обучению.

Давайте немножко вспомним задачу классификации с использованием классического kNN алгоритма:

У нас есть $\vec{x} \in \{0, 1\}^N$ – **тестовый образец**, а также **тренировочные образцы** – это набор векторов $\vec{v}_i \in \{0, 1\}^N$, в котором каждый вектор уже размечен. И наша задача подобрать правильную метку тестовому образцу.

Тогда мы пройдём следующие шаги:

1. Вычислим похожесть между тестовым образцом и **каждым** тренировочным образцом.
2. Найдем k ближайших к тестовому образцу соседей.
3. Подсчитаем количество представителей для каждого класса и приписываем метку самого часто встречающегося класса к тестовому образцу.

Самой трудозатратным шагом является вычисление расстояния от тестового образца к каждому тренировочному. Также и в квантовой версии алгоритма.

Note: На текущий момент разработано несколько разных версий квантового алгоритма поиска ближайших соседей. Есть версия основанная на расстоянии Хэмминга [[LTG21](#)]:

Расстояние Хэмминга между векторами \vec{x} и \vec{v}_i :

$$d_i = |\vec{x} - \vec{v}_i| = \sum_{j=1}^N (x_j \oplus v_{ij})$$

Но в данной работе мы обратим внимание на версию, которая вычисляет fidelity между двумя векторами состояниями.

Пусть задано Гильбертово пространство n кубитов, размерности $N = 2^n$. Вектор $|\psi\rangle \in H$ – это тестовое состояние, для которого нам нужно определить метку.

Пусть $\{|\phi\rangle : j \in \{0, \dots, M-1\}\} \subset H$ – это набор тренировочных состояний, для которых мы знаем их метки. $M = 2^m, m \in \mathbb{Z}$

Определим fidelity между тестовым состоянием и j -тым тренировочным $|\phi_j\rangle$ как

$$F_j = F(\psi, \phi_j) = |\langle \psi | \phi_j \rangle|^2$$

В свою очередь $F = [F_0, F_1, \dots, F_{M-1}]$ - это таблица fidelity значений между тестовым состоянием и каждым из тренировочных.

Заметим, что задача нахождения k ближайших соседей сводится к задаче нахождения k максимумов значений fidelity из таблицы F . Для этого мы должны реализовать оракул

$$O_{y,A}|j\rangle|0\rangle = |j\rangle|f_{y,A}(j)\rangle,$$

где $f_{y,A}$ - это булева функция определённая как

$$f_{y,A}(j) = \begin{cases} 1 : F_j > F_y \text{ and } j \notin A, \\ 0 : \text{otherwise ,} \end{cases}$$

43.2 Алгоритм

Далее мы алгоритм представленный в работе [BAG21].

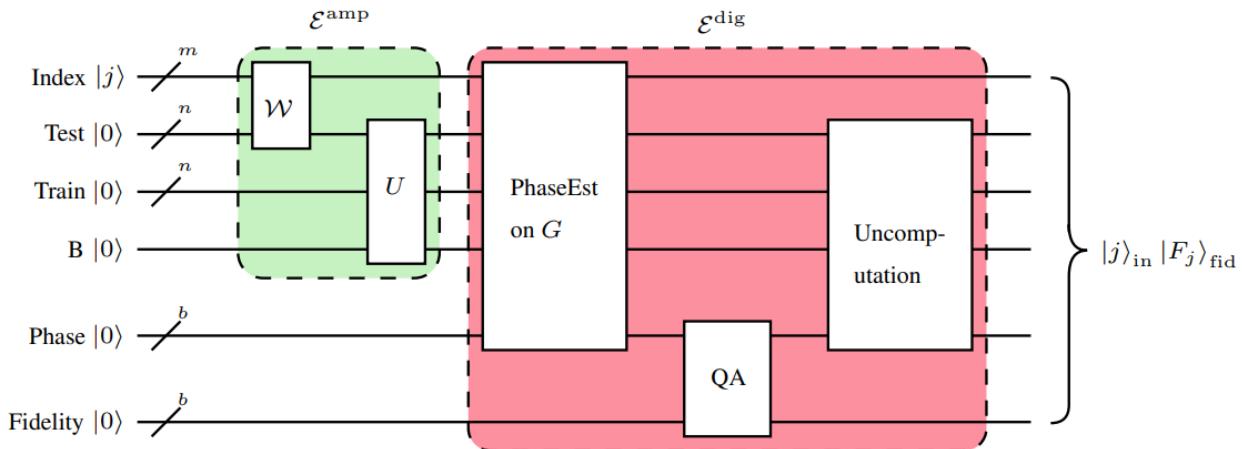


Fig. 43.1: Принципиальная схема QkNN алгоритма. Взято из работы [BAG21].

Квантовый алгоритм поиска k ближайших соседей будет состоять из двух основных шагов:

- Используй оракул $O_{y,A}$ (для алгоритма Гровера) мы находим k состояний $\{|\phi_{j1}\rangle, \dots, |\phi_{jk}\rangle\}$ для которых значение fidelity с тестовым состоянием максимально.
- Найти преобладающую метку среди k найденных состояний и присвоить её тестовому состоянию.

Самой нетривиальной задачей для нас будет построение оракула $O_{y,A}$.

- Вначале нужно составить оператор \mathcal{F} , который выполняет преобразование вида:

$$\mathcal{F}|j\rangle|0\rangle = |j\rangle|F_j\rangle$$

для $j \in \{0, \dots, M-1\}$. $|F_j\rangle$ - это одно из базисных состояний вычислительного базиса (выражающее двоичное представление числа F_j).

- Выполняется преобразование: $\xi^{amp}|j\rangle|0\rangle = |j\rangle|\psi_j\rangle$. В амплитуду состояния $|\psi_j\rangle$ закодирована информация о числе F_j . Делается это с помощью Swap test.

Swap test это применение контролируемой операции Swap, которым можно пользоваться для того, чтобы статистически определять fidelity: $F(\psi, \phi) = |\langle\psi|\phi\rangle|^2$ между двумя произвольными чистыми

состояниями $|\psi\rangle$ и $|\phi\rangle$.

$$\begin{aligned} CSWAP(|0\rangle|\psi\rangle|\phi\rangle) &= |0\rangle|\psi\rangle|\phi\rangle \\ CSWAP(|1\rangle|\psi\rangle|\phi\rangle) &= |1\rangle|\psi\rangle|\phi\rangle \end{aligned}$$

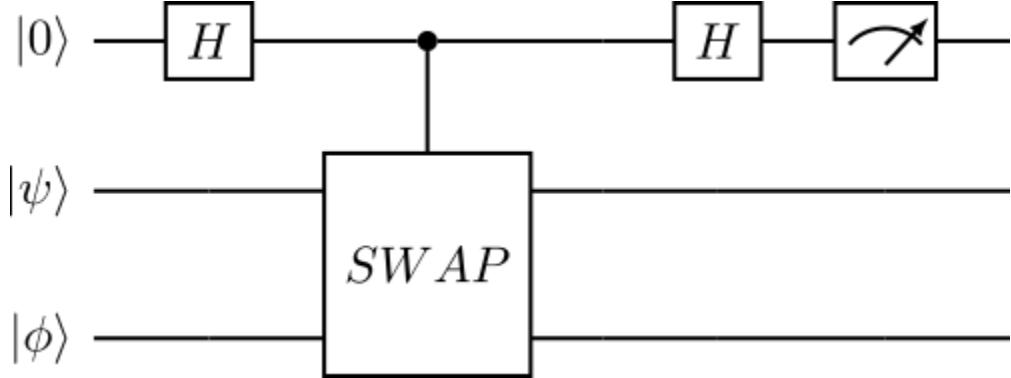


Fig. 43.2: Схема Swap test

- Выполняется преобразование $\xi^{dig}|j\rangle|\psi_j\rangle = |j\rangle|F_j\rangle$

И тогда $\mathcal{F} = \xi^{dig}\xi^{amp}$.

- Берём 2 пары регистров $i_1, f_1; i_2, f_2$. Инициализируются они в форме $|j\rangle_{i_1}|0\rangle_{f_1}|y\rangle_{i_2}|0\rangle_{f_2}$

Применяется \mathcal{F} на каждой паре:

$$\mathcal{F}(|j\rangle_{i_1}|0\rangle_{f_1})\mathcal{F}(|y\rangle_{i_2}|0\rangle_{f_2}) = |j\rangle_{i_1}|F_j\rangle_{f_1}|y\rangle_{i_2}|F_y\rangle_{f_2}$$

- Теперь информация закодирована в регистры и нам нужно реализовать функцию $f_{y,A}$. И пусть \mathcal{C} это оператор, реализующий $f_{y,A}$.

$$\mathcal{C}(|j\rangle_{i_1}|F_j\rangle_{f_1}|y\rangle_{i_2}|F_y\rangle_{f_2}) = |j\rangle_{i_1}|0\rangle_{f_1}|f_{y,A}\rangle_{i_2}|0\rangle_{f_2}$$

Теперь займёмся вопросом конструирования оракула $O_{y,A}$. Просьба держаться за ваши кресла.

Вначале мы подготовим состояние. Но чтобы это сделать нам нужны оракулы \mathcal{V}, \mathcal{W} . Как их implementировать указано в статье, которая указывалась выше.

$$\mathcal{V}|0^n\rangle = |\psi\rangle$$

$$\mathcal{W}|j\rangle|0^n\rangle = |j\rangle|\phi\rangle$$

для всех $j \in \{0, \dots, M-1\}$.

- Инициализируем 4 регистра i, tr, tst, B с соответствующим количеством кубитов в каждом $m, n, n, 1$, где $n = \log(N)$, $m = \log(M)$.

$$|j\rangle_i|0^{\otimes n}\rangle_{tr}|0^{\otimes n}\rangle_{tst}|0\rangle_B$$

- Применяем \mathcal{W}

$$\mathcal{W}(|j\rangle_i|0^{\otimes n}\rangle_{tr}|0^{\otimes n}\rangle_{tst}|0\rangle_B) = |j\rangle_i|\phi_j\rangle_{tr}|0^{\otimes n}\rangle_{tst}|0\rangle_B$$

3. Применяем \mathcal{V}

$$\mathcal{V}(|j\rangle_i|\phi_j\rangle_{tr}|0^{\otimes n}\rangle_{tst}|0\rangle_B) = |j\rangle_i|\phi_j\rangle_{tr}|\psi_j\rangle_{tst}|0\rangle_B$$

4. Применяем swap test между тренировочным регистром tr и тестовым регистром tst , а регистр B будет выступать в качестве контрольного.

$$\frac{1}{2} [(|\phi_j\rangle|\psi\rangle_{tst} + |\psi_j\rangle|\phi\rangle_{tst})|0\rangle_B + (|\phi_j\rangle|\psi\rangle_{tst} - |\psi_j\rangle|\phi\rangle_{tst})|1\rangle_B] = |j\rangle_i|\psi_j\rangle_{tr,tst,B}$$

Определим U как унитарное преобразование, которое объединяет шаги 3-4. Кстати, если мы сейчас произведём измерение регистра B , то будем иметь

$$\begin{aligned} Pr(B=0) &= \frac{1+F_j}{2} \\ Pr(B=1) &= \frac{1-F_j}{2} \end{aligned}$$

На этом шаге информация о *fidelity* теперь закодирована в амплитуды. Теперь же мы должны перевести *fidelity* из амплитуды в число.

5. Теперь мы будем конструировать новый гейт G . Вообще говоря, он описан в работе [MKF19], где вы можете подробнее с ним ознакомиться.

$$G = U_{tr,tst,B} \mathcal{W}_{i,tr} S_{0_{tr,tst,B}} \mathcal{W}_{i,tr}^\dagger U_{tr,tst,B}^\dagger Z_B,$$

где Z_B – это действие гейта Z на регистре B , $S_0 = I - 2|0\rangle\langle 0|$.

6. Текущее состояние $|\psi\rangle_{tr,tst,B}$ может быть представлено в виде композиции двух состояний

$$|\psi_j\rangle = \frac{-i}{\sqrt{2}} (e^{i\pi\theta_j} |\psi_{j+}\rangle - e^{-i\pi\theta_j} |\psi_{j-}\rangle)$$

7. Теперь применяем алгоритм QPE (Quantum Phase Estimation), чтобы перевести значение фазы θ_j в числовое представление.

$$QPE(|\psi_j\rangle) = \frac{-i}{\sqrt{2}} |j\rangle_i [e^{i\pi\theta_j} |\theta_j\rangle_{ph} |\psi_{j+}\rangle_{tr,tst,B} - e^{-i\pi\theta_j} |1-\theta_j\rangle_{ph} |\psi_{j-}\rangle_{tr,tst,B}] = |j\rangle_i |\psi_{j,AE}\rangle_{ph,tr,tst,B}$$

8. Применяем алгоритм квантовой арифметики:

$$|j\rangle |F_j\rangle_{fid} |\psi_{j,AE}\rangle_{ph,tr,tst,B}$$

9. Обнуляем регистры ph, tr, tst, B и получаем $|j\rangle_i |F_j\rangle_{fid}$. На самом деле шаги 5-9 составляют оператор ξ^{dig} , который мы упоминали ранее.

10. Теперь применяем оператор \mathcal{F}

$$|j\rangle_{i1} |F_j\rangle_{f1} |y\rangle_{i2} |F_y\rangle_{f2}$$

11. Добавим кубит Q_1 для выполнения сравнения

$$J|a\rangle|b\rangle|0\rangle = \begin{cases} |a\rangle|b\rangle|1\rangle : a > b, \\ |a\rangle|b\rangle|0\rangle : a \leq b, \end{cases}$$

$$|j\rangle_{i1} |F_j\rangle_{f1} |y\rangle_{i2} |F_y\rangle_{f2} |g(j)\rangle_{Q_1},$$

где

$$g(j) = \begin{cases} 1 : F_j > F_y, \\ 0 : F_j \leq F_y, \end{cases}$$

По кубиту Q_1 мы сможем распознать все индексы j для которых $F_j > F_y$.

12. Обнуляем регистры $i2, f2$.
13. Добавляем ещё один кубит Q_2 для каждого $i_l \in A$, применяя гейт $D^{(i_l)}$

$$D^{(i_l)}|j\rangle|0\rangle = \begin{cases} |j\rangle|1\rangle : j = i_l, \\ |j\rangle|0\rangle : j \neq i_l, \end{cases}$$

на индексах регистра. И в результате получим состояние

$$|j\rangle_{i1}|F_j\rangle_{f1}|g(j)\rangle_{Q_1}|\chi_A(j)\rangle_{Q_2}$$

14. О да... Теперь мы добавляем ещё один кубит Q_3 . Применяем гейт X на кубите Q_2 и гейт Тоффоли с контролирующими (Q_1, Q_2) и целевой Q_3

$$|j\rangle_{i1}|F_j\rangle_{f1}|g(j)\rangle_{Q_1}|\chi_A(j)\rangle_{Q_2}|f_{y,A}(j)\rangle_{Q_3}$$

15. Обнуляем все регистры, кроме Q_3

$$|j\rangle_{i1}|f_{y,A}(j)\rangle_{Q_3}$$

Что ж, вот мы и построили преобразование $O_{y,A}$ которое так хотели

$$O_{y,A}|j\rangle|0\rangle = |j\rangle|f_{y,A}(j)\rangle$$

CHAPTER
FORTYFOUR

О БЛОКЕ “ВАРИАЦИОННЫЕ СХЕМЫ”

Этот блок включает в себя только вариационные квантовые схемы (VQC).

ВАРИАЦИОННЫЕ КВАНТОВЫЕ СХЕМЫ

45.1 Описание лекции

На этой лекции мы впервые познакомимся непосредственно с квантовым машинным обучением. Лекция расскажет:

- в чем заключается идея квантово-классического обучения;
- что такое вариационное машинное обучение;
- как устроена вариационная квантовая схема и как закодировать в нее данные.

45.2 Введение

В течение всего курса мы будем говорить преимущественно о комбинированном квантово-классическом машинном обучении, построенном на базе вариационных квантовых схем. Именно он является наиболее перспективным в эпоху *NISQ*. Давайте забежим немного вперед и посмотрим, как выглядит типичный цикл такого обучения.

Красным цветом на диаграмме помечены блоки, выполняемые на квантовом устройстве, зеленым – те, что считает обычный компьютер. Квантовая часть, которая включает в себя операторы $\hat{U}(X)$ и $\hat{U}(\theta)$, а также измерение наблюдаемой \hat{M} , называется вариационной квантовой схемой. Именно ей посвящена данная лекция.

Но сначала сделаем шаг назад и обсудим в общих чертах идеи, которые лежат в основе квантово-классического обучения.

45.3 Квантово-классическое обучение

Основная идея квантово-классического обучения заключается в том, что в эпоху Noisy Intermediate-Scale Quantum (*NISQ*) у нас нет больших работающих квантовых компьютеров и квантовой памяти. Это сильно ограничивает применение алгоритмов, более быстрых, чем классические. Нам остается только “встраивать” квантовые схемы в классический цикл обучения.

В этом случае мы разделяем классическую и квантовую части алгоритма. Выполняем предварительную обработку и подготовку данных на классическом компьютере, затем “прогоняем” их через квантовую схему. Она должна возвращать нам “классические” данные, а значит, должна включать в себя набор последовательно применяемых операторов и измерение. Со стороны классического компьютера такая схема выглядит как “черный ящик” или “оракул”, с которым уже можно работать. Например, можно варьировать параметры схемы таким образом, чтобы она “обучалась” так же, как “обучаются” нейронные сети или другие алгоритмы классического машинного обучения.

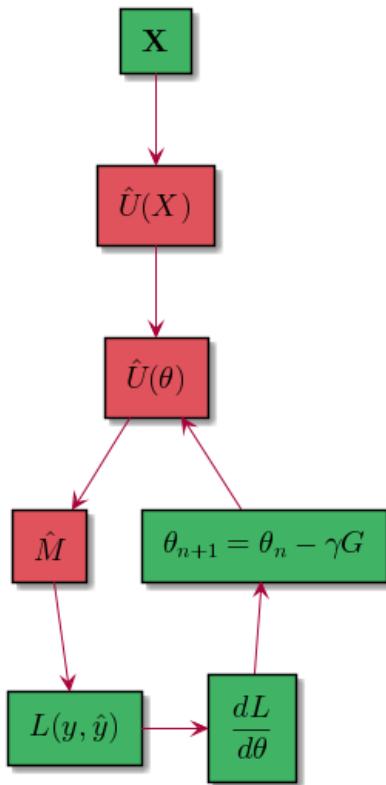


Fig. 45.1: Пример цикла смешанного обучения

Давайте более детально посмотрим на квантовую схему, которую можно использовать в таком подходе.

45.4 Вариационные квантовые схемы

В основе вариационных квантовых схем – Variational Quantum Circuits (**VQC**) – лежит простая идея. Чтобы проиллюстрировать ее, давайте сделаем схему, которая включает в себя набор унитарных операторов и переводит классические данные \mathbf{X} и параметры θ в квантовое состояние $\Psi(\theta, \mathbf{X})$. Затем будем варьировать наше состояние Ψ , меняя параметры θ так, чтобы при его измерении в каком-либо базисе мы получали нужный результат, например, результат классификации входных данных \mathbf{X} .

Разберем написанное выше на примере.

45.4.1 Аппроксимация матрицы Паули X оператором поворота

Из предыдущих лекций мы помним, что оператор \hat{X} работает как квантовый аналог оператора *NOT* – инвертирует значение кубита.

```
import pennylane as qml

dev = qml.device("default.qubit", 1)

@qml.qnode(dev)
def simple_x_gate():
    qml.PauliX(0)
    return qml.expval(qml.PauliZ(0))

print(f"simple_x_gate() = {simple_x_gate()}")
print(simple_x_gate.draw())
```

```
simple_x_gate() = tensor(-1., requires_grad=True)
0: —x—| <Z>
```

С другой стороны, у нас есть оператор $RX(\phi)$, который “вращает” состояние нашего кубита вокруг оси X на угол ϕ . Сделаем параметризованную схему с одним параметром:

```
@qml.qnode(dev)
def vqc(phi):
    qml.RX(phi, wires=[0])
    return qml.expval(qml.PauliZ(0))
```

Теперь попробуем подобрать ϕ так, чтобы параметризованная схема работала точно так же, как оператор \hat{X} :

```
for i in [1.0, 2.0, 3.0, 3.14159265359]:
    print(f"Try {i}: {vqc(i)}")
```

```
Try 1.0: 0.5403023058681398
Try 2.0: -0.4161468365471423
Try 3.0: -0.9899924966004454
Try 3.14159265359: -1.0
```

Получается, что нужный нам угол ϕ составляет ровно π . Это логично, мы могли бы это легко понять из простейших соображений линейной алгебры, либо взглянув на сферу Блоха. Но целью этого примера было проиллюстрировать работу VQC.

Наш процесс “подбора” параметра ϕ крайне примитивный, на следующих лекциях мы рассмотрим, как посчитать градиент параметров квантовой схемы и задействовать всю мощь известных на сегодня методов оптимизации. А пока еще немного подумаем о том, как можно закодировать данные в квантовую схему.

45.5 Кодирование данных в VQC

В конструировании VQC есть две важных части:

- кодирование классических данных в квантовые операторы;
- выбор наблюдаемой для измерений.

Вот как можно закодировать данные.

45.5.1 Кодирование поворотами

Один из самых популярных методов кодирования классических данных в квантовые схемы – это использование операторов поворота $\hat{R}Y$, $\hat{R}Z$. Представим, что мы хотим отображать вектор из двумерного пространства в один кубит $\vec{x} \in \mathbb{R}^2 \rightarrow \mathbb{C}^2$.

```
@qml.qnode(dev)
def angle_vqc(a, b):
    qml.RY(a, wires=[0])
    qml.RZ(b, wires=[0])

    return qml.expval(qml.PauliZ(0))
```

А теперь давайте посмотрим, как такая схема преобразует данные. Для этого сгенерируем набор случайных двумерных данных в диапазоне $[0, 2\pi]$ и применим к каждой точке нашей схемы, затем визуализируем результаты:

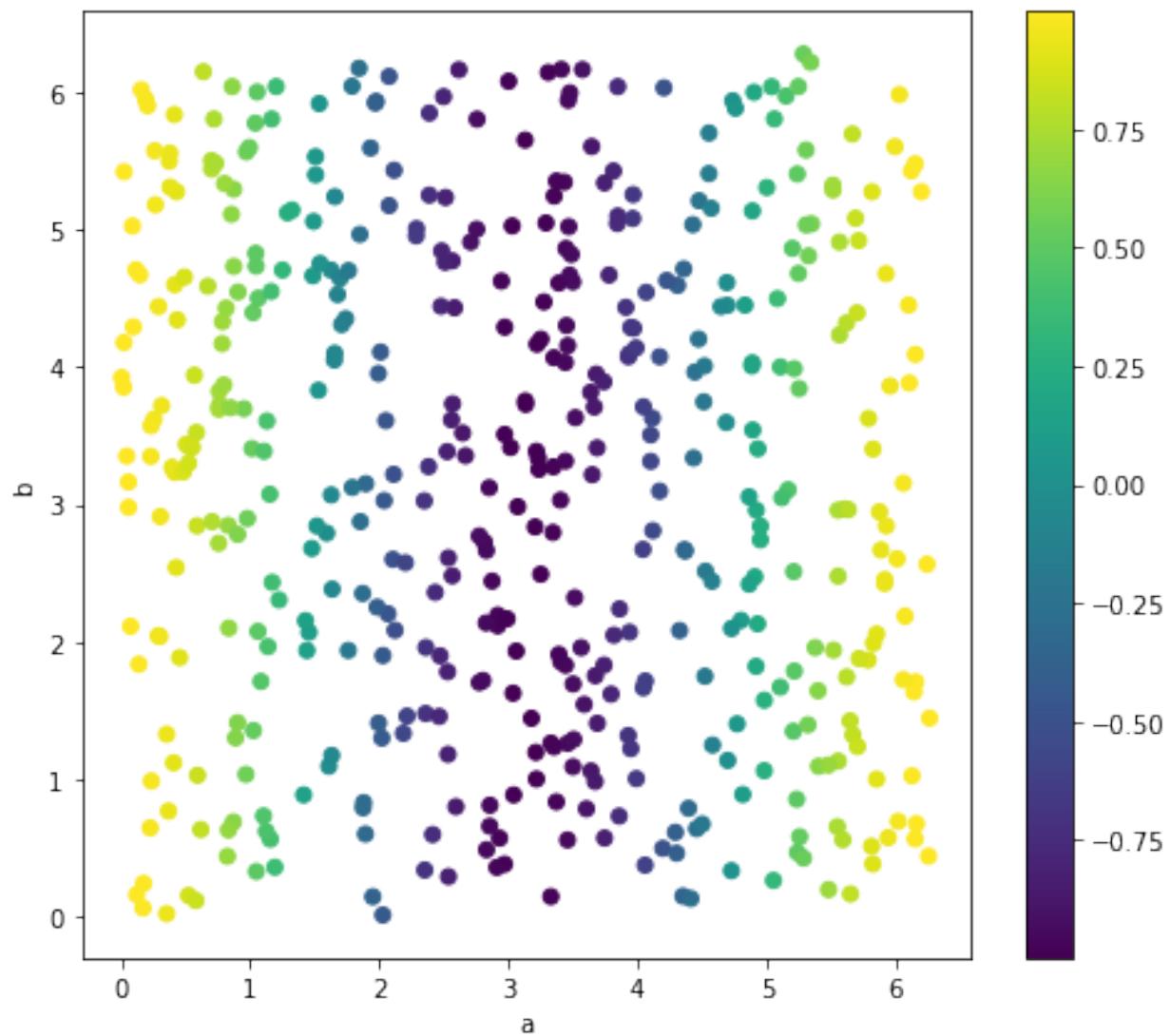
```
import numpy as np
import matplotlib.pyplot as plt

aa = np.random.uniform(0, 2 * np.pi, size=500)
bb = np.random.uniform(0, 2 * np.pi, size=500)

res = [angle_vqc(a, b) for a, b in zip(aa, bb)]

f = plt.figure(figsize=(8, 7))
clb = plt.scatter(aa, bb, c=res)
plt.xlabel("a")
plt.ylabel("b")
plt.colorbar(clb)

f.show()
```



Результат довольно необычный. Предсказать такое довольно сложно и хорошо видна возможность нелинейности VQC схем. В дальнейших лекциях мы будем более подробно разбирать возможные варианты кодирования данных. Кстати, кодирование углами настолько популярно, что в PennyLane для этого есть специальная функция, которая позволяет “повернуть” сразу множество кубитов, используя список из классических данных:

```
@qml.qnode(dev)
def angle_vqc_pl(a, b):
    qml.templates.embeddings.AngleEmbedding([a], wires=[0], rotation="Y")
    qml.templates.embeddings.AngleEmbedding([b], wires=[0], rotation="Z")
    return qml.expval(qml.PauliZ(0))
```

Давайте убедимся, что это даст нам ровно тот же результат:

```
res_pl = [angle_vqc_pl(a, b) for a, b in zip(aa, bb)]
print(np.allclose(res, res_pl))
```

```
True
```

45.5.2 Важность многокубитных гейтов

При кодировании параметров целесообразно использовать многокубитные гейты, так как именно они создают запутанные состояния и раскрывают все преимущества квантовых компьютеров. Давайте попробуем это сделать.

```
dev2 = qml.device("default.qubit", 2)

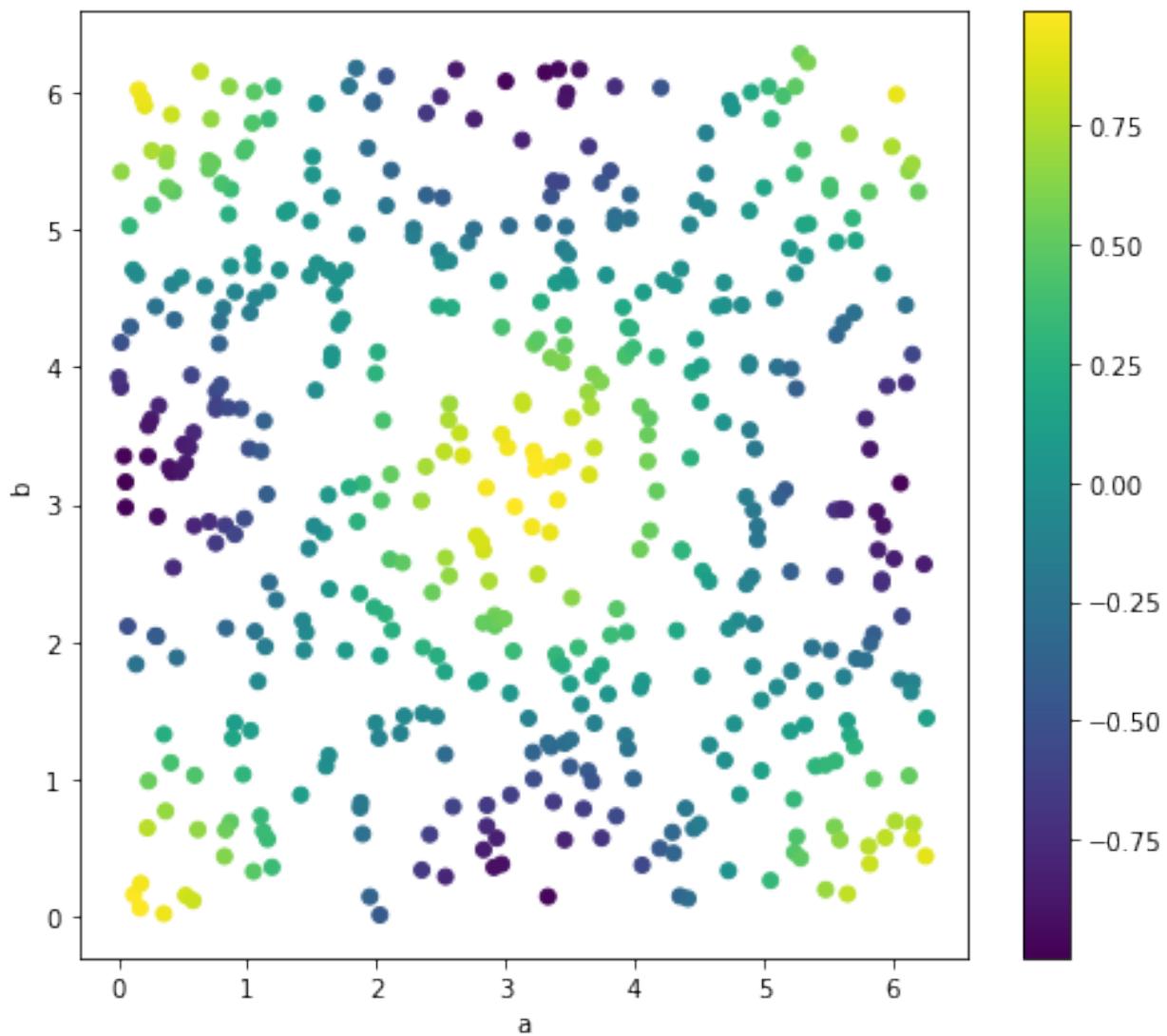
@qml.qnode(dev2)
def cnot_vqc(a, b):
    qml.RY(a, wires=0)
    qml.RY(b, wires=1)
    qml.RZ(a, wires=0)
    qml.RZ(b, wires=1)
    qml.CZ(wires=[0, 1])
    return qml.expval(qml.PauliZ(0) @ qml.PauliZ(1))
```

В этой схеме мы видим оператор $\hat{C}Z$, который создает запутанное состояние. Посмотрим, как такая схема преобразует наше исходное пространство случайных точек.

```
res = [cnot_vqc(a, b) for a, b in zip(aa, bb)]

f = plt.figure(figsize=(8, 7))
clb = plt.scatter(aa, bb, c=res)
plt.xlabel("a")
plt.ylabel("b")
plt.colorbar(clb)

plt.show()
```



Результат выглядит гораздо интереснее. В целом можно сказать, что выбор способа кодирования данных – один из важнейших при проектировании VQC. Мы еще много раз будем обсуждать способы сделать это.

45.6 Что мы узнали из лекции

- В эпоху NISQ мы чаще всего ограничены комбинированием квантового и классического машинного обучения.
- Ключевой элемент такого обучения – вариационные квантовые схемы (VQC). Основная идея VQC:
 - кодируем классические данные в квантовые операторы;
 - измеряем состояние;
 - варьируем параметры на классическом компьютере так, чтобы измерение давало желаемый результат.
- Один из широко применимых методов кодирования – использование операторов вращений $\hat{R}X, \hat{R}Y, \hat{R}Z$

CHAPTER
FORTYSIX

О БЛОКЕ “КВАНТОВЫЕ ГРАДИЕНТЫ”

Этот блок включает в себя обзор способов оценки градиента VQC.

Продвинутая тема блока рассказывает о квантовых градиентах старших порядков.

ГРАДИЕНТЫ КВАНТОВЫХ СХЕМ

47.1 Описание лекции

В этой лекции мы детально разберем, как можно оптимизировать параметры VQC:

- Как выглядит цикл обучения квантовой схемы
- Как работает оценка градиента “под капотом”
 - Метод конечных отрезков
 - Parameter-shift rule

47.2 Введение

Как мы уже говорили ранее, VQC выступают в роли “черных ящиков”, которые имеют параметры и как-то преобразуют поступающие в них данные. В этом случае сам процесс оптимизации параметров выполняется на классическом компьютере. Одними из самых эффективных на сегодня методов решения задач непрерывной оптимизации являются градиентные методы. Для этих методов разработан широкий арсенал эвристик и приемов, который применяется в обучении классических глубоких нейронных сетей. Очень хочется применить весь этот арсенал и для квантового машинного обучения. Но как же посчитать градиент вариационной квантовой схемы?

47.3 Задача лекции

На этой лекции мы рассмотрим простую задачку по оптимизации параметров квантовой схемы и на ее примере увидим, как работают квантовые градиенты. В качестве задачи возьмем известный набор данных “Two Moon” из библиотеки scikit-learn:

```
from pennylane import numpy as np
import pennylane as qml
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
from sklearn.datasets import make_moons

x, y = make_moons(n_samples=50)
```

Для удобства мы сразу переведем метки классов из $\{0, 1\} \rightarrow \{-1, 1\}$, а признаки \mathbf{X} переведем в $[0, \pi]$:

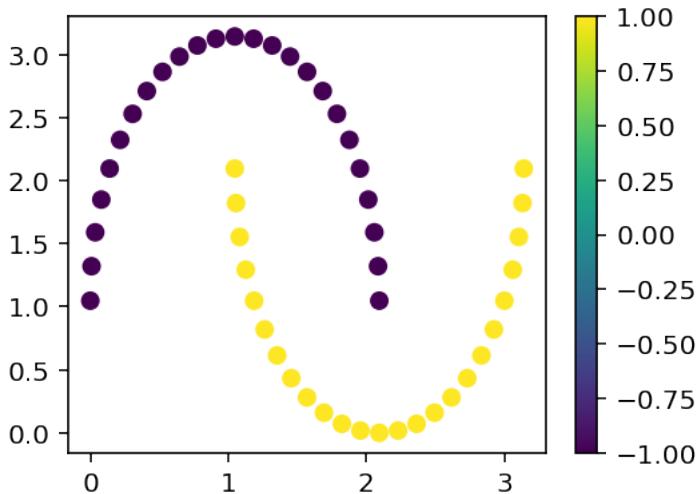
```

def normalize(x):
    """
    Переводит значения в интервал от 0 до pi
    """
    min_ = x.min()
    max_ = x.max()
    return np.pi * (x - min_) / (max_ - min_)

x[:, 0] = normalize(x[:, 0])
x[:, 1] = normalize(x[:, 1])
y = y * 2 - 1

plt.figure(figsize=(4, 3))
cb = plt.scatter(x[:, 0], x[:, 1], c=y)
plt.colorbar(cb)
plt.show()

```



47.4 Вариационная схема

Перед тем как мы начнем обучение и будем считать градиенты, нам необходимо определиться с тем, как будет выглядеть наша вариационная схема. Мы посвятим кодированию данных, выбору архитектуры схемы, а также измеряемого оператора еще много занятий. Так что пока просто воспользуемся кодированием признаков \mathbf{X} вращениями, а сверху применим несколько параметризованных слоев вращений.

47.4.1 Кодирование признаков

```

dev = qml.device("default.qubit", 2)

def encoding(x1, x2):
    qml.RY(x1, wires=0)
    qml.RY(x2, wires=1)
    qml.RZ(x1, wires=0)
    qml.RZ(x2, wires=1)
    qml.CZ(wires=[0, 1])

```

47.4.2 Параметризованные слои

В качестве одного слоя обучения мы будем использовать параметризованные вращения в связке с двухкубитным гейтом для создания запутанных состояний.

Note: Более детально о запутанных состояниях, а также квантовой энтропии и черных дырах можно посмотреть в [продвинутой лекции](#) блока про квантовые вычисления.

В этой лекции у нас нет цели идеально решить поставленную задачу – на самом деле это чуть сложнее, чем может показаться на первый взгляд. Поэтому пока не будем излишне усложнять нашу **VQC**. Сделаем нашу **VQC** содержащей несколько “слоев” следующего вида:

- Вращение 1-го кубита $\hat{R}ot(\theta_1^1, \theta_1^2, \theta_1^3)$
- Вращение 2-го кубита $\hat{R}ot(\theta_2^1, \theta_2^2, \theta_2^3)$
- “Запутывающий” оператор, который действует на оба кубита сразу – в нашем случае это \hat{CZ}

Как видно, на каждый “слой” у нас приходится шесть параметров. Реализуем это в коде:

```
def layer(theta):
    qml.Rot(theta[0, 0], theta[0, 1], theta[0, 2], wires=0)
    qml.Rot(theta[1, 0], theta[1, 1], theta[1, 2], wires=1)
    qml.CZ(wires=[0, 1])
```

Здесь у нас вращения каждого из кубитов по сфере Блоха и двухкубитное взаимодействие \hat{CZ} .

47.4.3 Все вместе

Теперь давайте объединим все это вместе, добавим пару наблюдаемых и оформим как `qml.qnode`:

```
@qml.qnode(dev)
def node(x1, x2, q):
    encoding(x1, x2)
    for q_ in q:
        layer(q_)

    return qml.expval(qml.PauliZ(0) @ qml.PauliY(1))
```

47.4.4 Функция “скоринга”

Наша квантовая схема принимает на вход лишь одну точку данных, а у нас их 50. Поэтому удобно сразу написать функцию, которая может работать с массивами NumPy:

```
def apply_node(x, q):
    res = []

    for x_ in x:
        vqc_output = node(x_[0], x_[1], q[0])
        res.append(vqc_output + q[1])

    return res
```

Может показаться немного запутанно, но так получилось. Дело в том, что параметры схемы это только углы поворотов. Но мы также хотим добавить еще и смещение, поэтому `tuple` параметров у нас содержит два элемента: массив параметров схемы, а также значение смещения. Так как схема у нас принимает на вход лишь одну пару значений x_1, x_2 , то для того, чтобы “проскорить” массив данных мы должны:

- итерироваться по строкам двумерного массива
- для каждой строки вычислять результат схемы – это функция от x_1, x_2, θ – массив параметров θ у нас первый элемент `tuple`
- добавлять смещение – это второй элемент `tuple`
- результат добавлять в итоговый массив

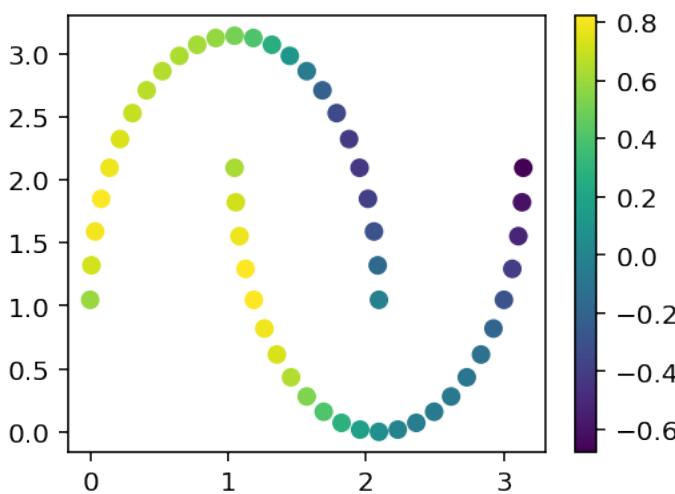
Именно это и реализовано в коде.

47.4.5 Визуализация

Давайте инициализируем нашу схему случайными параметрами и посмотрим, как она “сходу” классифицирует данные. Возьмем 4 параметризованных слоя.

```
np.random.seed(42)
q = (np.random.uniform(-np.pi, np.pi, size=(4, 2, 3)), 0.0)

plt.figure(figsize=(4, 3))
cb = plt.scatter(x[:, 0], x[:, 1], c=apply_node(x, q))
plt.colorbar(cb)
plt.show()
```



Как видно, результат “не очень” и наша цель – попытаться его улучшить.

47.5 Функция потерь

Прежде чем варировать параметры схемы, нам для начала необходимо понять, а что именно мы хотим оптимизировать. Для этого нам необходимо выбрать функцию потерь.

Note: Если у Вас трудности с функциями потерь в таком контексте, то рекомендуем вернуться к вводной лекции про классическое машинное обучение, где эта тема раскрыта достаточно подробно.

47.5.1 Квадратичное отклонение

В качестве функции потерь, которая является дифференцируемой, мы будем использовать наиболее простой вариант – среднеквадратичное отклонения. Это не самый лучший выбор для задач классификации, но зато самый простой. Простой вариант – это именно то, что нам нужно в этой лекции:

```
def cost(q, x, y):
    preds = np.array(apply_node(x, q))
    return np.mean(np.square(preds - y))
```

47.5.2 Точность классификации

В качестве метрики качества среднеквадратичное отклонение вообще не подходит – понять по этой цифре, хорошо или плохо работает модель почти невозможно! Поэтому для оценки модели в целом мы будем использовать точность:

```
def acc(q, x, y):
    preds = np.sign(apply_node(x, q))
    res = 0
    for p_, y_ in zip(preds, y):
        if np.abs(y_ - p_) <= 1e-2:
            res += 1

    return res / y.shape[0]
```

47.6 Решение средствами PennyLane

Библиотека PennyLane может использовать один из нескольких движков для автоматического дифференцирования:

- NumPy Autograd
- PyTorch
- Tensorflow
- Jax

По большей части, на наших занятиях мы будем использовать NumPy из-за простоты и привычности. Перед тем как разбираться с тем, как же именно происходит дифференцирование квантовой схемы, давайте посмотрим на весь цикл обучения.

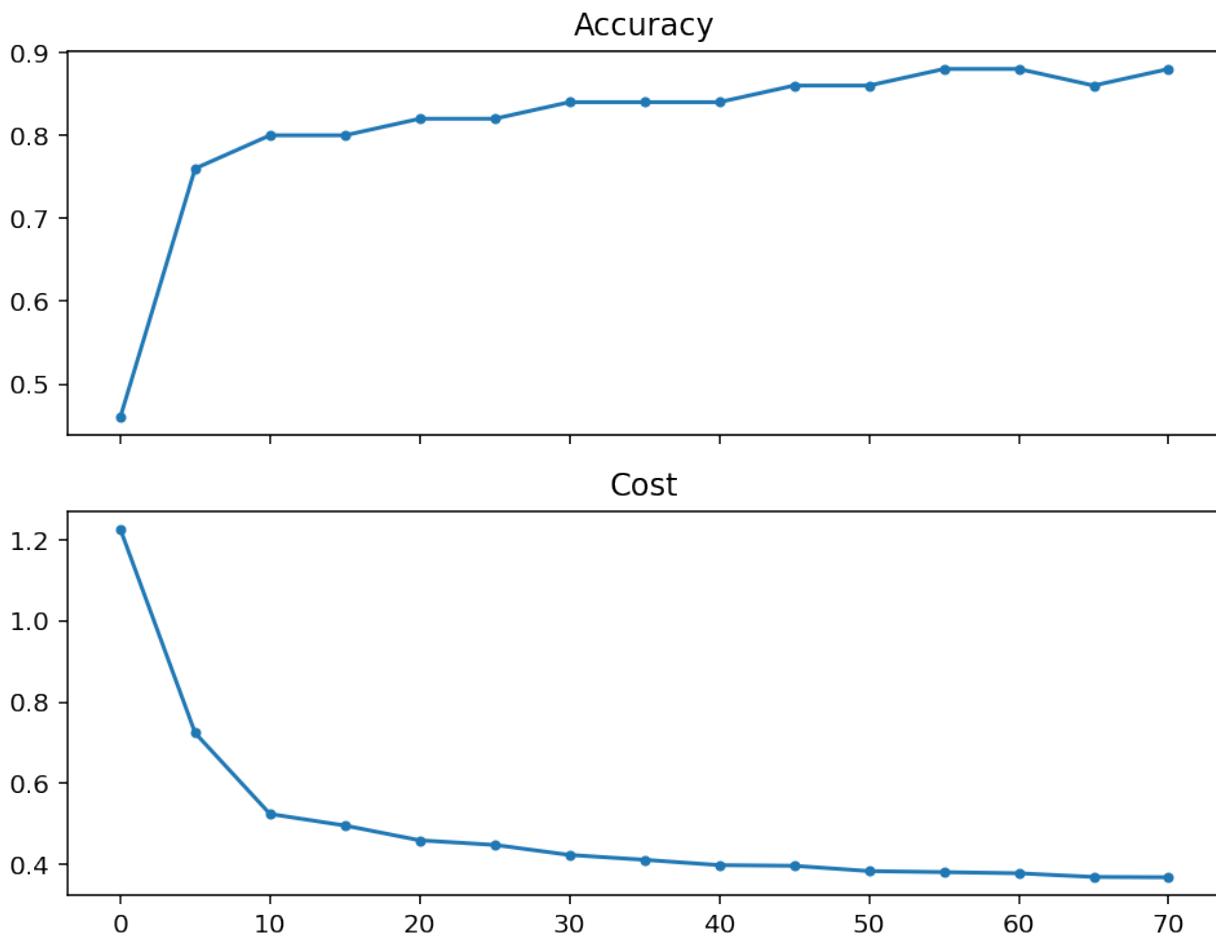
Note: Внимание, процесс обучения на обычном ноутбуке может занять около минуты! Это связано с трудностью симуляции квантового компьютера на классическом.

```
opt = qml.optimize.GradientDescentOptimizer(stepsizes=0.05)
acc_ = []
cost_ = []
ii = []
for i in range(75):
    batch = np.random.randint(0, len(x), (10,))
    x_batch = x[batch, :]
    y_batch = y[batch]
    q = opt.step(lambda q: cost(q, x_batch, y_batch), q)

    if i % 5 == 0:
        ii.append(i)
        acc_.append(acc(q, x, y))
        cost_.append(cost(q, x, y))
```

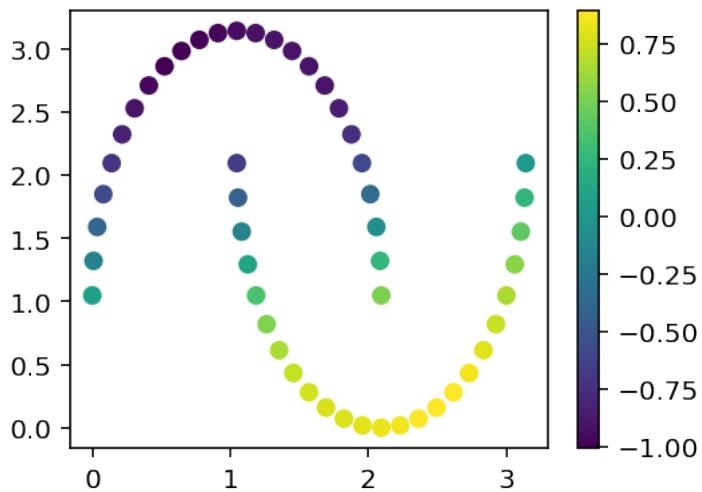
И посмотрим на получившиеся графики точности и функции потерь:

```
f, ax = plt.subplots(2, figsize=(8, 6), sharex=True)
ax[0].plot(ii, acc_, ".-")
ax[0].set_title("Accuracy")
ax[1].plot(ii, cost_, ".-")
ax[1].set_title("Cost")
plt.show()
```



А также на результаты классификации:

```
plt.figure(figsize=(4, 3))
cb = plt.scatter(x[:, 0], x[:, 1], c=apply_node(x, q))
plt.colorbar(cb)
plt.show()
```



47.7 А как оно работает?

Теперь, когда мы увидели процесс оптимизации квантовой схемы, давайте попробуем подумать, а как оно на самом деле работает?

47.7.1 Метод конечных отрезков

Для начала вспомним то, что является геометрическим (или визуальной) интерпретацией градиента функции. Правильно, градиент в каждой точке – это касательная. А приближенное значение угла наклона любой прямой можно найти, взяв конечные отрезки:

$$\frac{df}{dx} \simeq \frac{\Delta f}{\Delta x} = \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Давайте попробуем посчитать градиент нашей квантовой схемы. Для этого инициализируем ее заново случайными параметрами, а потом сравним это с тем значением, которое считает Autograd.

```
np.random.seed(42)
q = (np.random.uniform(0, np.pi, size=(4, 2, 3)), 0.0)

def naive_grad(cost, params, x, y, f0, delta):
    return (cost(params, x, y) - f0) / delta

def grad_i(q, f0, cost, x, y, i):
    new_params = q[0].copy().flatten()
    new_params[i] += 0.05

    return naive_grad(cost, (new_params.reshape(q[0].shape), q[1]), x, y, f0, 0.05)
```

В качестве эталона возьмем тот градиент, который нам считает Autograd:

```
grad =qml.grad(cost, argnum=0)
```

Сравним первые пять значений Autograd с нашим наивным алгоритмом, взяв $\Delta = 0.05$:

Note: Осторожно! Так как мы считаем градиенты очень наивно и на всех точках сразу, то следующий блок кода работает долго!

```
autograd = grad(q, x, y)
f0 = cost(q, x, y)
pretty_print = ""
for i in range(10):
    g_i = grad_i(q, f0, cost, x, y, i)
    pretty_print += f"Naive grad: {g_i:.3f}\tAutograd result: {autograd[0].
    flatten()[i]:.3f}\n"
print(pretty_print)
```

Naive grad: -0.028	Autograd result: -0.029
Naive grad: 0.081	Autograd result: 0.081
Naive grad: 0.025	Autograd result: 0.025
Naive grad: 0.013	Autograd result: 0.008
Naive grad: -0.000	Autograd result: -0.011
Naive grad: 0.009	Autograd result: 0.004
Naive grad: 0.025	Autograd result: 0.025
Naive grad: 0.002	Autograd result: -0.003
Naive grad: -0.068	Autograd result: -0.069
Naive grad: 0.009	Autograd result: 0.004

Можно заметить, что даже с таким большим значением Δ наши оценки получились достаточно близкими к тем, которые получены в Autograd. Хотя, конечно, для некоторых значений расхождения заметны и иногда они даже в знаке частной производной.

47.7.2 Parameter-shift rule

Более точная оценка может быть получена методом, который называется **Parameter-shift**. Он основан на том, что для квантового “черного ящика” $\hat{U}(\theta)$, которым является наша схема, частная производная по параметру θ_i выражается так:

$$\nabla_{\theta_i} \langle \hat{U} \rangle = \frac{1}{2} [\langle \hat{U} \rangle \left(\theta + \frac{\pi}{2} \mathbf{e}_i \right) - \langle \hat{U} \rangle \left(\theta - \frac{\pi}{2} \mathbf{e}_i \right)]$$

Note: Более строгую формулировку, а также вывод правила parameter-shift можно посмотреть в продвинутой лекции этого блока про производные высших порядков.

Если по-простому, то оценка частной производной по i -му параметру может быть получена вычислением сначала ожидаемого значения схемы с параметром θ_i , смещенным на $\frac{\pi}{2}$ в одну сторону, а потом – в другую. Давайте запишем это в коде, но перед этим давайте вспомним, как будет выглядеть производная функции потерь (а именно она нам нужна):

$$\frac{\partial L}{\partial \theta_i} = \frac{\partial (\langle \hat{U} \rangle(\theta) - y)^2}{\partial \theta_i} = 2(\langle \hat{U} \rangle - y) \frac{\partial \langle \hat{U} \rangle}{\partial \theta_i}$$

Реализуем явно и наивно эту формулу в коде:

```

def parameter_shift_i(q, cost, x, y, i, y_hat):
    new_params = q[0].copy().flatten()
    new_params[i] += np.pi / 2

    forward = np.array(apply_node(x, (new_params.reshape(q[0].shape), q[1])))

    new_params = q[0].copy().flatten()
    new_params[i] -= np.pi / 2

    backward = np.array(apply_node(x, (new_params.reshape(q[0].shape), q[1])))

    diff = (y_hat - y)

    return np.mean(2 * diff * (0.5 * (forward - backward)))

```

И также проверим на первых 10 точках:

```

y_hat = apply_node(x, q)
pretty_print = ""
for i in range(10):
    g_i = parameter_shift_i(q, cost, x, y, i, y_hat)
    pretty_print += f"Naive grad: {g_i:.3f}\tAutograd result: {autograd[0].
    flatten()[i]:.3f}\n"

print(pretty_print)

```

Naive grad: -0.029	Autograd result: -0.029
Naive grad: 0.081	Autograd result: 0.081
Naive grad: 0.025	Autograd result: 0.025
Naive grad: 0.008	Autograd result: 0.008
Naive grad: -0.011	Autograd result: -0.011
Naive grad: 0.004	Autograd result: 0.004
Naive grad: 0.025	Autograd result: 0.025
Naive grad: -0.003	Autograd result: -0.003
Naive grad: -0.069	Autograd result: -0.069
Naive grad: 0.004	Autograd result: 0.004

Как видно, этот результат уже совпадает с тем, что делает “под капотом” PennyLane и Autograd. На самом деле, правило **parameter-shift** позволяет использовать много интересных хитростей и оптимизаций, но их не получится легко показать без погружения в математические детали метода.

47.8 Что мы узнали?

- Мы попробовали провести полный цикл оптимизации параметров VQC
- Научились использовать автоматический расчет градиентов в PennyLane
- Познакомились с двумя способами оценки градиента:
 - Метод конечных отрезков
 - Parameter-shift rule

ГРАДИЕНТЫ ВЫСШИХ ПОРЯДКОВ

48.1 План лекции

В этой лекции мы посмотрим на ту математику, которая лежит “под капотом” у *parameter-shift rule*. Мы познакомимся с обобщением *parameter shift*, а также увидим, как можно оптимизировать этот метод. В конце мы узнаем, как можно посчитать производную второго порядка за минимальное количество обращений к квантовому компьютеру.

Для более детального погружения в вопрос можно сразу рекомендовать статью [MBK21].

48.2 Важность гейтов вращений

Если задуматься, то одним из основных (если не единственных) способов сделать параметризованную квантовую схему является использование гейтов вращений, таких как $\hat{R}X$, $\hat{R}Y$, $\hat{R}Z$. Более формально это можно выразить так, что нас больше всего интересуют операторы вида:

$$U(\theta) = e^{-\frac{i}{2}H\theta}$$

где H – оператор “вращения”, который удовлетворяет условию $H^2 = \mathbf{1}$. Другой возможный вариант записи – представить матрицу H как линейную комбинацию операторов Паули σ^x , σ^y , σ^z .

Если представить схему, содержащую множество параметризованных операторов, то итоговая запись имеет вид:

$$U_{j\dots k} = U_j, \dots, U_k \Psi$$

48.3 Производная от измерения

Давайте вспомним, как выглядит квантово-классическая схема обучения с VQC.

Видно, что мы хотим считать производную не от самой параметризованной схемы $U_{j\dots k}$, а от наблюдаемой. Для тех, кто забыл, что такое *наблюданная*, рекомендуем вернуться к [лекции про кубит](#). Если кратко, то это тот оператор, который мы “измеряем” на нашем квантовом компьютере. Математически производная, которая нам интересна, может быть записана для выбранного параметра i таким образом:

$$G_i = \frac{\partial U_{j\dots k} \Psi \hat{M} U_{j\dots k} \Psi}{\partial \theta_i}$$

То есть нам важно посчитать производную от результата измерения, так как именно результат измерения у нас будет определять “предсказание” нашей квантовой нейронной сети. Причем нам нужно уметь считать производную от любого параметра θ_i в цепочке $\theta_j, \dots, \theta_i, \dots, \theta_k$.

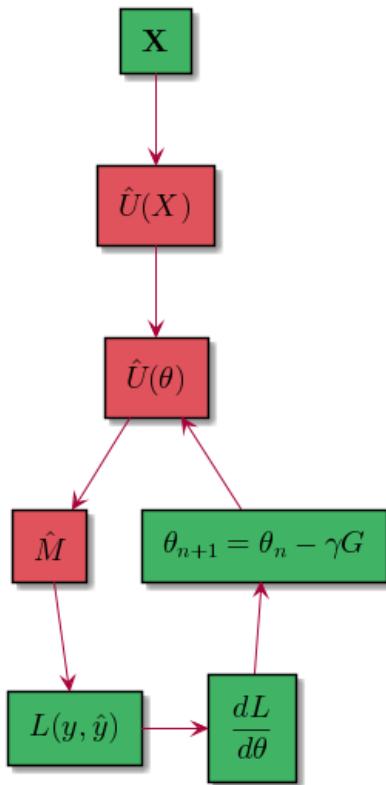


Fig. 48.1: Квантово-классическая схема

48.4 Parameter-shift для гейтов Паули

Note: Тут мы для простоты предложим, что U_1 это просто оператор вращения, иначе выкладки станут совсем сложными.

Тогда сам оператор U_i может быть также записан так:

$$U_i = e^{-\frac{i}{2}P_i\theta_i}$$

Запишем результат математического ожидания через состояние Ψ_i , которое пришло на вход i -го гейта в нашей последовательности:

$$\langle M(\theta) \rangle = \text{Tr}(MU_{k,\dots,1}\rho_i U_{k,\dots,1}^\dagger)$$

где ρ это матрица плотности ($\Psi\Psi$). Подробнее о матрицах плотности можно почитать в ранней продвинутой лекции про смешанные состояния.

Тогда частная производная от математического ожидания по i -му параметру θ_i записывается (подробнее в [MNKF18]) через коммутатор исходного состояния ρ , которое “пришло” на вход гейта U_i и того оператора Паули P_i , который мы используем в U_i :

$$\frac{\partial \langle M \rangle}{\partial \theta_i} = -\frac{i}{2} \text{Tr}(MU_{k,\dots,i}[P_i, U_{i-1,\dots,1}\rho_i U_{i-1,\dots,1}^\dagger]U_{k,\dots,i}^\dagger)$$

Этот коммутатор может быть переписан следующим образом:

$$[P_i, \rho] = i[U_i\left(\frac{\pi}{2}\right)\rho_i U_i^\dagger\left(\frac{\pi}{2}\right) - U_i\left(-\frac{\pi}{2}\right)\rho_i U_i^\dagger\left(-\frac{\pi}{2}\right)]$$

Тогда соответствующий градиент $\frac{\partial}{\partial \theta_i}$ можно записать через смещения на $\pm \frac{\pi}{2}$:

$$\begin{aligned} \frac{\partial \langle M \rangle}{\partial \theta_i} &= \frac{\langle M_i^+ \rangle - \langle M_i^- \rangle}{2} \\ \langle M_i^\pm \rangle &= \frac{1}{2} \text{Tr}[MU_{k,\dots,i+1}U_i(\pm\frac{\pi}{2})\rho_i^{\overset{\circ}{U}_i^\dagger(\pm\frac{\pi}{2})U_{k,\dots,i+1}^\dagger}] \\ &\quad \rho_i^{\overset{\circ}{U}_i} = U_{j,\dots,1}\rho_i U_{j,\dots,1}^\dagger \end{aligned}$$

По аналогии с классическими нейронными сетями и *backpropagation* (для тех, кто забыл это понятие, рекомендуем вернуться к вводным лекциям про классическое машинное обучение) тут явно можно выделить *forward* проход со смещением θ_i на значения $\frac{\pi}{2}$ и *backward* со смещением на $-\frac{\pi}{2}$.

48.5 Обобщенный parameter-shift

Предложенное в [MNKF18] выражение может быть на самом деле получено в более общем виде из других соображений. Так, выражение для нашей наблюдаемой $\langle M \rangle$ может всегда быть представлено [MBK21] как сумма вида:

$$U_i(\theta_i)\hat{M}U_i(\theta_i) = \hat{A} + \hat{B} \cos \theta_i + \hat{C} \sin \theta_i$$

где $\hat{A}, \hat{B}, \hat{C}$ – операторы, не зависящие от параметра θ_i .

Note: Действительно, явно выписав выражение для наблюдаемой и вспомнив формулы для косинуса и синуса двойного угла, а также воспользовавшись тем, что $U(\theta) = e^{-\frac{1}{2}H\theta} = \cos \frac{\theta}{2}\mathbf{1} - i \sin \frac{\theta}{2}H$, получаем:

$$\begin{aligned} & (\cos \frac{\theta}{2}\mathbf{1} + i \sin \frac{\theta}{2}H)\hat{M}(\cos \frac{\theta}{2}\mathbf{1} - i \sin \frac{\theta}{2}H) = \\ & \cos^2 \frac{\theta}{2}\hat{M} + i \sin \frac{\theta}{2} \cos \frac{\theta}{2}H\hat{M} - i \sin \frac{\theta}{2} \cos \frac{\theta}{2}\hat{M}H + \sin^2 \frac{\theta}{2}H\hat{M}H = \\ & \frac{1}{2}\cos \theta \hat{M} + \frac{1}{2}\hat{M} + \frac{i}{2}\sin \theta H\hat{M} - \frac{i}{2}\sin \theta \hat{M}H + \frac{1}{2}H\hat{M}H - \frac{1}{2}\cos \theta H\hat{M}H = \\ & \frac{1}{2}(\hat{M} + H\hat{M}H) + \frac{1}{2}(\hat{M} - H\hat{M}H)\cos \theta + \frac{i}{2}(H\hat{M} - \hat{M}H)\sin \theta \end{aligned}$$

Тогда можно воспользоваться правилами тригонометрии, а именно, тем что для любого $s \neq k\pi$, $k \in 1, 2, \dots$, справедливо:

$$\begin{aligned} \frac{d \cos \theta}{d\theta} &= \frac{\cos(\theta + s) - \cos(\theta - s)}{2 \sin s} \\ \frac{d \sin \theta}{d\theta} &= \frac{\sin(\theta + s) - \sin(\theta - s)}{2 \sin s} \end{aligned}$$

И подставим это в выражение для $\frac{\partial \langle M \rangle}{\partial \theta_i}$:

$$\frac{\partial \langle M(\theta_i) \rangle}{\partial \theta_i} = \frac{\langle M(\theta_i + s) \rangle - \langle M(\theta_i - s) \rangle}{2 \sin s}$$

Легко заметить, что подстановка сюда $s = \frac{\pi}{2}$ дает нам классический *parameter shift*, описанный в [MNKF18].

Наконец, запишем полученное выражение в более удобном виде, который позволит нам более эффективно выписывать производные высших порядков. Для этого введем вектор \mathbf{e}_i – единичный вектор для i -го параметра, то есть вектор, где все компоненты кроме i -й равны нулю, а i -я равна 1. Тогда наше финальное выражение для обобщенного *parameter shift* примет следующий вид:

$$\boxed{\frac{\partial f(\mathbf{\Theta})}{\partial \theta_i} = \frac{f(\mathbf{\Theta} + s\mathbf{e}_i) - f(\mathbf{\Theta} - s\mathbf{e}_i)}{2 \sin s}}$$

48.6 Вторая производная и гессиан

В классической теории оптимизации, также как и в машинном обучении, очень часто на первый план выходят так называемые методы 2-го порядка. Эти методы похожи на обычный градиентный спуск, но для ускорения сходимости они также используют информацию из матрицы вторых производных, которая называется гессианом. Более подробно про методы 2-го порядка и гессиан можно посмотреть в вводных лекциях курса.

Методы второго порядка требуют больше вызовов, чтобы вычислить гессиан, но взамен они обеспечивают гораздо лучшую сходимость, а также менее склонны “застревать” в локальных минимумах. Это обеспечивает, в итоге, более быстрое обучение. В классических нейронных сетях вычисление гессиана это часто проблема, так как это матрица размерности $\sim O(N^2)$, где N – число весов нейронной сети, и эта матрица получается слишком большой. Но, как мы помним, основная “фича” **VQC** это их экспоненциальная экспрессивность – возможность линейным числом параметров (и гейтов) обеспечить преобразование, эквивалентное экспоненциальному числу весов классической нейронной сети. А значит, для них проблема размерности гессиана не стоит так остро. При этом использование гессиана теоретически позволяет в итоге обучить **VQC** за меньшее число вызовов. Именно поэтому методы второго порядка потенциально очень интересны в квантово-классическом обучении. Но для начала нам необходимо разобраться, как именно можно посчитать матрицу вторых производных.

Пользуясь обобщенным правилом *parameter shift*, можно выписать выражение для второй производной [MBK21]:

$$\frac{\partial_2 f}{\partial \theta_i \theta_j} = \frac{f(\underline{\theta} + s_1 \mathbf{e}_i + s_2 \mathbf{e}_j) + f(\underline{\theta} - s_1 \mathbf{e}_i - s_2 \mathbf{e}_j) - f(\underline{\theta} + s_1 \mathbf{e}_i - s_2 \mathbf{e}_j) - f(\underline{\theta} - s_1 \mathbf{e}_i + s_2 \mathbf{e}_j)}{4 \sin s_1 \sin s_2}$$

Взяв $s_1 = s_2$, можно упростить это выражение к следующему виду:

$$\frac{f(\underline{\theta} + s\mathbf{a}) + f(\underline{\theta} + s\mathbf{b}) - f(\underline{\theta} + s\mathbf{c}) - f(\underline{\theta} + s\mathbf{d})}{(2 \sin s)^2}$$

$$\mathbf{a} = \mathbf{e}_i + \mathbf{e}_j$$

$$\mathbf{b} = -\mathbf{e}_i - \mathbf{e}_j$$

$$\mathbf{c} = \mathbf{e}_i - \mathbf{e}_j$$

$$\mathbf{d} = -\mathbf{e}_i + \mathbf{e}_j$$

Но чаще всего нам необходимо не просто посчитать гессиан, а еще и посчитать градиент, так как в большинстве методов 2-го порядка требуются оба эти значения. В этом случае хочется попробовать подобрать такое значение для s_g при вычислении вектора градиента, а также такое значение s_h при вычислении гессиана, чтобы максимально переиспользовать результаты квантовых вызовов и уменьшить их общее количество.

Внимательно взглянув на выражение для 2-х производных, можно заметить, что оптимизация там возможна при расчете диагональных элементов гессиана. Давайте выпишем выражение для диагонального элемента явно:

$$\frac{f(\underline{\theta} + 2s\mathbf{e}_i) + f(\underline{\theta} - 2s\mathbf{e}_i) - 2f(\underline{\theta})}{(2 \sin s)^2}$$

Можно заметить, что, например, использование $s = \frac{\pi}{4}$ для гессиана, а также “стандартного” $s = \frac{\pi}{2}$ для градиента позволит полностью переиспользовать в диагональных элементах гессиана значения, которые мы получили при расчете градиента. А значение $f(\underline{\theta})$ вообще считается один раз для всех диагональных вызовов.

Note: На самом деле, диагональные элементы гессиана можно использовать и сами по себе, например для квазиньютоновских методов оптимизации, где матрица Гессе аппроксимируется какой-то другой матрицей, чтобы не считать все вторые производные. Например, она может быть аппроксимирована диагональной матрицей, как в работе [And19].

48.7 Заключение

В этой лекции мы познакомились с классическим *parameter shift rule*, а также его обобщением. Также мы узнали, как можно посчитать гессиан VQC, и даже узнали маленькие хитрости, которые можно применять для уменьшения общего количества вызовов квантовой схемы.

CHAPTER
FORTYNINE

О БЛОКЕ “КВАНТОВЫЙ SVM”

Этот блок включает в себя только квантово-классический SVM.

Продвинутая тема блока дополнительно рассказывает о классическом SVM.

КВАНТОВО-КЛАССИЧЕСКИЙ SVM

50.1 Описание лекции

Лекция будет построена следующим образом:

- посмотрим, как можно использовать VQC как ядро SVM;
- напишем и применим код обучения смешанного SVM.

Перед тем как приступить, рекомендуется освежить [материал](#) по классическому методу опорных векторов, в частности, нужно понимать, как работает ядерный трюк.

50.2 VQC как ядерная функция

Как мы много говорили в более ранних лекциях, квантовые схемы позволяют нам оперировать в гильбертовых пространствах волновых функций. Эти пространства имеют экспоненциально большую размерность, при этом они параметризуются линейным количеством параметров. А еще в этих пространствах определены скалярные произведения волновых функций, более того, именно результат выборки из скалярного произведения мы чаще всего и получаем как результат измерений!

Давайте попробуем посмотреть, что общего у VQC и ядер.

- оба оперируют в пространстве большой (или бесконечной размерности);
- и там, и там работа идет в гильбертовом пространстве и определено скалярное произведение;
- и там, и там результат вычисляется как скалярное произведение.

Note: Это интересно, что многие специалисты в области QML сегодня даже предлагают вместо термина “квантовая нейросеть” использовать термин “квантовое ядро”, так как математически VQC гораздо ближе именно к ядрам, чем к слоям современных глубоких сетей.

Очевидная идея – попробовать как-то воспользоваться квантовой схемой, чтобы реализовать скалярное произведение двух классических векторов. Именно это и сделали авторы работы [[HavlíčekCorcolesT+19](#)].

50.2.1 Преобразование состояния

На самом деле, если просто использовать какие-то простые квантовые операции, мы не получим какого-то преимущества над классическим ядерным SVM – ведь все то же самое можно будет сделать и на классическом компьютере.

Чтобы получить реальное преимущество, нам необходимо использовать запутывание и прочие “фишки” квантовых вычислений.

Дальше мы не станем изобретать велосипеды, а вместо этого воспользуемся примерами хороших преобразований из работы [SYG+20]. Рассмотрим, что именно там описано.

Общая схема

Для простоты формул мы не будем выписывать обобщенные формулы, а все будем писать для нашего двумерного пространства. Тогда наша схема может быть разделена на несколько частей:

- гейты Адамара и гейты $CNOT$;
- операции, основанные на элементах входного вектора;
- попарные операции над парами элементов вектора.

Мы начинаем с того, что переводим кубиты в состояние суперпозиции, применяя операторы Адамара. Далее мы применяем однокубитные параметризованные операции и снова гейты Адамара. После этого мы применяем связку $CNOT \rightarrow$ параметризованная парой операция $\rightarrow CNOT$.

Выбор операции

Следуя идее упомянутой статьи, в качестве что одно-элементной, что двух-элементной операции мы будем использовать гейт U_1 . Разница будет лишь в том, что мы передаем на вход в качестве параметра.

Feature function

В качестве параметров на входе гейта U_1 , как мы уже говорили, выступают один или два элемента вектора x . Строго это можно записать как функцию такого вида:

$$\phi(x_1, x_2) = \begin{cases} \phi(x), & x_1 = x_2 \\ \phi(x_1, x_2), & x_1 \neq x_2 \end{cases}$$

Мы будем называть ее *feature function*. В некотором смысле можно сказать, что именно эта функция определяет тип ядра по аналогии с классическим SVM. В работе [SYG+20] описано много разных вариантов таких *feature function*, мы будем использовать следующую:

$$\begin{aligned} \phi(x) &= x \\ \phi(x_1, x_2) &= \pi \cos x_1 \cos x_2 \end{aligned}$$

Скалярное произведение

Все что мы описали выше, обозначим как квантовую схему $U(x)$. Она преобразует нам вектор классических данных x в квантовое состояние Ψ . Но нам то нужно получить скалярное произведение $U(x_1)|U(x_2)!$ Выглядит сложно, но на самом деле существует эффективный способ получить эту величину без необходимости восстанавливать весь вектор состояния. Можно показать, что величина $U(x_1)|U(x_2)$ равна вероятности нулевой битовой строки $(0, 0, \dots, 0)$ при измерении другой схемы: $U(x_1)U(x_2)^\dagger$.

Все это может казаться сложным и запутанным, но должно стать гораздо понятнее, когда мы посмотрим на пример реализации от начала и до конца.

50.3 Пример реализации

50.3.1 Схема

Для начала необходимые импорты.

```
from pennylane import numpy as np
import pennylane as qml
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
from sklearn.datasets import make_moons
```

Помимо всех привычных, нам еще потребуется классический SVM из scikit-learn:

```
from sklearn.svm import SVC
```

Мы будем работать с уже привычным нам набором “Tow Moons”. Только в этом случае мы будем использовать чуть-чуть другую нормализацию – для нашего ядра элементы вектора x должны быть в интервале $[-1, 1]$. Сразу переведем наши данные в этот диапазон:

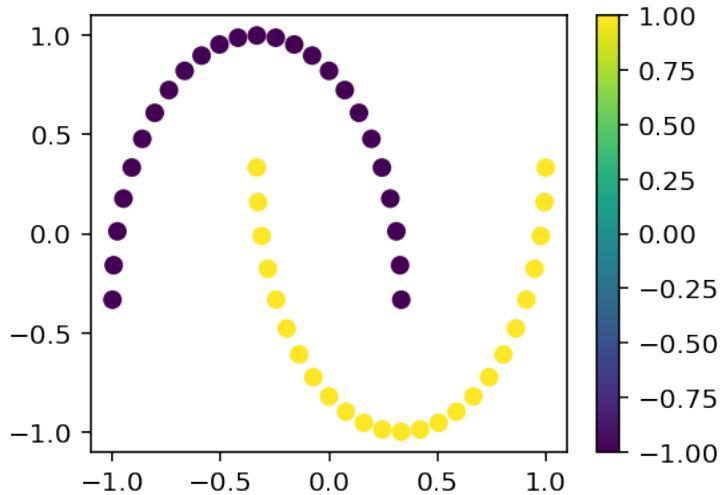
```
x, y = make_moons(n_samples=50)
y = y * 2 - 1

def normalize(x):
    """Переводит значения в интервал от -1 до 1"""

    min_ = x.min()
    max_ = x.max()
    return 2 * (x - min_) / (max_ - min_) - 1

x[:, 0] = normalize(x[:, 0])
x[:, 1] = normalize(x[:, 1])

plt.figure(figsize=(4, 3))
cb = plt.scatter(x[:, 0], x[:, 1], c=y)
plt.colorbar(cb)
plt.show()
```



И привычное нам объявление устройства.

```
dev = qml.device("default.qubit", 2)
```

Теперь давайте для начала реализуем наше преобразование над одним из векторов ($U(x)$). Поскольку далее нам потребуется еще и $U(x)^\dagger$, то мы сразу воспользуемся декоратором `@qml.template`, который позволит нам автоматически получить обратную схему.

```
@qml.template
def var_layer(x):
    qml.Hadamard(wires=0)
    qml.Hadamard(wires=1)

    qml.U1(x[0], wires=0)
    qml.U1(x[1], wires=1)

    qml.Hadamard(wires=0)
    qml.Hadamard(wires=1)

    qml.CNOT(wires=[0, 1])
    qml.U1(np.pi * np.cos(x[0]) * np.cos(x[1]), wires=1)
    qml.CNOT(wires=[0, 1])
```

А теперь реализуем

$$U(x_1)|U(x_2) = U(x_1)U(x_2)^\dagger \Psi M_0 U(x_1)U(x_2)^\dagger \Psi,$$

где M_0 – проектор на один из собственных векторов системы кубитов, а именно на “нулевой”:

$$M_0 = 0, \dots, 00, \dots, 0.$$

Проще говоря, мы реализуем схему, которая нам дает вероятности каждой из битовых строк (а дальше мы просто возьмем первую, она и отвечает строке 0, ...0):

```
@qml.qnode(dev)
def dot_prod(x1, x2):
    var_layer(x1)
    qml.inv(var_layer(x2))

    return qml.probs(wires=[0, 1])
```

Ну и сразу вспомогательную функцию, которая нам считает то, что нам было нужно:

```
def q_dot_prod(i, j):
    x1 = (x[i, 0], x[i, 1])
    x2 = (x[j, 0], x[j, 1])
    return dot_prod(x1, x2)[0]
```

Для самопроверки убедимся в том, что наше “скалярное произведение” симметрично:

```
print(np.allclose(q_dot_prod(0, 1), q_dot_prod(1, 0)))
```

```
True
```

```
/home/runner/work/qmlcourse/qmlcourse/.venv/lib/python3.8/site-packages/pennylane/
  ↪utils.py:371: UserWarning: Use of qml.inv() is deprecated and should be replaced
  ↪with qml.adjoint().
  warnings.warn(
```

И сразу посмотрим на то, как выглядит наша схема:

```
print(dot_prod.draw())
```

```
0: —H—Rφ(0.333)—H—rC—————rC—rC—————rC—H—Rφ(-0.246)—H—r|
  ↪Probs
1: —H—Rφ(-0.333)—H—lX—Rφ(2.81)—lX—lX—Rφ(-1.68)—lX—H—Rφ(0.989)—H—l|
  ↪Probs
```

50.3.2 Гибридный SVM

Мы не будем сами с нуля писать решение задачи квадратичного программирования, просто воспользуемся готовой рутиной из `scikit-learn`. Используемая там реализация позволяет вместо ядерной функции передать сразу матрицу Грамма (Gram matrix). На самом деле это просто матрица всех попарных скалярных произведений наших векторов. Вычислим ее, сразу воспользовавшись тем, что

$$U(x)|U(x) = 1 \quad \text{и} \quad U(x_1)|U(x_2) = U(x_2)|U(x_1)$$

```
gram_mat = np.zeros((x.shape[0], x.shape[0]))

for i in range(x.shape[0]):
    for j in range(x.shape[0]):
        if i == j:
            gram_mat[i, j] = 1
        if i > j:
            r = q_dot_prod(i, j)
            gram_mat[i, j] = r
            gram_mat[j, i] = r
```

Обучим нашу модель:

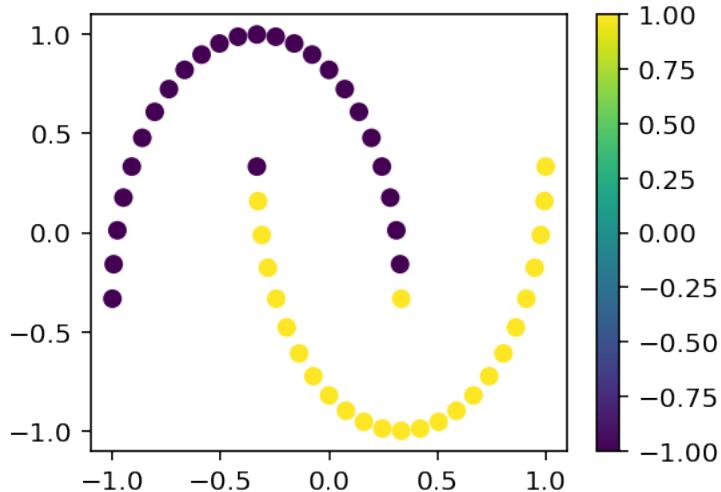
```
model = SVC(kernel="precomputed")
model.fit(gram_mat, y)
```

```
SVC(kernel='precomputed')
```

Посчитаем предсказания и посмотрим на результат:

```
preds = model.predict(X=gram_mat)

plt.figure(figsize=(4, 3))
cb = plt.scatter(x[:, 0], x[:, 1], c=preds)
plt.colorbar(cb)
plt.show()
```



Результат выглядит неплохо!

50.4 Заключение

Расчет полной матрицы скалярных произведений дает нам сложность $O(N^2)$ вызовов. Но основной потенциал гибридного SVM в том, что задачу квадратичной оптимизации на самом деле можно тоже решать на квантовом компьютере, используя алгоритм Гровера (про него рассказано в ранних лекциях), причем за сложность всего $O(N)$ и без расчета полной матрицы Грама!

Многие считают, что NISQ квантовые компьютеры могут стать для SVM чем-то типа видеокарт для нейронных сетей и вернуть этот алгоритм на пьедестал лучших алгоритмов машинного обучения!

КЛАССИЧЕСКИЙ SVM

51.1 Описание лекции

В этой лекции мы рассмотрим классический метод опорных векторов (SVM) и разберем стоящую за ним математику. С согласия Евгения Соколова, мы во многом переиспользуем конспекты лекций курса “Машинное обучение”, читаемого на ФКН ВШЭ.

План лекции такой:

- линейная классификация;
- интуиция метода опорных векторов;
- метод опорных векторов для линейно-разделимой выборки;
- метод опорных векторов для линейно-неразделимой выборки;
- решение задачи метода опорных векторов;
- ядерный переход;
- плюсы и минусы SVM.

51.2 Линейная классификация

Основная идея линейного классификатора заключается в том, что признаковое пространство может быть разделено гиперплоскостью на два полупространства, в каждом из которых прогнозируется одно из двух значений целевого класса. Если это можно сделать без ошибок, то обучающая выборка называется *линейно разделимой*.

Рассмотрим задачу бинарной классификации, в которой $\mathbb{X} = \mathbb{R}^d$ – пространство объектов, $Y = \{-1, +1\}$ – множество допустимых ответов (целевой признак), $X = \{(x_i, y_i)\}_{i=1}^\ell$ – обучающая выборка. Будем класс $+1$ называть положительным, а класс -1 – отрицательным. Здесь d – размерность признакового пространства, ℓ – количество примеров в обучающей выборке.

Линейная модель классификации определяется следующим образом:

$$a(x) = \text{sign}(\langle w, x \rangle + b) = \text{sign}\left(\sum_{j=1}^d w_j x_j + b\right),$$

где $w \in \mathbb{R}^d$ – вектор весов, $b \in \mathbb{R}$ – сдвиг (bias), $\text{sign}(\cdot)$ – функция “сигнум”, возвращающая знак своего аргумента, $a(x)$ – ответ классификатора на примере x .

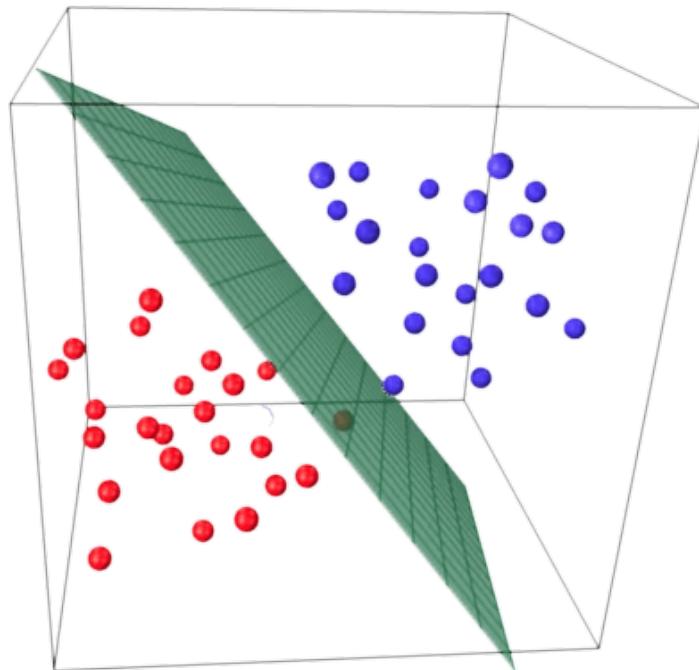


Fig. 51.1: Линейная классификация

Note: Хорошо бы для векторов ставить стрелку и писать \vec{w} , но мы этого не будем делать, предполагая, что из контекста ясно, что вектор, а что скаляр. В частности, в формуле выше w и x – вектора, а $a(x)$, w_j , x_j , d и b – скаляры.

Часто считают, что среди признаков есть константа, $x_{d+1} = 1$. В этом случае нет необходимости вводить сдвиг b , и линейный классификатор можно задавать как

$$a(x) = \text{sign}\langle w, x \rangle.$$

Геометрически линейный классификатор соответствует гиперплоскости с вектором нормали w , которая задается уравнением $\langle w, x \rangle = 0$. Величина скалярного произведения $\langle w, x \rangle$ пропорциональна расстоянию от гиперплоскости до точки x , а его знак показывает, с какой стороны от гиперплоскости находится данная точка. Если быть точным, расстояние от точки с радиус-вектором x_A до плоскости $\langle w, x \rangle = 0$:

$$\rho(x_A, \langle w, x \rangle = 0) = \frac{\langle w, x_A \rangle}{\|w\|}.$$

Упражнение на метод Лагранжа

Метод Лагранжа – очень важный метод оптимизации функций при наличии ограничений. Этот же метод вовсю используется ниже в методе опорных векторов.

Рекомендуется ознакомиться с методом Лагранжа и решить (ручками!) простую оптимизационную задачу вида

$$\begin{cases} x + y \rightarrow \min_{x,y} \\ x^2 + y^2 = 1 \end{cases} \quad (51.1)$$

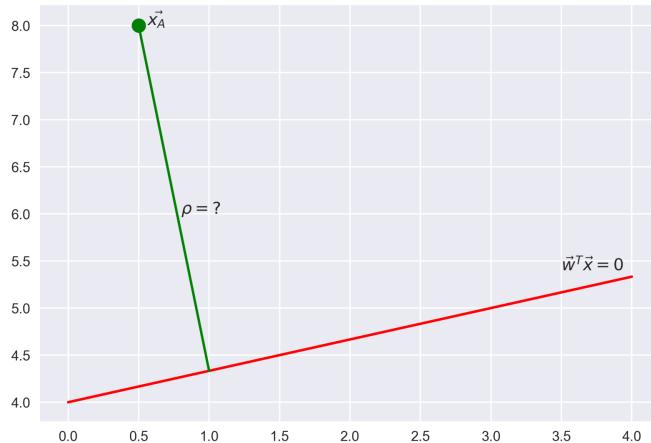


Fig. 51.2: Расстояние от точки до плоскости

$$x^2 + y^2 = 1$$

А мы выведем указанную выше формулу расстояния от точки до плоскости. Вообще это можно сделать разными способами – алгебраически и геометрически. Но давайте посмотрим на эту задачу (внезапно) как на задачу оптимизации и решим ее методом Лагранжа. Это послужит неплохой тренировкой перед тем как окунуться в SVM.

Итак, представим задачу в таком виде: хотим найти точку x на плоскости $\langle w, x \rangle = 0$ такую, что расстояние от x до точки x_0 минимально:

$$\begin{cases} \rho(x, x_0) \rightarrow \min_x \\ \langle w, x \rangle = 0 \end{cases} \quad (51.2)$$

$$\langle w, x \rangle = 0$$

Лагранжиан: $\mathcal{L}(x, \lambda) = \|x - x_0\|^2 + 2\lambda \langle w, x \rangle$. Тут мы для $\rho(x, x_0)$ подставили квадрат расстояния $\|x - x_0\|^2$ (такой переход от расстояния к его квадрат хорошо бы обосновать монотонностью оптимизируемой функции, но мы это опустим), а также перед коэффициентом λ для удобства поставили 2, что несущественно.

Далее надо приравнять нулю частные производные лагранжиана по его аргументам. Из $\frac{\partial \mathcal{L}}{\partial x} = 0$ получаем: $2(x - x_0) + 2\lambda w = 0 \Rightarrow x = x_0 - \lambda w$.

Теперь домножим это уравнение скалярно на w и выразим λ :

$$\langle w, x \rangle = \langle w, x_0 \rangle - \lambda \|w\|^2 \Rightarrow \lambda = \frac{\langle w, x_0 \rangle}{\|w\|^2}$$

Тогда наконец

$$\min_{x, \langle w, x \rangle = 0} \rho(x, x_0) = \|x - x_0\| = \|(x_0 - \lambda w) - x_0\| = |\lambda| \|w\| = \frac{\langle w, x_0 \rangle}{\|w\|} \quad \square.$$

Таким образом, линейный классификатор разделяет пространство на две части с помощью гиперплоскости, и при этом одно полупространство относит к положительному классу, а другое – к отрицательному.

Пожалуй, самый известный и популярный на практике представитель семейства линейных классификаторов – логистическая регрессия. На русском языке про нее можно почитать в [статье](#) открытого курса по машинному обучению или в упомянутых [лекциях](#) Евгения Соколова. В этих лекциях также объясняется, как происходит обучение модели (подбор весов w) за счет минимизации функции потерь.

51.2.1 Отступ классификации

Оказывается полезным рассмотреть выражение $M(x_i, y_i, w) = y_i \cdot \langle w, x \rangle$.

Это **отступ классификации** (margin) для объекта обучающей выборки (x_i, y_i) . К сожалению, его очень легко перепутать с зазором классификации, который появится чуть ниже в изложении интуиции метода опорных векторов. Чтобы не путать: отступ определен на конкретном объекте обучающей выборки.

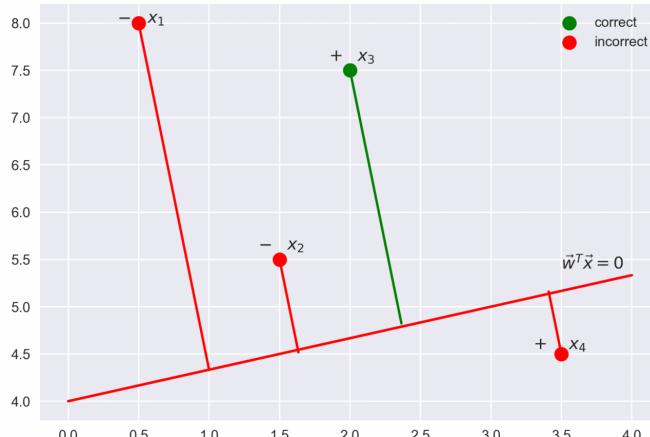


Fig. 51.3: Иллюстрация к понятию отступа классификации

Отступ – это своего рода “уверенность” модели в классификации объекта (x_i, y_i) :

- если отступ большой (по модулю) и положительный, это значит, что метка класса поставлена правильно, а объект находится далеко от разделяющей гиперплоскости (такой объект классифицируется уверенно). На рисунке – x_3 .
- если отступ большой (по модулю) и отрицательный, значит метка класса поставлена неправильно, а объект находится далеко от разделяющей гиперплоскости (скорее всего такой объект – аномалия, например, его метка в обучающей выборке поставлена неправильно). На рисунке – x_1 .
- если отступ малый (по модулю), то объект находится близко к разделяющей гиперплоскости, а знак отступа определяет, правильно ли объект классифицирован. На рисунке – x_2 и x_4 .

Далее увидим, что понятие отступа классификации – часть функции потерь, которая оптимизируется в методе опорных векторов.

51.3 Интуиция метода опорных векторов

Метод опорных векторов (Support Vector Machine, SVM) основан на идее максимизации зазора между классами. Пока не вводим этот термин формально, но передадим интуицию метода. На Рис. Fig. 51.4 показана линейно-разделимая выборка, кружки соответствуют положительным примерам, а квадраты – отрицательным (или наоборот), а оси – некоторым признакам этих примеров. На рисунке слева показаны две прямые (в общем случае – гиперплоскости), разделяющие выборку. Кажется, что синяя прямая лучше тем, что она дальше отстоит от примеров обучающей выборки, чем красная прямая (зазор – больше), и потому лучше будет разделять другие примеры из того же распределения, что и примеры обучающей выборки. То есть такой линейный классификатор будет лучше обобщаться на новые данные. Теория подтверждает описанную интуицию [MRT18].

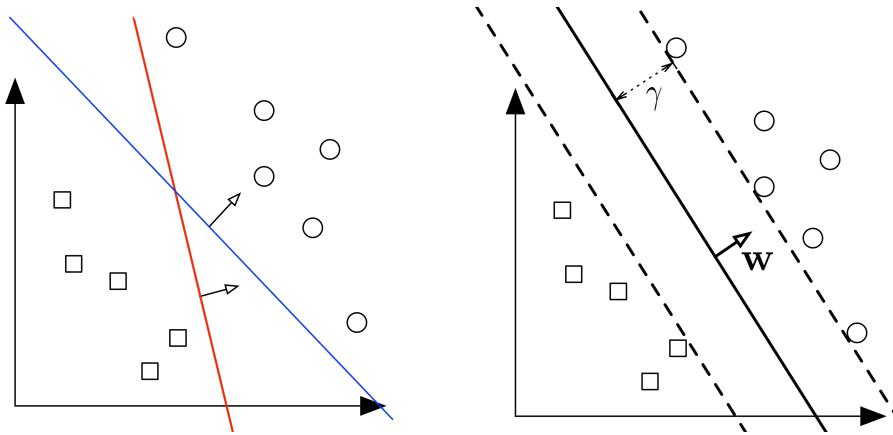


Fig. 51.4: Слева показаны две прямые (в общем случае – гиперплоскости), разделяющие выборку. Справа показана прямая, максимизирующая зазор между классами. Источник: [лекция Cornell](#)

Note: Одним из ключевых авторов алгоритма SVM является Владимир Вапник – советский и американский (с 1991-го года) ученый, который также сделал огромный вклад в теорию классического машинного обучения. Его имя носит одно из ключевых теоретических понятий машинного обучения – размерность Вапника-Червоненкиса.

51.4 Метод опорных векторов для линейно-разделимой выборки

Будем рассматривать линейные классификаторы вида

$$a(x) = \text{sign}(\langle w, x \rangle + b), \quad w \in \mathbb{R}^d, b \in \mathbb{R}.$$

Заметьте, что мы вернули сдвиг (bias) b . Будем считать, что существуют такие параметры w_* и b_* , что соответствующий им классификатор $a(x)$ не допускает ни одной ошибки на обучающей выборке. В этом случае говорят, что выборка **линейно разделима**.

Заметим, что если одновременно умножить параметры w и b на одну и ту же положительную константу, то классификатор не изменится. Распорядимся этой свободой выбора и отнормируем параметры так, что

$$\min_{x \in X} |\langle w, x \rangle + b| = 1.$$

Как мы увидели выше, расстояние от произвольной точки $x_0 \in \mathbb{R}^d$ до гиперплоскости, определяемой данным

классификатором, равно

$$\rho(x_0, a) = \frac{|\langle w, x \rangle + b|}{\|w\|}.$$

Тогда расстояние от гиперплоскости до ближайшего примера из обучающей выборки равно

$$\min_{x \in X} \frac{|\langle w, x \rangle + b|}{\|w\|} = \frac{1}{\|w\|} \min_{x \in X} |\langle w, x \rangle + b| = \frac{1}{\|w\|}.$$

Данная величина также называется **зазором** (margin). Опять же, эту величину очень легко перепутать с отступом классификации, про который мы говорили выше и который тоже margin в англоязычном варианте. Заметим, что отступ – функция параметров w и конкретного примера обучающей выборки (x_i, y_i) , а зазор – функция только параметров w при описанном трюке с нормировкой w и b (в противном случае зазор – также функция сдвига b и всех примеров x).

Таким образом, если классификатор без ошибок разделяет обучающую выборку, то ширина его разделяющей полосы равна $\frac{2}{\|w\|}$. Максимизация ширины разделяющей полосы приводит к повышению обобщающей способности классификатора [MRT18]. На повышение обобщающей способности направлена и регуляризация, которая штрафует большую норму весов – а чем больше норма весов, тем меньше ширина разделяющей полосы.

Итак, требуется построить классификатор, идеально разделяющий обучающую выборку, и при этом имеющий максимальный отступ.

Запишем соответствующую оптимизационную задачу, которая и будет определять метод опорных векторов для линейно разделимой выборки (hard margin support vector machine):

$$\begin{cases} \frac{1}{2} \|w\|^2 \rightarrow \min_{w,b} \\ y_i (\langle w, x_i \rangle + b) \geq 1, \quad i = 1, \dots, \ell. \end{cases}$$

Здесь мы воспользовались тем, что линейный классификатор дает правильный ответ на примере x_i тогда и только тогда, когда $y_i(\langle w, x_i \rangle + b) \geq 0$ (вспомним, что $M_i = y_i(\langle w, x_i \rangle + b)$ – отступ классификации для примера (x_i, y_i) обучающей выборки). Более того, из условия нормировки $\min_{x \in X} |\langle w, x \rangle + b| = 1$ следует, что $y_i(\langle w, x_i \rangle + b) \geq 1$.

В данной задаче функционал является строго выпуклым, а ограничения линейными, поэтому сама задача является выпуклой и имеет единственное решение. Более того, задача является квадратичной и может быть решена крайне эффективно.

51.5 Метод опорных векторов для линейно-неразделимой выборки

Рассмотрим теперь общий случай, когда выборку невозможно идеально разделить гиперплоскостью. Это означает, что какие бы w и b мы ни взяли, хотя бы одно из ограничений в предыдущей задаче будет нарушено:

$$\exists x_i \in X : y_i (\langle w, x_i \rangle + b) < 1.$$

Сделаем эти ограничения *мягкими*, введя штраф $\xi_i \geq 0$ за их нарушение:

$$y_i (\langle w, x_i \rangle + b) \geq 1 - \xi_i, \quad i = 1, \dots, \ell.$$

Отметим, что если отступ объекта лежит между нулем и единицей ($0 \leq y_i (\langle w, x_i \rangle + b) < 1$), то объект верно классифицируется, но имеет ненулевой штраф $\xi > 0$. Таким образом, мы штрафуем объекты за попадание внутрь разделяющей полосы.

Величина $\frac{1}{\|w\|}$ в данном случае называется **мягким зазором** (soft margin). С одной стороны, мы хотим максимизировать зазор, с другой – минимизировать штраф за неидеальное разделение выборки $\sum_{i=1}^{\ell} \xi_i$.

Эти две задачи противоречат друг другу: как правило, излишняя подгонка под выборку приводит к маленькому зазору, и наоборот – максимизация зазора приводит к большой ошибке на обучении. В качестве компромисса будем минимизировать взвешенную сумму двух указанных величин.

Приходим к оптимационной задаче, соответствующей методу опорных векторов для линейно неразделимой выборки (soft margin support vector machine):

$$\begin{cases} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{\ell} \xi_i \rightarrow \min_{w,b,\xi} \\ y_i(\langle w, x_i \rangle + b) \geq 1 - \xi_i, \quad i = 1, \dots, \ell, \\ \xi_i \geq 0, \quad i = 1, \dots, \ell. \end{cases}$$

Чем больше здесь параметр C , тем сильнее мы будем настраиваться на обучающую выборку. Данная задача также является выпуклой и имеет единственное решение.

51.5.1 Сведение к безусловной задаче оптимизации

Покажем, что задачу метода опорных векторов можно свести к задаче безусловной оптимизации функционала, который имеет вид верхней оценки на долю неправильных ответов.

Перепишем условия задачи:

$$\begin{cases} \xi_i \geq 1 - y_i(\langle w, x_i \rangle + b) \\ \xi_i \geq 0 \end{cases}$$

Поскольку при этом в функционале требуется, чтобы штрафы ξ_i были как можно меньше, то можно получить следующую явную формулу для них:

$$\xi_i = \max(0, 1 - y_i(\langle w, x_i \rangle + b)).$$

Данное выражение для ξ_i уже учитывает в себе все ограничения задачи, описанной выше. Значит, если подставить его в функционал, получим безусловную задачу оптимизации:

$$\frac{1}{2}\|w\|^2 + C \sum_{i=1}^{\ell} \max(0, 1 - y_i(\langle w, x_i \rangle + b)) \rightarrow \min_{w,b}$$

Эта задача является негладкой, поэтому решать её может быть достаточно тяжело. Тем не менее, она показывает, что метод опорных векторов, по сути, как и логистическая регрессия, строит верхнюю оценку на долю ошибок и добавляет к ней стандартную квадратичную регуляризацию. Только если в случае логистической регрессии этой верхней оценкой была логистическая функция потерь (опять сделаем ссылку к [статье](#) из открытого курса машинного обучения), то в случае метода опорных векторов это функция вида $L(y, z) = \max(0, 1 - yz)$, которая называется **кусочно-линейной функцией потерь (hinge loss)**.

Это становится понятнее в терминах упомянутого выше отступа $M_i = y_i(\langle w, x_i \rangle + b)$ на примере обучающей выборки. В идеале мы хотели бы штрафовать классификатор за ошибку на примере: $L_{1/0}(M_i) = [M_i < 0]$. Это пороговая функция потерь (zero-one loss), ее график изображен черным на [Fig. 51.5](#) как функция от отступа. К сожалению, напрямую мы не можем эффективно оптимизировать такую функцию градиентными методами из-за разрыва в нуле, поэтому оптимизируется верхняя оценка zero-one loss. В случае логистической регрессии – логистическая функция потерь $L(M_i) = \log(1 + e^{-M_i})$ (красная на рисунке выше), а в случае метода опорных векторов – кусочно-линейная функция $L(M_i) = \max(0, 1 - M_i)$ (зеленая на рисунке выше).

Note: Бытует мнение, что метод опорных векторов сегодня никогда не используется из-за его сложности (как минимум квадратичной по числу примеров). Однако, это не так. Линейный SVM вполне неплохо можно применять в задачах с высокой размерностью объектов обучающей выборки, например, для классификации

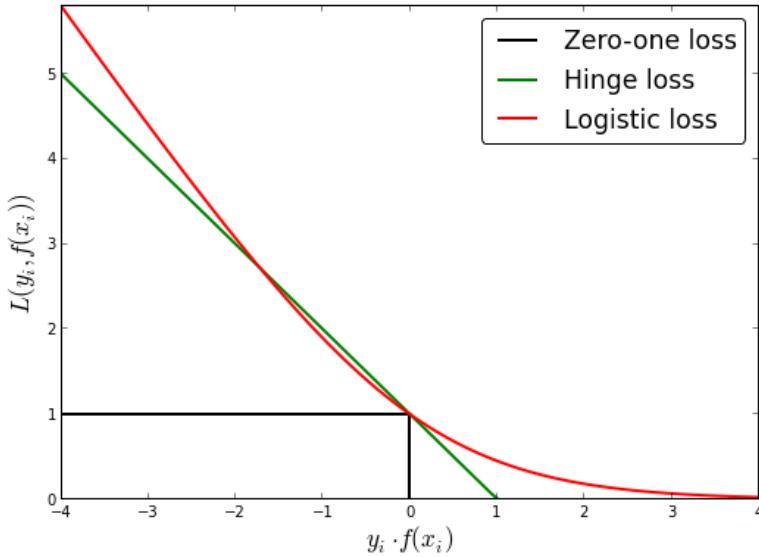


Fig. 51.5: Пороговая, кусочно-линейная и логистическая функции потерь

текстов с Tf-Idf или любым другим разреженным представлением. В частности, [Vowpal Wabbit](#) – очень эффективная утилита для решения многих задач машинного обучения – по умолчанию использует hinge-loss для задач классификации, то есть по сути в этом сценарии применения является линейным SVM. Кусочно-линейная функция потерь хороша тем, что у нее очень простая производная – положительная константа либо ноль. Это удобно использовать с SGD и большими выборками, когда приходится делать миллиарды обновлений весов.

Про прелести Vowpal Wabbit и обучение на гигабайтах данных за считанные минуты можно почитать в [статье](#) открытого курса машинного обучения.

51.6 Решение задачи метода опорных векторов

Итак, метод опорных векторов сводится к решению задачи оптимизации

$$\begin{cases} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{\ell} \xi_i \rightarrow \min_{w,b,\xi} \\ y_i (\langle w, x_i \rangle + b) \geq 1 - \xi_i, \quad i = 1, \dots, \ell, \\ \xi_i \geq 0, \quad i = 1, \dots, \ell. \end{cases}$$

Для решения таких *условных* задач оптимизации с условиями в виде неравенств или равенств часто используют лагранжиан и двойственную задачу оптимизации. Этот подход исчерпывающе описан в классической книге Бойда по оптимизации [BV04], а на русском языке можно обратиться к конспекту Евгения Соколова. Также для понимания материала рекомендуется рассмотреть “игрушечный” пример решения задачи метода опорных векторов в случае линейно-разделимой выборки из 5 примеров.

Построим двойственную задачу к задаче метода опорных векторов. Запишем лагранжиан:

$$L(w, b, \xi, \lambda, \mu) = \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{\ell} \xi_i - \sum_{i=1}^{\ell} \lambda_i [y_i (\langle w, x_i \rangle + b) - 1 + \xi_i] - \sum_{i=1}^{\ell} \mu_i \xi_i.$$

Выпишем условия Куна-Таккера:

$$\nabla_w L = w - \sum_{i=1}^{\ell} \lambda_i y_i x_i = 0 \implies w = \sum_{i=1}^{\ell} \lambda_i y_i x_i \quad (51.3)$$

$$\nabla_b L = - \sum_{i=1}^{\ell} \lambda_i y_i = 0 \implies \sum_{i=1}^{\ell} \lambda_i y_i = 0 \quad (51.4)$$

$$\nabla_{\xi_i} L = C - \lambda_i - \mu_i \implies \lambda_i + \mu_i = C \quad (51.5)$$

$$\lambda_i [y_i (\langle w, x_i \rangle + b) - 1 + \xi_i] = 0 \implies (\lambda_i = 0) \text{ или } (y_i (\langle w, x_i \rangle + b) = 1 - \xi_i) \quad (51.6)$$

$$\mu_i \xi_i = 0 \implies (\mu_i = 0) \text{ или } (\xi_i = 0) \quad (51.7)$$

$$\xi_i \geq 0, \lambda_i \geq 0, \mu_i \geq 0. \quad (51.8)$$

Проанализируем полученные условия. Из (51.3) следует, что вектор весов, полученный в результате настройки SVM, можно записать как линейную комбинацию объектов, причем веса в этой линейной комбинации можно найти как решение двойственной задачи.

В зависимости от значений ξ_i и λ_i объекты x_i разбиваются на три категории:

- $\xi_i = 0, \lambda_i = 0$. Такие объекты не влияют решение w (входят в него с нулевым весом λ_i), правильно классифицируются ($\xi_i = 0$) и лежат вне разделяющей полосы. Объекты этой категории называются *периферийными*.
- $\xi_i = 0, 0 < \lambda_i < C$. Из условия (51.6) следует, что $y_i (\langle w, x_i \rangle + b) = 1$, то есть объект лежит строго на границе разделяющей полосы. Поскольку $\lambda_i > 0$, объект влияет на решение w . Объекты этой категории называются *опорными граничными*.
- $\xi_i > 0, \lambda_i = C$. Такие объекты могут лежать внутри разделяющей полосы ($0 < \xi_i < 2$) или выходить за ее пределы ($\xi_i \geq 2$). При этом если $0 < \xi_i < 1$, то объект классифицируется правильно, в противном случае – неправильно. Объекты этой категории называются *опорными нарушителями*.

Отметим, что варианта $\xi_i > 0, \lambda_i < C$ быть не может, поскольку при $\xi_i > 0$ из условия дополняющей нежесткости (51.7) следует, что $\mu_i = 0$, и отсюда из уравнения (51.5) получаем, что $\lambda_i = C$.

Итак, итоговый классификатор зависит только от объектов, лежащих на границе разделяющей полосы, и от объектов-нарушителей (с $\xi_i > 0$).

Построим двойственную функцию. Для этого подставим выражение (51.3) в лагранжиан и воспользуемся уравнениями (51.4) и (51.5) (даные три уравнения выполнены для точки минимума лагранжиана при любых фиксированных λ и μ):

$$\begin{aligned} L &= \frac{1}{2} \left\| \sum_{i=1}^{\ell} \lambda_i y_i x_i \right\|^2 - \sum_{i,j=1}^{\ell} \lambda_i \lambda_j y_i y_j \langle x_i, x_j \rangle - b \underbrace{\sum_{i=1}^{\ell} \lambda_i y_i}_{0} + \sum_{i=1}^{\ell} \lambda_i + \sum_{i=1}^{\ell} \xi_i \underbrace{(C - \lambda_i - \mu_i)}_{0} \\ &= \sum_{i=1}^{\ell} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{\ell} \lambda_i \lambda_j y_i y_j \langle x_i, x_j \rangle. \end{aligned}$$

Мы должны потребовать выполнения условий (51.4) и (51.5) (если они не выполнены, то двойственная функция обращается в минус бесконечность), а также неотрицательность двойственных переменных $\lambda_i \geq 0, \mu_i \geq 0$. Ограничение на μ_i и условие (51.5), можно объединить, получив $\lambda_i \leq C$. Приходим к следующей двойственной

задаче:

$$\begin{cases} \sum_{i=1}^{\ell} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{\ell} \lambda_i \lambda_j y_i y_j \langle x_i, x_j \rangle \rightarrow \max_{\lambda} \\ 0 \leq \lambda_i \leq C, \quad i = 1, \dots, \ell, \\ \sum_{i=1}^{\ell} \lambda_i y_i = 0. \end{cases} \quad (51.9)$$

Она также является вогнутой, квадратичной и имеет единственный максимум.

51.7 Ядерный переход

Двойственная задача SVM (51.9) зависит только от скалярных произведений объектов – отдельные признаковые описания никак не входят в неё.

Note: Обратите внимание, как много это значит: решение SVM зависит только от скалярных произведений объектов (то есть *похожести*, если упрощать), но не от их признаковых описаний объектов. Это значит, что метод обобщается и на те случаи, когда признаковых описаний объектов нет или их получить очень дорого, но зато есть способ задать расстояние (то есть “измерить сходство”) между объектами.

Значит, можно сделать ядерный переход:

$$\begin{cases} \sum_{i=1}^{\ell} \lambda_i - \frac{1}{2} \sum_{i,j=1}^{\ell} \lambda_i \lambda_j y_i y_j K(x_i, x_j) \rightarrow \max_{\lambda} \\ 0 \leq \lambda_i \leq C, \quad i = 1, \dots, \ell, \\ \sum_{i=1}^{\ell} \lambda_i y_i = 0. \end{cases}$$

Здесь $K(x_i, x_j)$ – это функция-ядро, определенная на парах векторов, которая должна быть симметричной и неотрицательно определенной ([теорема Мерсера](#)).

Вернемся к тому, какое представление классификатора дает двойственная задача. Из уравнения (51.3) следует, что вектор весов w можно представить как линейную комбинацию объектов из обучающей выборки. Подставляя это представление w в классификатор, получаем

$$a(x) = \text{sign} \left(\sum_{i=1}^{\ell} \lambda_i y_i \langle x_i, x \rangle + b \right).$$

Таким образом, классификатор измеряет сходство нового объекта с объектами из обучающей выборки, вычисляя скалярное произведение между ними. Это выражение также зависит только от скалярных произведений, поэтому в нём тоже можно перейти к ядру.

Note: Опять подчеркнем, что классификация нового примера зависит только от скалярных произведений – “похожести” нового примера на примеры из обучающей выборки, и то не все, а только опорные.

Note: В указанном выше представлении фигурирует переменная сдвига b , которая не находится непосредственно в двойственной задаче. Однако ее легко восстановить по любому граничному опорному

объекту x_i , для которого выполнено $\xi_i = 0, 0 < \lambda_i < C$. Для него выполнено $y_i (\langle w, x_i \rangle + b) = 1$, откуда получаем

$$b = y_i - \langle w, x_i \rangle.$$

Как правило, для численной устойчивости берут медиану данной величины по всем граничным опорным объектам:

$$b = \text{med } y_i - \langle w, x_i \rangle, \xi_i = 0, 0 < \lambda_i < C.$$

Fig. 51.6 – пожалуй, самый известный рисунок в контексте SVM, он иллюстрирует ядерный трюк, в свою очередь, одну из самых красивых идей в истории машинного обучения. За счет ядерного перехода можно достигнуть линейной разделимости выборки даже в том случае, когда исходная обучающая выборка не является линейно разделимой.

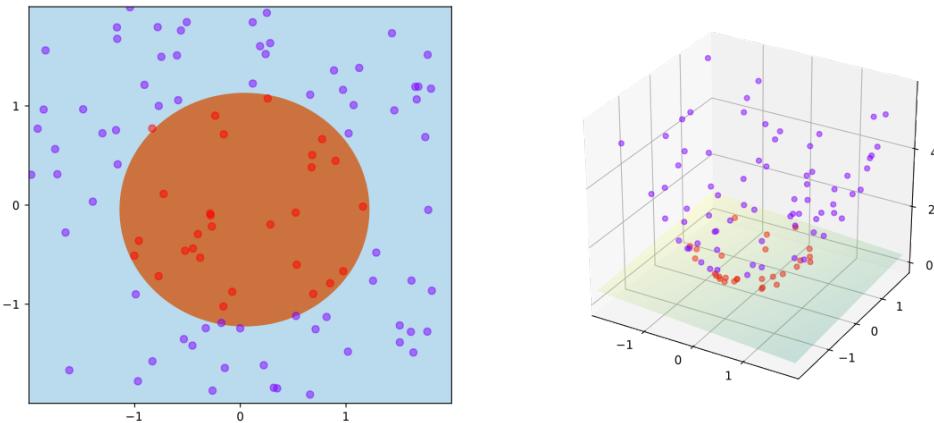


Fig. 51.6: Пример разделимости в новом пространстве

Наиболее часто используемые ядра:

- *Линейное* $K(x, y) = \langle x, y \rangle$ – по сути, линейный SVM, рассмотренный выше;
- *Полиномиальное ядро* $K(x, y) = (\langle x, y \rangle + c)^d$, определенное для степени ядра d и параметра нормализации c ;
- *Гауссово ядро*, также известное как RBF (radial-basis functions) $K(x, y) = e^{-\frac{\|x-y\|^2}{\sigma^2}}$ с параметром ядра σ .

51.8 Плюсы и минусы SVM

Плюсы:

- хорошо изучены, есть важные теоретические результаты;
- красиво формулируется как задача оптимизации;
- линейный SVM быстрый, может работать на очень больших выборках;
- линейный SVM так же хорошо интерпретируется, как и прочие линейные модели;

- решение зависит только от скалярных произведений векторов, а идея “ядерного трюка” – одно из самых красивых в истории машинного обучения;
- нелинейный SVM обобщается на работу с самыми разными типами данных (последовательности, графы и т.д.) за счет специфичных ядер.

Минусы:

- нелинейный SVM имеет высокую вычислительную сложность и принципиально плохо масштабируется (оптимизационную задачу нельзя “решить на подвыборках” и как-то объединить решения);
- нелинейный SVM по сути не интерпретируется (“black box”);
- в задачах классификации часто хочется выдать вероятность отнесения к классу, SVM это не умеет делать, а эвристики, как правило, приводят к плохо откалиброванным вероятностям;
- ядерный SVM уступает специфичным нейронным сетям уже во многих задачах, например, в приложениях к графикам

О БЛОКЕ “КВАНТОВЫЕ НЕЙРОННЫЕ СЕТИ”

Этот блок включает в себя обзор:

- квантовых нейронных сетей;
- сверточных вариационных квантовых схем.

КВАНТОВЫЕ НЕЙРОННЫЕ СЕТИ

53.1 Описание лекции

В этой лекции мы пройдёмся по расширению идеи нейронных сетей на квантовые компьютеры – мы уже прошли и *вариационные квантовые схемы (VQC)*, и комбинацию *квантовых и классических градиентов в них* в соответствующих блоках. Всё что осталось – это объединить всё изученное в общую картину и заняться обучением этих самых квантовых нейронных сетей. Для того чтобы лучше разобраться в том, какие ограничения и возможности предоставляет этот подход, мы приведём несколько публикаций о квантовых и гибридных генеративных состязательных сетях.

53.2 Введение

Как уже было упомянуто в *лекции по VQC*, на данный момент квантовые вычислители ещё недостаточно развиты для того, чтобы в одиночку решать большие задачи, имеющие практическое значение для индустрии – это в особенной степени актуально для нейронных сетей, которые и в классическом сценарии требуют значительных вычислительных ресурсов. Именно поэтому на данный момент наиболее популярна категория гибридных вариационных алгоритмов, которые обучают квантовую параметрическую схему (QNN) при помощи классической оптимизации, например, *VQ Eigensolvers* и *Quantum Approximate Optimization Algorithms*. В общем и целом идея гибридных алгоритмов заключается в оптимизации над некоторым классом параметрических вычислений для минимизации энергии волновой функции (**VQE/QAOA**), экстракции нелокальной информации (**QNN Classifiers**) или генерации данных, соответствующих квантовому распределению (**Quantum Generative Models**).

53.3 Применение

Лучше всего понять адекватность и применимость какой-то технологии, особенно основанной на комплексном научном базисе, позволяет как раз таки ее применение – чего удалось добиться кому-то на практике и насколько результаты пригодны для того, чтобы хвататься за технологию. Вместе с тем, практические результаты дадут нам глубже понять и в последствии объяснить, как гибридные сети работают под капотом. Приведём два примера, которые хорошо показывают разницу в ограничениях при использовании полностью квантовых и гибридных квантово-классических сетей.

В первом примере была построена полностью квантовая генеративная состязательная сеть, задачей которой являлось воспроизвести *MNIST*, однако у всего эксперимента был один нюанс. Из-за текущего размера квантовых вычислителей было предложено уменьшить размерность *MNIST* с 784 до 4 при помощи метода главных компонент (*PCA*), чтобы его хоть как-то можно было отправить в квантовую схему. Очевидно, сеть такого размера не в состоянии тягаться с классическими сетями на десятки тысяч весов, поэтому конечным результатом стало сравнение по количеству весов при идентичных результатах. Полностью квантовая сеть

смогла получить такие же результаты, как и классическая, при этом имея на 95% меньше параметров. Эти результаты всё ещё поднимают вопрос того, будут ли они справедливыми для моделей, адекватно справляющихся с задачей.

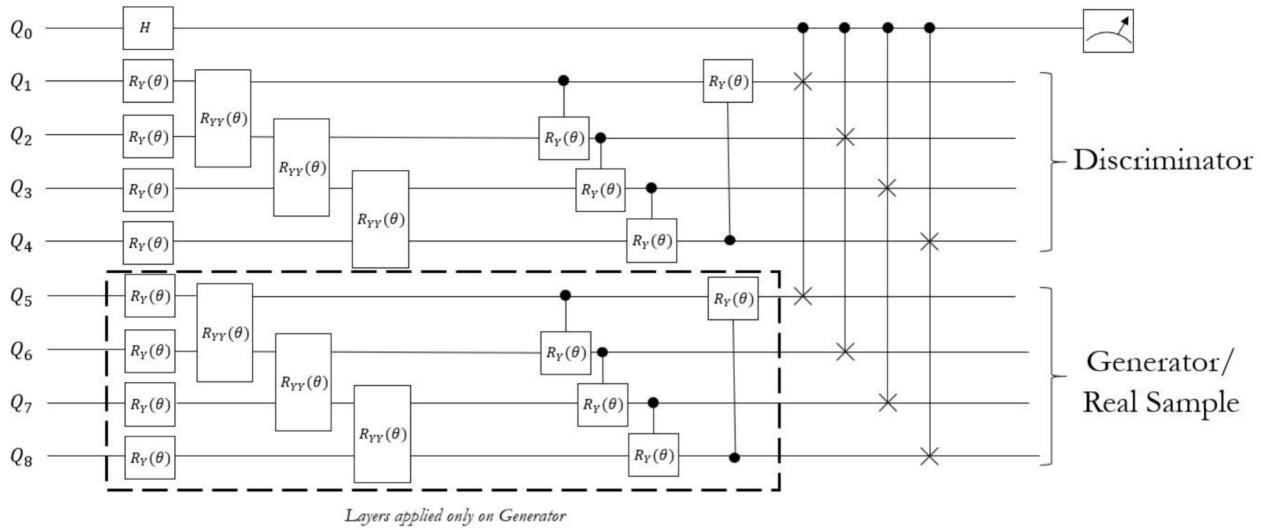


Fig. 53.1: Схема квантовой генеративной состязательной сети из публикации [SBC+21].

Во втором примере была построена гибридная квантово-классическая генеративная состязательная сеть, задачей которой являлось воспроизвести QM-9 – популярный датасет молекул до 9 атомов, представленный молекулярным графом в виде матрицы 9 на 9 и атомарным вектором из 9 элементов. Генеративная часть модели состояла из нескольких квантовых схем и, по большей части, классической сети, предсказывающей узлы и связи между ними. Дискриминативная часть была полностью классической во всех вариациях модели. В конечном итоге им удалось добиться снижения числа параметров по сравнению с полностью классической сетью на 85% и, при увеличении числа кубитов, на 98%. Таким образом они показали, что комбинация классических и квантовых схем должна соответствовать той же самой закономерности, что и полностью квантовые сети.

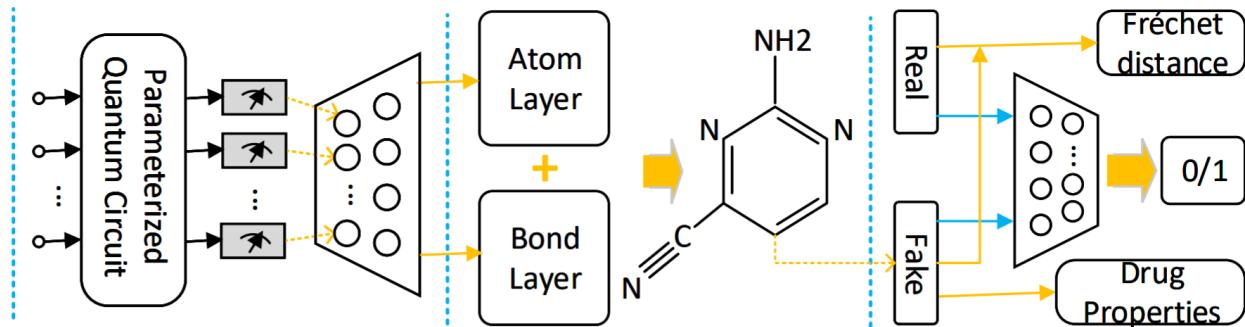


Fig. 53.2: Схема гибридной генеративной состязательной сети из публикации [LTG21].

За счёт большей актуальности на текущих практических задачах именно последнего, гибридного подхода, дальнейшая часть лекции будет рассматривать именно его.

53.4 Архитектура

В идеале этот подход подразумевал бы, что при помощи классического оптимизатора мы обучаем некоторую параметрическую схему на квантовом вычислителе, однако в текущих реалиях *NISQ* этот подход невозможен, поэтому большая часть параметрической схемы остается на классических вычислителях. В данном блоке мы поговорим о подходе, связанном с **QNN Classifiers**, которые следуют вышеупомянутому принципу и обучаются градиентным спуском практически так же, как и обычные классические сети, позволяя градиенту протекать между квантовой и классической частью сети.

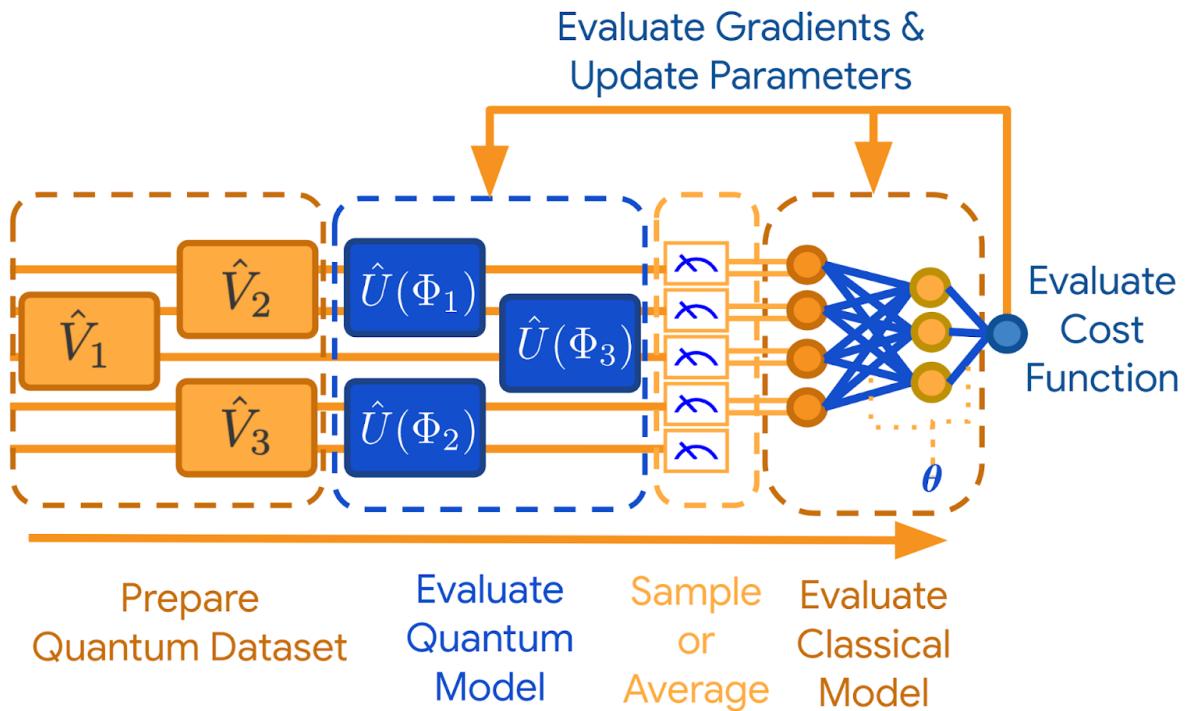


Fig. 53.3: Схема обучения гибридной нейронной сети из [BVM+21].

На изображении гибридной сети процедура практически идентична классическому обучению сетей, в котором добавляется процесс кодирования классических данных в квантовые операторы и процесс измерения квантового состояния для того, чтобы передать уже классическую информацию для дальнейших вычислений на классическом устройстве, как это было описано в [лекции по VQC](#).

53.5 Анзац

Зачастую в литературе по **VQC**, особенно когда речь идёт о нейронных сетях, упоминается такая вещь как **ansatz** – по своей сути это заранее подготовленные участки параметрической схемы, которые могут быть использованы как составные блоки сети. Если проводить параллели с классическим машинным обучением, то в рамках библиотеки PennyLane эти схемы называются **templates** (шаблоны) и могут представлять собой, например, свёрточный слой или эмбеддинг, а также более общие элементы квантовой схемы вроде подготовки состояний или перестановок между кубитами. Более подробно мы остановимся на них слегка позже в курсе, в одной из следующих глав, а пока что в общих чертах пройдёмся по обучению квантовых нейронных сетей без сложностей внутренней кухни.

Note: Интересно, но термин *ansatz* пришел в квантовые вычисления и QML из теоретической физики. Этот термин имеет немецкое происхождение, так как в первой половине XX века именно немецкие научные журналы были самыми передовыми. Частое употребление этого термина в отношении квантового машинного обучения объясняется тем, что большая часть специалистов в этой области это именно люди, занимающиеся теоретической физикой.

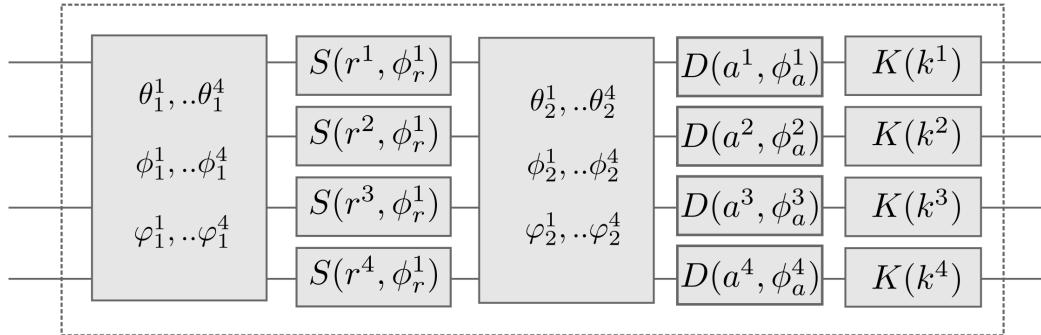


Fig. 53.4: Ansatz, соответствующий свёрточному слою нейронной сети в PennyLane. Источник

53.6 Функция потерь

Функция потерь работает таким же образом, как и в полностью классических сетях, так как оптимизация происходит на классическом железе. Единственное, что отличается, – это объединение квантовых и классических градиентов. Градиент по нашей квантовой схеме получается при помощи замера состояния, которое может варьироваться из-за вероятностной природы кубита, поэтому несколько замеров позволяют аппроксимировать ожидаемый градиент при помощи методов вроде *finite differences* или *parameter-shift*, после чего остаётся только совместить его с классическим градиентом.

53.7 Сеть от начала до конца

В конечном итоге мы имеем следующую последовательность действий для того, чтобы собрать гибридную нейронную сеть:

- трансформировать данные из классических в квантовые представления;
- отправить эти данные для вычисления на квантовой схеме;
- просэмплировать и замерить результат квантовой схемы;
- отправить результаты для вычисления на классической схеме;
- оценить ошибку, рассчитать градиенты и обновить параметры.

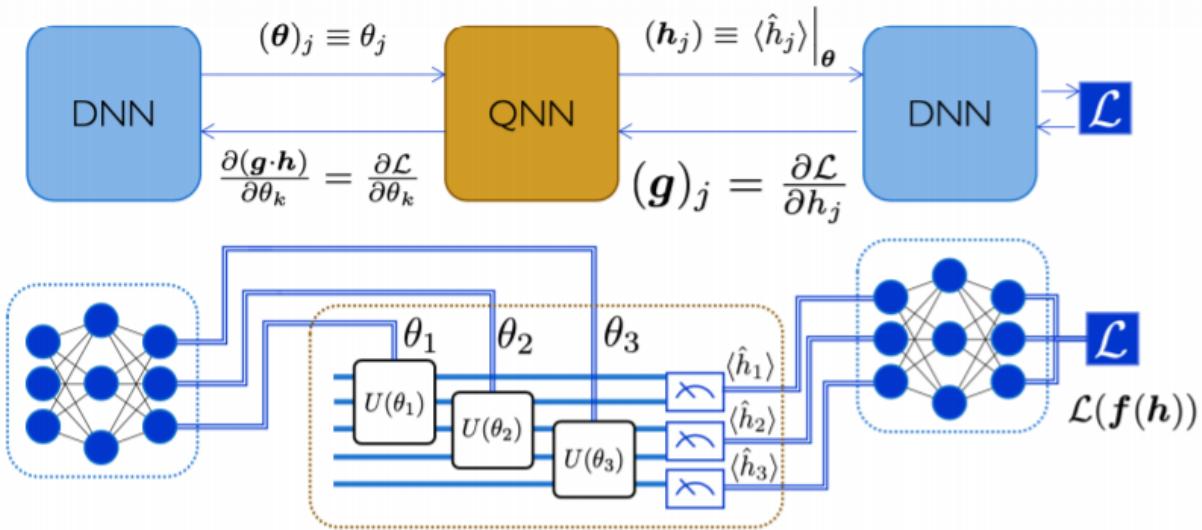


Fig. 53.5: Распространение градиентов от функции потерь в гибридной схеме.

53.8 Что мы узнали из лекции

- В ближайшие годы полностью квантовые нейронные сети не смогут решать задачи целиком, поэтому будут использоваться в качестве составляющей гибридного квантово-классического решения.
- Так же как и для полностью квантовых сетей, гибридные сети позволяют уменьшить количество необходимых параметров по сравнению с полностью классическими сетями.
- Обучение подобных сетей практически идентично обучению классических сетей за исключением нескольких трюков, необходимых для работы с параметрами квантовых схем.

СВЕРТОЧНЫЕ ВАРИАЦИОННЫЕ КВАНТОВЫЕ СХЕМЫ

54.1 Описание лекции

На этой лекции мы рассмотрим аналог сверточных слоев нейронной сети и приведем пример их использования. Лекция расскажет:

- как работают “классические” свертки в нейронных сетях;
- что такое сверточные вариационные квантовые схемы;
- как их можно использовать для задач машинного обучения на примере датасета MNIST.

54.2 Сверточные нейронные сети

Данный тип нейронных сетей нацелен на обработку изображений и представляет из себя последовательный набор сверточных слоев, чередующиеся с другими вспомогательными слоями, например, с функциями нелинейного преобразования (активациями). Пример работы сверточного слоя представлен на рисунке ниже.

Свое название сверточный слой получил из-за наличия операции свертки (конволюции), суть которой в том, что каждый фрагмент входного изображения (input) умножается на матрицу (ядро размера 3x3, kernel) свертки поэлементно, а результат суммируется и записывается в аналогичную позицию выходного изображения (output).

Сверточные слои являются ключевыми элементами алгоритмов по распознаванию и классификации изображений. Большинство успехов в области компьютерного зрения за последнее десятилетие связано именно с ними.

54.3 Квантовая свертка

Активное развитие квантовых вычислений создает запрос на построение квантового аналога свертки. Работа ([Henderson M. et.al., 2019](#)), описывающая подобный подход, предлагает следующую схему реализации на основе вариационных квантовых схем:

- изображение разбивается на блоки размера 2x2;
- для каждого блока осуществляется кодирование входного сигнала к квантовому состоянию, которое в данной лекции осуществляется параметризованным вращением кубита из основного состояния;
- квантовые вычисления выполняются над входными кубитами посредством случайной квантовой цепи;
- проводится измерение квантовой системы для получения классических величин;

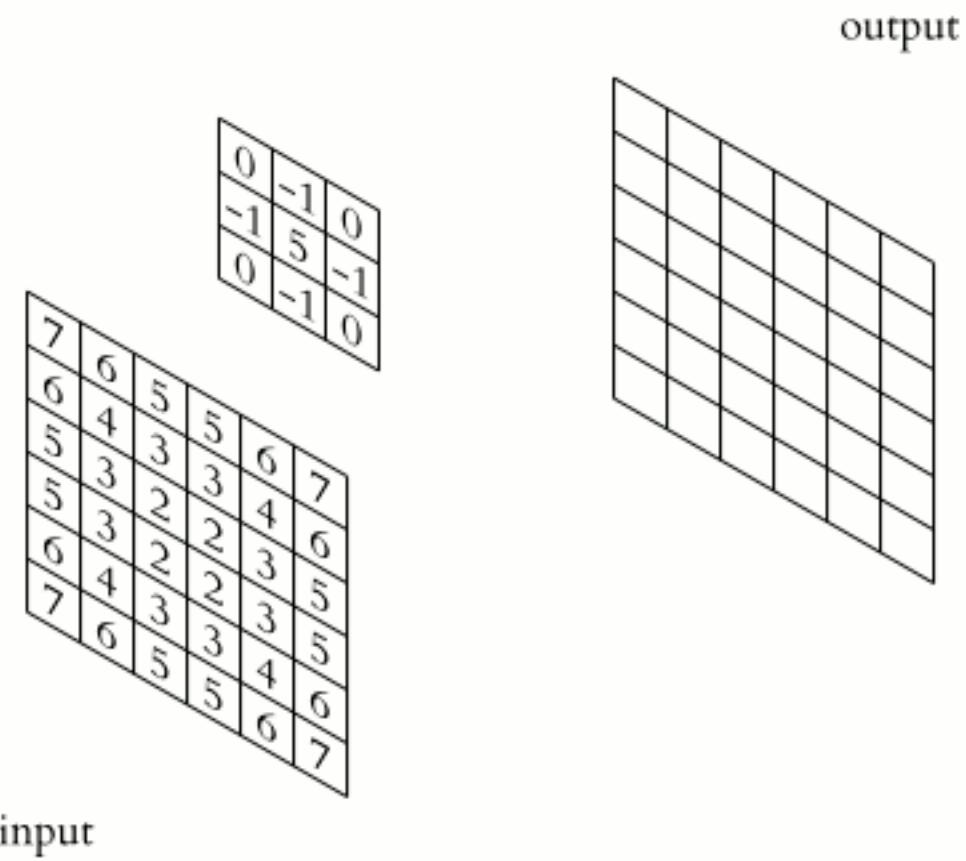


Fig. 54.1: Классическая свертка

- выполняя данную процедуру для всех 2×2 областей на выходе получается многоканальное изображение (четырехканальное на примере ниже), которое будет использоваться в следующих слоях классической или квантовой нейронной сети.

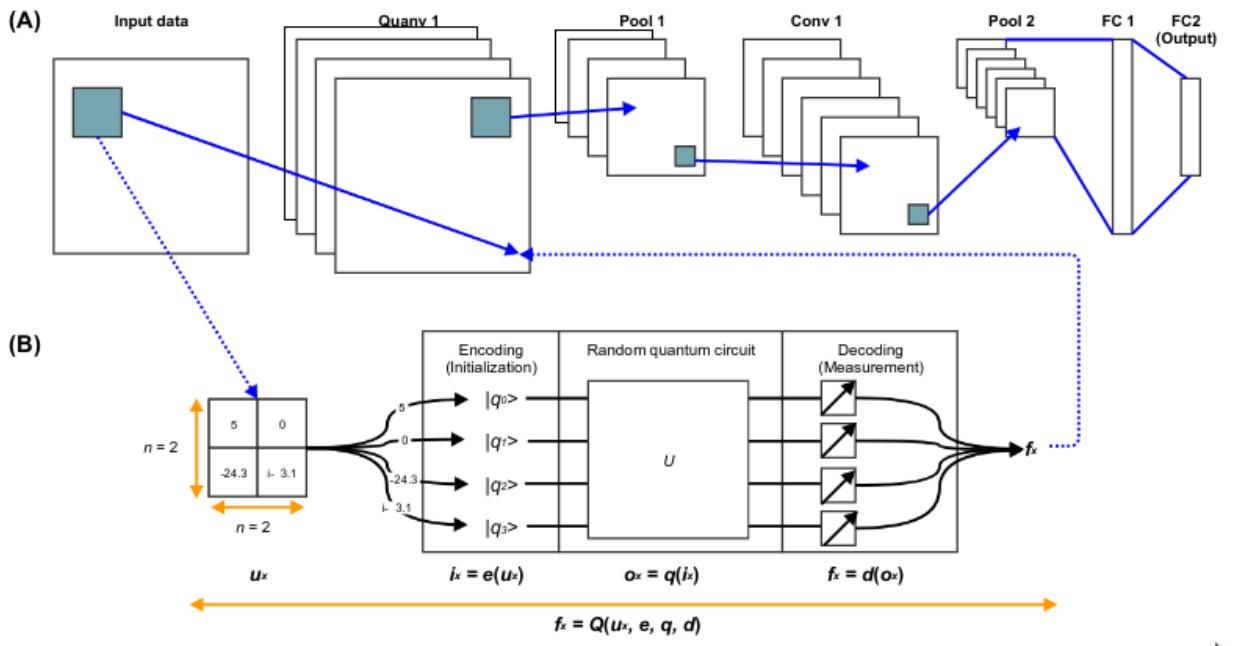


Fig. 54.2: Квантовая свертка

Основное отличие по отношению к классической свертке состоит в том, что квантовая схема может генерировать очень сложные ядра, вычисления которых могут быть классически невоспроизводимыми. Указанные сложные ядра, а именно ядра в высокоразмерном пространстве Гильберта, могут дать преимущества квантовым сверткам по отношению к классическим.

Пример реализации и работы такого блока квантовой свертки будет продемонстрирован ниже.

54.4 Подготовка модели и данных

Пример кода реализации гибридной квантово-классической сети с квантовым сверточным слоем взят из [документации](#) библиотеки PennyLane с небольшими изменениями.

Сначала загружаем библиотеки PennyLane для квантовых вычислений, tensorflow для обучения классификатора и matplotlib для отрисовки результатов.

```
import pennylane as qml
from pennylane import numpy as np
from pennylane.templates import RandomLayers

import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import tensorflow as tf
from tensorflow import keras
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'
```

Устанавливаем параметры модели для обучения: число эпох, число слоев квантовой свертки и его выходных каналов, а также число тренировочных и тестовых примеров. Нужно использовать 4 кубита, для обработки каждого из 4 пикселей входного блока и генерации 4 каналов. Число примеров выбрано малым для увеличения скорости демонстрации. Также фиксируем инициализацию генераторов случайных чисел для numpy и tensorflow.

```
n_epochs = 20      # Количество эпох обучения
n_layers = 1       # Число случайных квантовых блоков
n_wires = 4        # Число выходных каналов после квантовых блоков
n_train = 20        # Размер тренировочного датасета
n_test = 10         # Размер тестового датасета

# Инициализация генераторов случайных чисел
np.random.seed(0)
tf.random.set_seed(255)
```

Для демонстрации используем датасет MNIST, который создан для предсказания цифры (от 0 до 9) по его изображению

```
mnist_dataset = keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist_dataset.load_data()

# Ограничение размера датасета
train_images = train_images[:n_train]
train_labels = train_labels[:n_train]
test_images = test_images[:n_test]
test_labels = test_labels[:n_test]

# Нормализация изображений из диапазона (0, 255) в (0, 1)
train_images = train_images / 255
test_images = test_images / 255

# Добавление дополнительной размерности к данным для сверточных каналов
train_images = np.array(train_images[..., tf.newaxis], requires_grad=False)
test_images = np.array(test_images[..., tf.newaxis], requires_grad=False)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/
  ↘mnist.npz
```

```
8192/11490434 [.....] - ETA: 0s
```

```
8224768/11490434 [=====>.....] - ETA: 0s
```

```
11493376/11490434 [=====] - 0s 0us/step
```

54.5 Случайная квантовая цепь

В качестве блока квантовой свертки используется **случайная квантовая цепь** (RandomLayers в PennyLane). Блок реализуется путем случайного выбора для части кубитов операций вращения, а для части пар кубитов парных квантовых гейтов. На последнем этапе генерации блока осуществляется случайное перемешивание очередности применения операций.

В нашем примере случайная квантовая цепь обрабатывает 4 кубита. Таким образом часть из 4 кубитов получит какие-либо операции вращения, а часть из 6 пар кубитов - гейты. Финальная очередьность операций будет случайной.

Следует заметить, что в нашем примере используется фиксированный (необучаемый) блок случайной квантовой цепи. Он будет использован как блок препроцессинга. В будущем, когда в библиотеке будет реализована возможность расчета градиентов и изменения параметров блока на их основании, указанный блок тоже можно будет обучать.

54.6 Реализация квантовой свертки

Далее создаем устройство, симулирующее работу 4 кубитов. Оно состоит из трех частей:

- кодировщик, преобразующий входные данные в квантовые состояния с помощью RY поворотов;
- случайная квантовая цепь;
- измерения, дающие 4 выходных значения.

```
dev = qml.device("default.qubit", wires=n_wires)

# Генерация значений параметров для квантовых слоев
rand_params = np.random.uniform(high=2 * np.pi, size=(n_layers, n_wires))

@qml.qnode(dev)
def circuit(phi):
    # Кодирование 4 классических входных данных
    for j in range(n_wires):
        qml.RY(np.pi * phi[j], wires=j)

    # Случайная квантовая цепь
    RandomLayers(rand_params, wires=list(range(n_wires)))

    # Измерения, которые дают 4 классических выходных значений для следующих слоев
    return [qml.expval(qml.PauliZ(j)) for j in range(n_wires)]
```

Следующая функция quanv определяет квантовую свертку по следующей схеме:

- разделение изображения на блоки 2×2 пикселей;
- обработку каждого блока квантовой цепью `circuit`, описанной выше;
- 4 выходных значения для каждого блока помещаются в 4 разных канала одного пикселя выходного изображения.

Описанный выше процесс уменьшает разрешение входного изображения в 2 раза, что эквивалентно классической свертке с ядром 2×2 и шагом 2

```
def quanv(image):
    """Функция квантовой свертки над входным изображением."""
```

(continues on next page)

(continued from previous page)

```

out = np.zeros((14, 14, n_wires))

# Циклы по координатам верхнего левого пикселя блоков 2x2
for j in range(0, 28, 2):
    for k in range(0, 28, 2):
        # Обработка блока 2x2 из изображения квантовой цепью
        q_results = circuit(
            [
                image[j, k, 0],
                image[j, k + 1, 0],
                image[j + 1, k, 0],
                image[j + 1, k + 1, 0]
            ]
        )
        # Запись результатов наблюдения в выходной пиксель (j/2, k/2)
        for c in range(n_wires):
            out[j // 2, k // 2, c] = q_results[c]
return out

```

54.7 Подготовка датасета

Так как квантовый сверточный слой в нашем случае не обучается, то лучше провести предварительную обработку им всех используемых изображений из датасета. Подготовленные данные будут использоваться классической нейронной сетью для обучения модели классификатора.

```

q_train_images = []
for idx, img in enumerate(train_images):
    q_train_images.append(quanv(img))
q_train_images = np.asarray(q_train_images)
print("Препроцессинг тренировочных изображений квантовой сверткой выполнен.")

q_test_images = []
for idx, img in enumerate(test_images):
    q_test_images.append(quanv(img))
q_test_images = np.asarray(q_test_images)
print("Препроцессинг тестовых изображений квантовой сверткой выполнен.")

```

Препроцессинг тренировочных изображений квантовой сверткой выполнен.

Препроцессинг тестовых изображений квантовой сверткой выполнен.

Давайте посмотрим на 4 первых тренировочных примера (первый ряд) и их 4 канала подготовленных данных (2-5 строки)

```

n_samples = 4
n_channels = n_wires
fig, axes = plt.subplots(1 + n_channels, n_samples, figsize=(10, 10))

for k in range(n_samples):
    axes[0, 0].set_ylabel("Input")
    if k != 0:
        axes[0, k].yaxis.set_visible(False)

```

(continues on next page)

(continued from previous page)

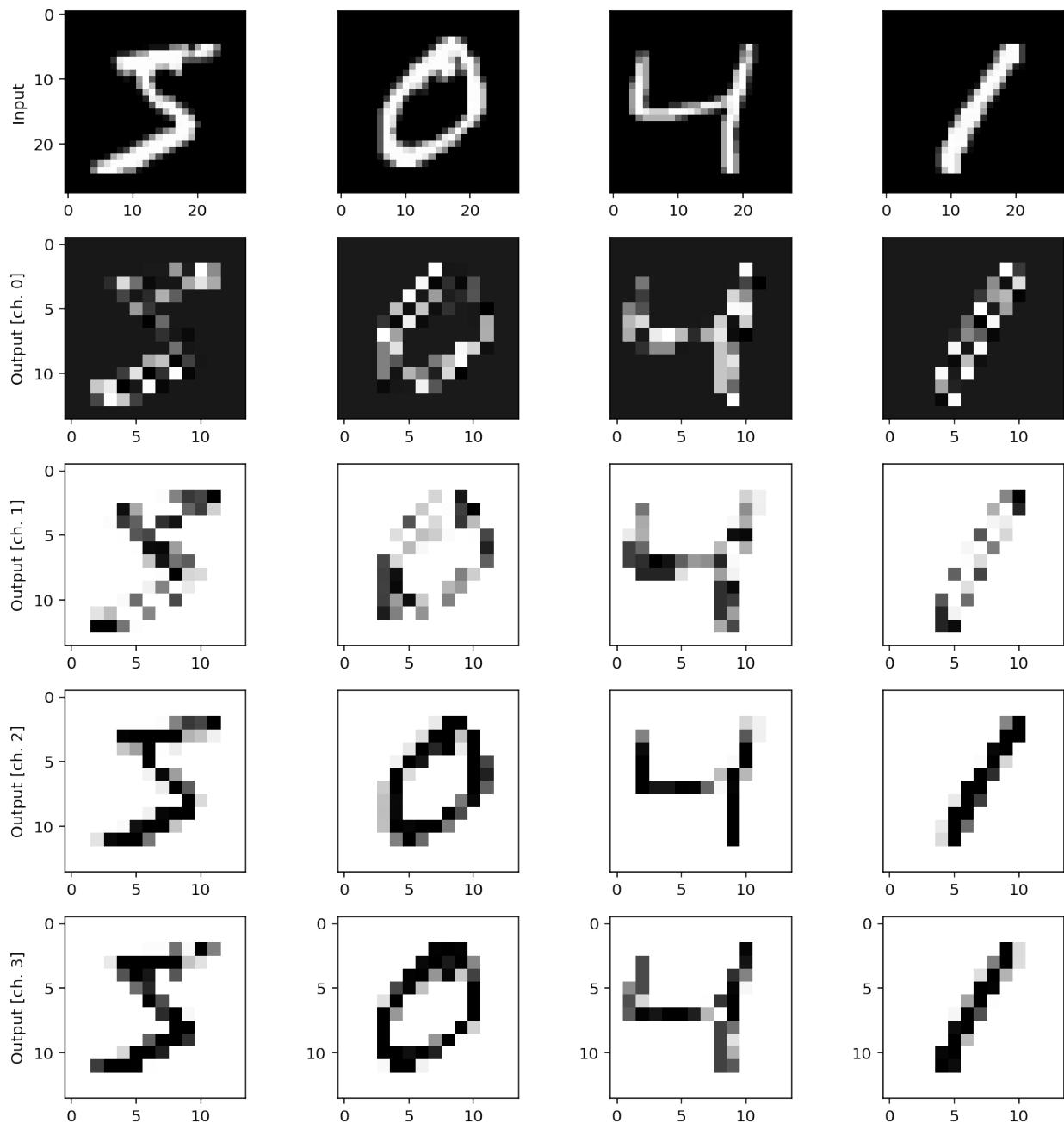
```

axes[0, k].imshow(train_images[k, :, :, 0], cmap="gray")

# Отрисовка
for c in range(n_channels):
    axes[c + 1, 0].set_ylabel(f"Output [ch. {c}]")
    if k != 0:
        axes[c, k].yaxis.set_visible(False)
    axes[c + 1, k].imshow(q_train_images[k, :, :, c], cmap="gray")

plt.tight_layout()
plt.show()

```



По обработанным данным (последние 4 ряда) видно уменьшение разрешения в 2 раза, а также то, что пространственная структура цифр сохраняется, что и должно наблюдаться для сверточных слоев

54.8 Гибридная квантово-классическая модель

Ниже опишем классическую часть нейронной сети, которая будет обучена для классификации 10 цифр. Будет использована очень простая модель, основанная на одном полносвязном слое и финальном применении функции softmax, которая выдает вероятности представленных 10 классов.

Обучение будет осуществлено с помощью оптимизатора Adam по функции потерь, в качестве которой используется кросс-энтропия.

```
def MyModel():
    """Функция инициализирует и возвращает keras модель, готовая к обучению"""

    model = keras.models.Sequential([
        keras.layers.Flatten(),
        keras.layers.Dense(10, activation="softmax")
    ])

    model.compile(
        optimizer='adam',
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model
```

54.9 Тренировка моделей

Сначала обучим модель на данных, подготовленных с помощью квантовой цепи.

```
q_model = MyModel()

q_history = q_model.fit(
    q_train_images,
    train_labels,
    validation_data=(q_test_images, test_labels),
    batch_size=4,
    epochs=n_epochs,
    verbose=0,
)
```

Для сравнения с результатами чисто классической модели проведем ее обучение на входных необработанных картинках.

```
c_model = MyModel()

c_history = c_model.fit(
    train_images,
    train_labels,
    validation_data=(test_images, test_labels),
```

(continues on next page)

(continued from previous page)

```

batch_size=4,
epoches=n_epochs,
verbose=0,
)

```

54.10 Сравнение результатов

Сравнение проведем на основе тестовой точности и значению функции потерь в зависимости от номера эпохи

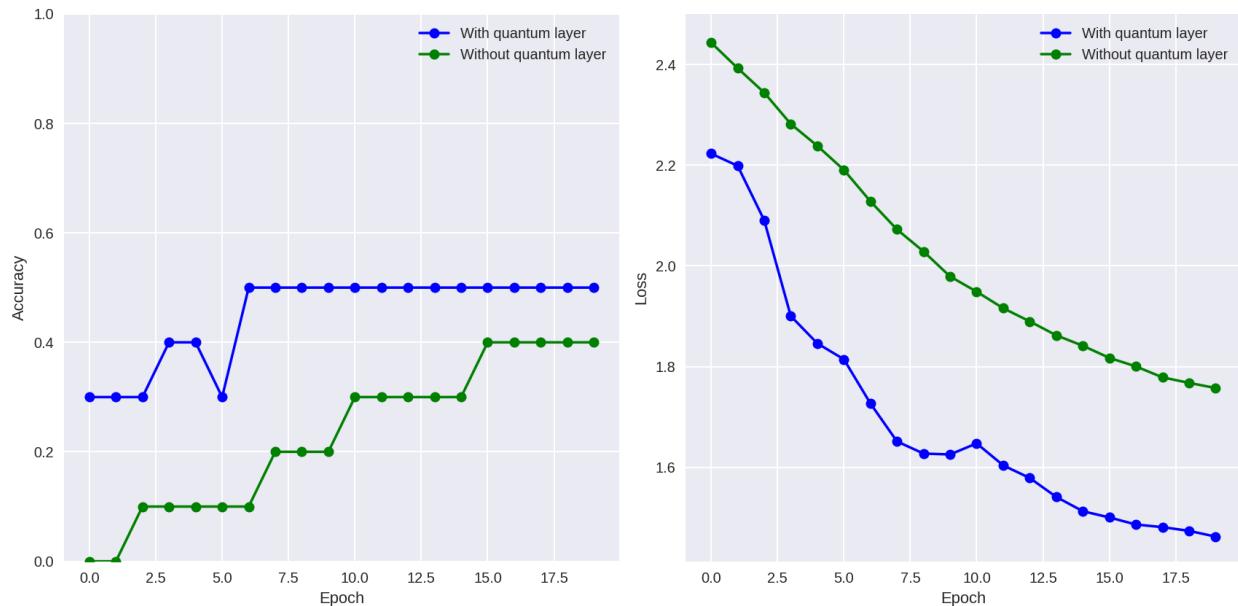
```

plt.style.use("seaborn")
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

ax1.plot(q_history.history["val_accuracy"], "-ob", label="With quantum layer")
ax1.plot(c_history.history["val_accuracy"], "-og", label="Without quantum layer")
ax1.set_ylabel("Accuracy")
ax1.set_ylim([0, 1])
ax1.set_xlabel("Epoch")
ax1.legend()

ax2.plot(q_history.history["val_loss"], "-ob", label="With quantum layer")
ax2.plot(c_history.history["val_loss"], "-og", label="Without quantum layer")
ax2.set_ylabel("Loss")
ax2.set_ylim(top=2.5)
ax2.set_xlabel("Epoch")
ax2.legend()
plt.tight_layout()
plt.show()

```



Из рисунков видно наличие сильного сходства результатов для двух моделей: гибридной и чисто классической. В наших малых экспериментах достигнута довольно низкая точность в 50% (классический аналог - 40%). Однако при увеличении числа используемых примеров точность должна быть существенно выше.

54.11 Замечания

- в статье авторов подхода ([Henderson M. et.al., 2019](#)) для гибридной и чисто классической сети с большим количеством слоев (см. [схему сети](#)) и большим объемом данных, достигнута ожидаемая высокая точность выше 95%;
- блоки квантовой свертки сейчас особо интересны, так как они ориентированы на использование малого количества кубитов, которое доступно в текущий период времени для устройств с промежуточным количеством кубитов (NISQ).

54.12 Что мы узнали из лекции

- существует аналог сверточных нейронных сетей для квантовых вычислений;
- аналог квантовой свертки может быть реализован с использованием случайной квантовой цепи;
- использование квантовой свертки позволяет достичь результатов схожих с классическими сетями;
- основное преимущество квантовой свертки по отношению к классической в том, что первая является более широкой операцией, которая не может быть описана классическим случаем.

О БЛОКЕ “ПРОБЛЕМЫ КВАНТОВОГО МАШИННОГО ОБУЧЕНИЯ”

Этот блок будет полностью посвящен описанию тех задач реального мира, для которых использование квантового машинного обучения на NISQ-компьютерах выглядит наиболее перспективно.

Тут мы рассмотрим:

- основы квантовой химии;
- модель Изинга;
- задачи комбинаторной оптимизации.

Продвинутые темы блока более подробно расскажут об этих темах.

КВАНТОВАЯ ХИМИЯ. ВВЕДЕНИЕ.

56.1 Описание лекции

Из этой лекции узнаем:

- что такое квантовая химия, что с ее помощью можно сделать, а что – нельзя;
- как устроена самая простая теория, описывающая квантовую физику атома;
- какие у этой теории есть ограничения.

56.2 Введение

В предыдущих лекциях говорили о применении квантовой механики для вычислений и обработки информации. Однако, исторически квантовая теория развивалась в первую очередь из-за того, что классическая физика не могла объяснить некоторые наблюдаемые эффекты: дискретный спектр излучения атомов, фотоэффект в металлах, интерференцию частиц на дифракционных щелях.

Считается, что квантовая механика дает полное и точное описание состояния и эволюции любой системы при нерелятивистских условиях – по крайней мере, на сегодняшний день этому нет экспериментальных или теоретических противоречий. Это значит, что в теории возможно для любой системы частиц записать уравнения Шредингера, решить их и предсказать, как себя поведет система. Между тем, на практике оказывается, что в реальных задачах вроде моделирования лекарств и материалов, просто “взять и посчитать” – задача весьма сложная, а иногда – неразрешимая.

Проблемами применения квантовой механики к химии и материаловедению занимается квантовая химия. Она делает это уже около 100 лет, по теме написаны толстые книжки с многоэтажными формулами, так что в лекции будут даны основы и простые примеры без полного вывода.

Предполагается, что читатель знаком с уравнением Шредингера и основными операторами (импульса, эволюции), бра-кет нотацией, а также помнит основы физики и химии на уровне старших классов школы.

56.3 Атом водорода

56.3.1 В чем проблема?

Спектр излучения и поглощения атомов – то есть на какой длине волны происходит поглощение и излучение света веществом – был одной из первых “нерешаемых” проблем, приведших в итоге к появлению квантовой физики. Для простых веществ в газообразной форме спектр является дискретным, и для атома водорода спектры поглощения и излучения в видимом диапазоне выглядят так:

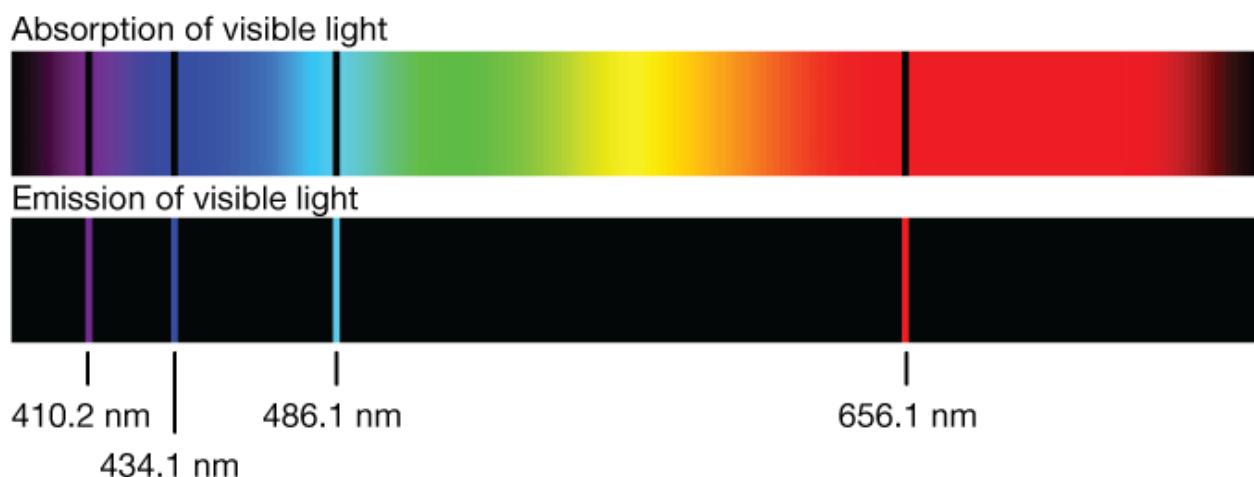


Fig. 56.1: Спектр поглощения и излучения водорода в видимом диапазоне

В конце 19 века было уже известно, что атом водорода состоит из двух заряженных частиц – протона и электрона, но из классической физики следовало, что спектр излучения такой системы должен быть непрерывным, что никак не стыковалось с наблюдениями (дискретные линии на графике). Попробуем вслед за исследователями квантовой физики начала 20 века разобраться с этой проблемой.

Про имена

“Квантовая физика” и “квантовая механика” – связанные понятия, но не тождественные.

Квантовая механика – это теория (а точнее, множество теорий с различающимися терминологиями), в основе которой лежит аксиоматика о том, что сущности описываются волновыми функциями, что они эволюционируют и, зная эти функции, можно посчитать физические величины – например, энергию.

Квантовая физика – это область физики, исследующая квантовые эффекты, при этом она местами может не иметь строгого обоснования – лишь бы предсказания работали.

С квантовой химией тоже есть некоторая путаница – в двадцатом веке под ней имели в виду аналитические и численные методы решения задач квантовой механики применительно к молекулам и кристаллам. Но с развитием квантовых компьютеров эту область стали иногда называть вычислительной химией, а квантовой химией – применение квантовых компьютеров к этой области.

56.3.2 Быстрое и неправильное решение

Первой теорией в квантовой физике была “Боровская теория” – почти классическая теория, основанная на постуатах, разрешающих противоречия классической физики с экспериментами на уровне “хардкода”. Главный постулат заключается в том, что момент импульса может принимать только определенные дискретные значения (кванты), а не произвольные:

$$p \cdot r = n \cdot \hbar,$$

и отсюда

$$p = \frac{n \cdot \hbar}{r},$$

где $p = mv$ – импульс электрона, r – радиус орбиты электрона, \hbar – постоянная Планка, n – любое целое число. Если применить этот постулат к атому водорода – паре из протона и электрона, то получится, что:

- электрон вокруг протона “летает по орбите”;
- спектр (уровни энергии электрона) – дискретный.

Можно показать это следующим образом.

Во-первых, поскольку частиц две и протон имеет массу много больше массы электрона, можно перейти в почти инерциальную систему отсчета протона, где он неподвижен, и решить задачу только для электрона.

Из классической механики берем [теорему о вириале](#) (“для стабильной системы из двух частиц, связанных потенциальными силами, в среднем кинетическая энергия равна половине потенциальной”):

$$\begin{aligned} E_k &= -E_p/2 \\ E &= E_k + E_p = E_p/2 \end{aligned}$$

Записываем кинетическую энергию и потенциал Кулона для электрона в поле протона (в СИ):

$$\frac{p^2}{2m} = k \frac{e^2}{2R},$$

где k – постоянная из закона Кулона.

Используем постулат Бора и выразим “радиус” орбиты:

$$\begin{aligned} \frac{n^2 \hbar^2}{2m R^2} &= k \frac{e^2}{2R} \\ R &= \frac{n^2 \hbar^2}{k m e^2} \end{aligned}$$

Находим полную энергию:

$$E = E_p/2 = -\frac{ke^2}{2R} = -\frac{k^2}{n^2} \frac{me^4}{2\hbar^2} \quad (56.1)$$

Уровни энергии пропорциональны $\frac{1}{n^2}$ и именно [такая зависимость](#) наблюдается в эксперименте.

Посчитаем минимальную энергию ($n = 1$):

```
from scipy import constants as consts

# ПОСТОЯННАЯ КУЛОНА
k = (1 / (4 * consts.pi * consts.epsilon_0))
```

(continues on next page)

(continued from previous page)

```
# энергия в Джоулях
E1 = - k**2 * consts.m_e * consts.e**4 / (2 * consts.hbar ** 2)

# энергия в электрон-вольтах
E1_ev = E1 / consts.e

print(f"Hydrogen Bohr ground state energy: {E1_ev} eV")
```

```
Hydrogen Bohr ground state energy: -13.605693122885837 eV
```

Получившийся ответ (13.6 eV) в точности совпадает с экспериментальным значением энергии ионизации водорода.

Итого: постулировав несколько очень удачных гипотез (главная – что момент импульса квантуется, то есть принимает только дискретные значения), удалось разрешить фундаментальную нестыковку между классической теорией и экспериментом: энергия стала дискретной, линии спектра (длины волн) стали обратно пропорциональны $\left(\frac{1}{n_{final}^2} - \frac{1}{n_{initial}^2}\right)$, значения энергии ионизации совпадают с реальными измерениями.

Казалось бы, замечательно, проблема решена! Однако теория Бора имеет ряд проблем. Главная из них заключается в том, что теория работает только для “водородоподобных” атомов, то есть состоящих из ядра и одного электрона на внешней оболочке. Уже для атома гелия спектр не согласуется с боровской теорией.

Кроме того, с точки зрения науки, теория Бора – это в некотором смысле “читетство”. Выбрав удачные постулаты и подставив соответствующие формулы в законы классической физики, получили правильный результат. Но будь постулаты другими, результат получился бы тоже другой, так что фактически постулаты Бора – это гениальная догадка, позволившая угадать правильные результаты для некоторых систем.

КВАНТОВАЯ ХИМИЯ. ТЕОРИЯ САМОСОГЛАСОВАННОГО ПОЛЯ.

57.1 Описание лекции

Из этой лекции узнаем:

- как с помощью квантовой химии предсказать спектр атома водорода “из первых принципов”;
- как посчитать энергию атома с помощью Python;
- какие бывают волновые функции электронов и как их вычислять методом Self-Consistent Field;
- как посчитать энергию спирта.

57.1.1 Введение

Боровская теория оказалось недостаточной для описания таких сложных систем, как молекулы или неводородоподобные атомы. В этой лекции разберемся с теорией самосогласованного поля – это один из первых вычислительных методов квантовой химии, пригодный на практике для моделирования разных систем (атомов, молекул, кристаллов).

Несмотря на то, что основа метода была разработана чуть ли не сто лет назад, он активно используется и сегодня, особенно с различными модификациями и дополнениями.

57.1.2 Что мы ищем?

С точки зрения квантовой химии, чаще всего хотим для произвольного набора частиц (атомов для молекулы или кристалла, протонов и электронов для атома) с известным потенциалом взаимодействия предсказывать стационарное состояние, т.е. находить стационарные волновые функции всех частиц системы. Квантовая механика постулирует, что для любой наблюдаемой физической величины существует оператор, которым можем действовать на волновую функцию и получить измеренное значение этой величины.

Например, если волновые функции электронов и ядер в кристалле известны, то можно предсказать теплопроводность кристалла. Или электрическую проводимость, или еще что-нибудь – поэтому основной задачей квантовой химии является поиск волновых функций-решений уравнения Шредингера для системы частиц.

57.1.3 Долгое и правильное решение, часть 1

Начнем с атома водорода и определения его уровней энергии – их можно сравнить с экспериментальными данными, а также результатами Боровской теории. Квантовая механика позволяет решить эту задачу, исходя только из “первых принципов”, то есть аксиоматики квантовой механики. При этом после некоторых усилий решения успешно обобщаются с атома водорода на любой другой.

Давайте опишем атом водорода – систему из электрона и протона – на языке квантовой механики, то есть в виде уравнения Шредингера, и посмотрим, что получится.

Электрон находится в потенциале протона, и его волновая функция должна зависеть от расстояния до протона.

Уравнение Шредингера для электрона будет следующим:

$$i\hbar \frac{d}{dt} \Psi(t, \vec{r}) = \left(\frac{\hat{p}^2}{2m} + \hat{V}(\vec{r}) \right) \Psi(t, \vec{r})$$

Мы ищем решение, в котором электрон остается в атоме, то есть решаем стационарное уравнение, в котором Ψ не зависит от времени. Тогда оператор эволюции (в левой части уравнения Шредингера) при применении к Ψ должен вернуть нам тот же вектор Ψ , умноженный на E (энергия частицы).

С точки зрения математики, искомая волновая функция является собственным вектором оператора эволюции, а энергия – собственным значением (подробнее в разделе *Матрицы*). Если объяснять “на пальцах”, то оператор эволюции при применении к волновой функции должен вернуть нам новую (эволюционированную или изменившуюся во времени) волновую функцию. Если ищем стационарную – не меняющуюся во времени – волновую функцию, то при применении к ней оператора эволюции она не должна изменяться, иначе будет уже не стационарной.

$$i\hbar \frac{d}{dt} \Psi(t, \vec{r}) \equiv i\hbar \frac{d}{dt} \Psi(\vec{r}) \equiv E\Psi(\vec{r})$$

Оператор импульса раскрывается через градиент так:

$$\hat{p} = i\hbar \nabla$$

Потенциал взаимодействия двух частиц с противоположными единичными зарядами раскрывается так:

$$\hat{V}(\vec{r}) = -\frac{e^2}{r}$$

Итого имеем:

$$\left(-\frac{\hbar^2}{2m} \nabla^2 - \frac{e^2}{r} \right) \Psi(\vec{r}) = E\Psi(\vec{r}) \quad (57.1)$$

Прежде чем погружаться глубже в решение этого уравнения, нам надо разобраться с квантовыми числами.

57.1.4 Про квантовые числа

Теория Бора базируется на идее квантования импульса, и ее следствием является возникшее в формуле энергии (56.1) число n . Каждое число n соответствует определенному состоянию, в котором может находиться электрон, и эти состояния отличаются энергией.

Идея, что электрон может находиться в одном состоянии из некоторого конечного набора вариантов, характеризуемого дискретными (квантовыми) числами, оказалась правильной. В дальнейшем она получила развитие в квантовой механике.

Число n называется главным *квантовым числом*, оно определяет энергетический уровень электрона.

Из анализа свойств операторов гамильтонiana и момента импульса в квантовой механике выводятся еще три квантовых числа:

- ℓ – орбитальное квантовое число, $0 \leq \ell \leq n - 1$;
- m – магнитное квантовое число, $-\ell \leq m \leq \ell$;
- s – спиновое квантовое число, $s = 0|1$ (для атома водорода оно не играет роли, так как электрон только один).

В химии они [используются](#) в другой форме записи для описания орбиталей электронов в атоме.

Эти числа нам скоро понадобятся, чтобы описывать электрон в атоме.

Любая функция Ψ , для которой уравнение (57.1) верно, описывает электрон, который стабильно находится где-то около ядра и не покидает его, то есть образует с ним атом. При этом функций-решений у уравнения на самом деле много, что физически соответствует тому, что электрон может находиться на разных орбитах. Каждая орбиталь характеризуется квантовыми числами – n , ℓ , m и s , и обозначается как $\Psi_{n\ell m}$. Ψ (орбиталь) с минимальной энергией E соответствует основному состоянию (ground state) – она описывает невозбужденный электрон. У водорода только один электрон, поэтому единственная Ψ с минимальной энергией соответствует невозбужденному атому водорода.

В целом, основная задача квантовой химии – найти ground state произвольной системы частиц. В реальности материя редко находится непосредственно в ground state состоянии, т.к. при ненулевой температуре какие-то флуктуации неизбежно возникают. Тем не менее, ground state обычно в наибольшей степени определяет поведение системы, а другие состояния являются “поправками” к нему.

Частицы могут переходить из одного состояния в другое: если электрону в основном состоянии придать энергии, например, попасть в него другой частицей, то он может перейти на другую орбиталь, при этом оставшись частью атома. Со временем возбужденные атомы рано или поздно переходят в основное состояние, выбрасывая лишнюю энергию в виде фотона, что и дает спектр излучения атома.

Итого: $\Psi_{n\ell m}(\vec{r})$ – это функция, соответствующая какой-то стабильной “траектории” (распределению плотности вероятности) электрона вокруг ядра, она является решением уравнения Шредингера, то есть собственной функцией гамильтониана. Соответствующее ей собственное число – энергия электрона на этой орбитали. Для всех волновых функций с одним n энергия одинакова, но одной энергии может соответствовать несколько волновых функций, каждая – со своим уникальным набором волновых чисел.

Если электрон описывается такой волновой функцией, то он часть атома, и если какой-либо электрон – часть атома, то он описывается такой волновой функцией. Электрон может переходить между этими волновыми функциями, получая и отдавая энергию, оставаясь при этом частью атома.

Волновая функция Ψ_{100} (т.е. с числами $n = 1, \ell = 0, m = 0$) соответствует минимальной энергии для единственного электрона в атome водорода. Т.к. в атome водорода есть лишь один электрон, и других вкладов в энергию нет – весь атome водорода в такой конфигурации имеет минимальную энергию.

57.1.5 Долгое и правильное решение, часть 2

А зачем нам вообще сдался спектр и энергии?

Вслед за физиками двадцатого века нам приходится проридаться через дебри уравнений, абстракций и формул. Дело это непростое и может возникнуть резонный вопрос – а зачем оно все надо?

Помимо фундаментальной ценности в виде лучшего понимания устройства мира, объясняющая спектр теория дает крутую возможность: по измеренному спектру понять, что за вещество перед нами и какие процессы в нем протекают. Например, изучение спектров – один из немногих способов узнать что-то про процессы в звездах или биологических клетках. Для совсем практиков: спектроскопия также используется для детекции взрывчатых и наркотических веществ в малых дозах, а полевые транзисторы (основа большинства современных вычислительных устройств) работают на основе туннельного эффекта – квантового явления. Так что можно сказать, что квантовая механика используется сплошь и рядом!

Если перейти в уравнении (57.1) в сферические координаты со следующей параметризацией $\Psi(\vec{r}) = \frac{\chi(r)}{r} Y(\theta, \phi)$ и воспользоваться несколькими волшебными выводами квантовмеха [89], то получится:

$$-\frac{\hbar^2}{2m} \frac{d^2\chi}{dr^2} + \left(-\frac{e^2}{r} + \frac{\hbar^2\ell(\ell+1)}{2mr^2} - E\right)\chi(r) = 0$$

Все константы (массы, заряда, импульса, энергии) можно убрать, если перейти в кулоновские единицы измерений, где они приняты за единицу (то есть 1 единица заряда = заряд электрона, 1 единица массы = масса электрона).

$$-\frac{\chi''(r)}{2} + \left(\frac{\ell(\ell+1)}{2r^2} - \frac{1}{r}\right)\chi(r) = 0$$

Опустим несколько страниц выкладок [15a], учтем граничные условия и получим следующее решение:

$$R_{n\ell}(r) = r^\ell \cdot e^{-r/n} \cdot \sum_{k=0}^{n-\ell-1} \frac{(-2r/n)^k}{(2\ell+2+k)!(n-\ell-k-1)!k!} \cdot C$$

$$\Psi(r, \theta, \phi) = R_{n\ell}(r)Y_{\ell m}(\theta, \phi),$$

где $Y_{\ell m}(\theta, \phi)$ – сферические функции.

Если подставить это решение в уравнение Шредингера и найти энергию, то получим:

$$E_n = -\frac{1}{n^2} \frac{me^4}{2\hbar^2}$$

То есть получим ту же формулу, что и в теории Бора, и тот же численный результат – 13.6 eV.

Здесь начинает проступать основная проблема квантовой химии – математическая и вычислительная сложность. Пока что проблема только концептуальная (сложно разобраться в формулах и уравнениях), но при росте числа частиц в системе даже отличное владение матаппаратом окажется недостаточным.

57.1.6 От теории к практике

Вернемся из начала двадцатого века обратно в день сегодняшний. Сто лет назад невычислимые аналитически интегралы и замороченные дифференциальные уравнения были почти непреодолимыми препятствиями, поэтому поначалу теория квантовой химии развивалась в сторону более хитрых приближений и упрощений, позволяющих решить эти уравнения аналитически.

Сегодня реальные научные задачи решаются численно – и для этого написано множество высокопроизводительных пакетов, позволяющих на основе входных данных и ограничений вычислить определенным методом желаемые характеристики.

Как и в задачах машинного обучения, в первом приближении достаточно подготовить данные, скормить их волшебному программному пакету-вычислителю и забрать ответ, но без понимания происходящего под капотом есть шансы получить что-то неправильное.

В этой лекции воспользуемся Python-пакетом `psi4`. Он реализует многие алгоритмы квантовой химии и имеет неплохое Python API.

Давайте посчитаем с помощью `psi4` энергию основного состояния атома водорода. Некоторые параметры сейчас придется использовать, “поверив на слово”. Их смысл будет объяснен в дальнейшем.

```
import psi4
psi4.core.be_quiet() # отключаем логирование в stdout

h_atom = psi4.geometry("H")
```

Мы задали атом водорода, по умолчанию атом помещается в начало координат.

```
psi4.set_options({
    'basis': 'STO-3G',
    'reference': 'rohf',
})
```

Тут уже поинтереснее – объяснение этих параметров пока отложим и вернемся после объяснения теории. В целом они определяют, каким именно методом и в каком базисе нужно численно решить уравнение Шредингера.

```
from scipy.constants import physical_constants

h2ev = physical_constants['hartree-electron volt relationship']

def e_in_ev(energy_in_ht):
    return energy_in_ht * h2ev[0]

# Энергия в единицах Hartree
e_in_ht = psi4.energy('scf')

print(f"Hydrogen ground state energy: {e_in_ev(e_in_ht)} eV")
```

```
Hydrogen ground state energy: -12.696338923670485 eV
```

Здесь посчитали энергию в единицах Хартри – специальной физической системе единиц, где истинная энергия атома водорода равна 1/2, и перевели ее в электрон-вольты. Результат не очень точный (правильный, как помним, равен 13.6 eV), и мы его улучшим после того как разберемся, что и как только что посчитали. Это будет удобнее сделать на примере атома гелия, потому что в атоме водорода есть только один электрон, а в любой реальной системе – больше одного.

57.2 Теория самосогласованного поля

57.2.1 Атом гелия

Следующим “по простоте” после атома водорода идет атом гелия – как говорит нам школьная химия, это атом из двух протонов, двух нейтронов и двух электронов. Протоны и нейтроны находятся близко друг к другу в ядре и имеют почти одинаковую массу, так что можно просто считать, что есть ядро с зарядом +2 и массой 4. А вот с электронами все сложнее: с одной стороны, это независимые частицы, а с другой – они взаимодействуют друг с другом по закону Кулона, так как оба имеют отрицательный заряд.

Note: Для гравитационного взаимодействия “проблема трех тел” не имеет известного аналитического решения. Это означает, что если мы знаем, что где-то в глубоком космосе вдалеке от остального мира есть три объекта с известными массами, импульсами и координатами, то, увы, в общем случае не сможем предсказать их движение аналитически (хотя сможем предсказать численно, либо найти приближенное аналитическое решение, если масса одного объекта много больше других, например).

Для трех классических тел с кулоновским потенциалом все тоже сложно – можно посмотреть [тут](#), как поведет себя система трех тел с различными зарядами.

Попробуем записать уравнение Шредингера для системы из ядра и двух электронов:

$$i\hbar \frac{d}{dt} \Psi(t, \vec{r}_A, \vec{r}_B) = \left(\frac{\hat{p}_A^2}{2m} - \frac{2e \cdot e}{r_A} + \frac{\hat{p}_B^2}{2m} - \frac{2e \cdot e}{r_B} + \frac{e \cdot e}{|r_A - r_B|} \right) \Psi(t, \vec{r}_A, \vec{r}_B)$$

Оно состоит из кинетической и потенциальной энергии двух электронов в поле ядра и (последнее слагаемое) потенциала взаимодействия двух электронов. Учитывая, что уравнение примерно такое же, как у водорода, только еще более сложное, решение стоит искать численно и с использованием приближений, а не аналитически.

Если бы не последний член гамильтониана, то можно было бы разбить все выражение на две независимых части – с переменными электрона и с переменными электрона B . Так как это дифференциальное уравнение, можно было бы воспользоваться разделением переменных и найти отдельные решения для двух электронов – задача аналогична атому водорода, а ее мы уже решили.

Но из-за потенциала взаимодействия решение существенно усложняется, поскольку электроны влияют друг на друга. Придется прибегнуть к упрощениям – и одним из наиболее популярных подходов является *теория самосогласованного поля* (Self-Consistent Field).

57.2.2 Теория самосогласованного поля

Теория самосогласованного поля (self-consistent field theory) – это подход итеративного решения уравнения Шредингера для многочастичной системы, на основе которого построено много квантово-химических методов, наиболее известный из которых – метод Хартри-Фока. В снппете выше строчка `psi4.energy('scf')` означает, что энергия посчитана этим методом.

Основная идея теории заключается в следующем.

У нас есть несколько частиц, которые взаимодействуют между собой, и найти цельное решение уравнения Шредингера для всех сразу не получается. Тогда вместо этого будем рассматривать частицы по очереди и считать, что все остальные действуют “в среднем” на выбранную частицу. То есть будем считать усредненный по пространству потенциал вместо точного.

Для электрона в атоме гелия нам нужно учесть усредненное влияние электрона B . Можем для электрона B взять волновую функцию от атома водорода, посчитать на ее основе усредненное влияние на электрон :

$$(h_A + \hat{V}_{eff})\psi_A = (h_A + \psi_B \frac{e^2}{r_{AB}}\psi_B)\psi_A = \epsilon_A \psi_A,$$

где h_A – это кинетическая энергия и потенциал ядра для электрона A , V_{eff} – влияние электрона B на электрон и вычисляется как влияние усредненной электронной плотности, распределенной в соответствии с ψ_B .

На этом шаге считаем, что ψ_B – фиксированная волновая функция и находим “переменную” ψ_A . Однако ψ_A тоже будет влиять на ψ_B и, записав аналогичное уравнение для частицы B , следующим шагом найдем новую ψ_B .

После изменения волновых функций каждого электрона можем заново записать их уравнения с новыми волновыми функциями и так по кругу, пока волновые функции и их энергии не сойдутся к какому-то стабильному (самосогласованному) решению.

Все вместе это создает итеративную процедуру:

1. на основе имеющихся волновых функций посчитать среднее поле, которое создают частицы (например, электрон B для электрона A);
2. решить уравнение Шредингера с потенциалом, учитывающим среднее поле, то есть вычислить энергию и новые волновые функции;
3. вернуться к шагу 1.

Для инициализации можно взять какие-то приблизительные волновые функции для всех частиц (двух электронов гелия), например, решения из уравнения водорода, то есть водородоподобные волновые функции. Далее итерацию повторяют, пока волновые функции и их энергии не перестанут изменяться, то есть самосогласуются.

Система таких уравнений, записанных для каждого электрона, называется *уравнениями Хартри*. На основе таких волновых функций для отдельных электронов можно собрать общую волновую функцию $\Psi(\vec{r}_A, \vec{r}_B)$, самый

простой вариант – это $\Psi(\vec{r}_A, \vec{r}_B) = \psi_A(\vec{r}_A)\psi_B(\vec{r}_B)$, он и был предложен первоначально. Однако есть проблема: такая Ψ получается не антисимметричной, а только такие волновые функции для системы электронов являются “физичными”, то есть могут существовать в реальности.

Note: Один из основополагающих принципов квантовой механики – тождественность частиц и их неразличимость. Принцип означает, что все частицы одного типа (например, электроны) одинаковы и характеризуются только своим состоянием. Например, если мы “переставим местами” два электрона в атоме (не только пространственно, но и в смысле их состояния и энергий), то получившийся атом будет неотличим от исходного.

Из принципа тождественности следует, что при перестановке не должна меняться плотность вероятности:

$$|\Psi(X_A, X_B)|^2 = |\Psi(X_B, X_A)|^2$$

Кроме того, представим что мы обменяли частицы дважды: A с B и обратно. Никакие физические свойства системы из-за этого измениться не должны. В общем случае, из этого не следует что волновая функция не изменилась – например, так как глобальная фаза волновой функции неизмерима, то могли получить $\Psi' = e^{2i\pi\theta}\Psi$.

Но, если работаем больше чем в двух пространственных измерениях, то такой двойной обмен эквивалентен отсутствию обмена. Для начала представим, что обмениваем частицы медленно, описывая полукруг частицей A вокруг частицы B и потом сдвигая обе частицы. Тогда двойной обмен значит, что частица A описывает полный круг вокруг частицы B . Также, если непрерывно изменим ее маршрут не приближая ее к частице B , то ожидаем получить тот же результат.

Итак, какие замкнутые маршруты можем получить непрерывно деформируя маршрут в евклидовом пространстве без точки (B)? Ответ для любого измерения больше 2 – любые. Следовательно двойной обмен должен давать такой же результат как и если бы просто оставили частицу на месте [NSS+08]. Это свойство задает ограничение на то, какими могут быть волновые функции.

У этого ограничения есть два решения: либо $\Psi(X_A, X_B) = \Psi(X_B, X_A)$ (симметричность), либо $\Psi(X_A, X_B) = -\Psi(X_B, X_A)$ (антисимметричность). У антисимметричных функций есть интересное свойство: если функция $f(x_1, x_2)$ антисимметрична, то $f(x_1 = X, x_2 = X) = 0$, то есть антисимметричная функция равна нулю, если ее аргументы одинаковы. В этом легко убедиться на примере $f = x_1 - x_2$.

Для антисимметричной волновой функции это означает, что две частицы не могут иметь полностью одинаковое состояние – волновая функция (и вероятность) такой конфигурации равна нулю.

В эксперименте это строго выполняется для всех частиц с полуцелым спином (фермионов), к которым относятся электроны, а частицы с целым спином (бозоны), например, фотоны, имеют симметричную волновую функцию и могут иметь одинаковые состояния.

Для квантовой химии это все имеет одно важное следствие: электроны – это фермионы и волновая функция, описывающая всю систему электронов целиком, должна быть антисимметрична относительно перестановки (замены пары индексов). Любое не антисимметричное решение не физично, поскольку допускает существование электронов в одинаковом состоянии, а такого не бывает. В химии этот вывод называется [принцип запрета Паули](#).

Чтобы сделать волновую функцию системы из двух электронов антисимметричной, используется такой прием:

$$\Psi(X_A, X_B) = \psi_A(X_A)\psi_B(X_B) - \psi_B(X_A)\psi_A(X_B)$$

Во-первых, легко проверить, что если поменять местами X_A и X_B , то вся функция просто изменит знак. Во-вторых, можно заметить, что формулу можно записать как определитель матрицы:

$$\Psi(X_A, X_B) = \begin{vmatrix} \psi_A(X_A) & \psi_B(X_A) \\ \psi_A(X_B) & \psi_B(X_B) \end{vmatrix}$$

Из курса линейной алгебры можно вспомнить, что определитель меняет знак при перестановке двух столбцов или двух строк – это свойство позволяет делать антисимметричные волновые функции систем из N волновых функций отдельных электронов, если использовать метод Хартри не для атома гелия, а для системы с большим числом электронов. Для этого составляется определитель $N \times N$ по аналогии с формулой выше: элемент в строке i , столбце j – это i -я волновая функция с параметрами j -го электрона в качестве аргумента. Волновая функция системы частиц вычисляется как нормированный определитель матрицы, а определитель всегда антисимметричен. Такой вариант сборки волновой функции системы частиц называется “определитель Слэтера”.

Так как весь подход является аппроксимацией, не любая система может быть точно представлена таким детерминантом, но он является очень распространенным методом “сборки” волновой функции системы электронов в квантовой химии. Его использование также немножко меняет вид одноэлектронных уравнений: чтобы корректно учесть антисимметрию, в эффективный потенциал добавляется так называемое “обменное взаимодействие”.

Все вместе составляет метод Хартри-Фока:

- итеративная процедура самосогласованного поля;
- усредненное действие электронов друг на друга, учет обменного взаимодействия;
- детерминант Слэтера.

57.2.3 SCF в psi4

Теперь можно вернуться к коду и взглянуть на него чуть более осмысленно. При вычислении энергии явно передаем, что хотим посчитать ее методом Self-Consistent Field:

```
e_in_ht = psi4.energy('scf')
```

Но что происходит в настройках, пока по-прежнему неясно:

```
psi4.set_options(
{
    'basis': 'STO-3G',
    'reference': 'rohf',
}
)
```

Начнем с параметра `basis`. `STO-3G` – не стандарт связи, а **Slater Type Orbital** с **3 Гауссианами** в базисном наборе, то есть базис на основе детерминанта Слэтера. В описании метода SCF мы собирались начинать итерации с водородоподобных волновых функций, но так как весь метод является аппроксимирующим, нам никто не мешает выбирать другие волновые функции, если результаты лучше согласуются с экспериментом. Выбор базиса может существенно влиять на результат вычислений и современные базисы сложнее, чем Слэтеровский детерминант – он просто один из первых и наиболее популярных.

Параметр `reference` означает, какие предположения о волновой функции делаем, в данном случае используется **Restricted Open Shell Hartree-Fock**, так как у атома водорода только один электрон и его оболочка не заполнена (на уровне энергии $n = 1$ для этого нужно 2 электрона).

Давайте повторим вычисления с более “современными” опциями.

```
psi4.core.clean()

h_atom = psi4.geometry("H")

psi4.set_options({
```

(continues on next page)

(continued from previous page)

```
'basis': 'd-aug-cc-pv5z', # разбор этого базиса выходит за рамки этого интро
'scf_type': 'pk',
'reference': 'rohf',
})

e_in_ht = psi4.energy('scf')

print(f"Better hydrogen ground state energy: {e_in_ev(e_in_ht)} eV")
```

Better hydrogen ground state energy: -13.605551648965163 eV

Та-дам! Используя более прокачанные базисы, получили правильный ответ.

Давайте посмотрим, что еще можно сделать с помощью self-consistent field.

57.2.4 Атом Гелия (численно)

Раз мы разобрали SCF на примере атома гелия, то наверняка можно посчитать его энергию в psi4.

```
psi4.core.clean()

he_atom = psi4.geometry("He")

psi4.set_options({
    'basis': 'STO-3G',
    'reference': 'rohf',
})

e_in_ht = psi4.energy('scf')

print(f"Helium ground state energy: {e_in_ev(e_in_ht)} eV")
```

Helium ground state energy: -76.403693763909 eV

Экспериментальное значение энергии атома гелия **равно -79.0 eV**.

57.2.5 Молекула водорода

Пока рассматривали только атомы, но SCF можно использовать и для молекул – потенциалы становятся сложнее, электронов больше, но общая логика не меняется.

```
psi4.core.clean()

# задаем 2 атома водорода с явными координатами
h_mol = psi4.geometry("""
H 0 0 0
H 0 0 0.74
""")

psi4.set_options({
    'basis': 'STO-3G',
    'reference': 'rohf',
})
```

(continues on next page)

(continued from previous page)

```
e_in_ht_h = psi4.energy('scf', molecule=h_mol)
print(f"Hydrogen ground state energy: {e_in_ev(e_in_ht_h)} eV")
```

```
Hydrogen ground state energy: -30.388568856869107 eV
```

Здесь задали явно координаты обоих атомов водорода в молекуле и энергия электронов была вычислена в предположении, что ядра водородов неподвижны. Здесь расстояние в 0.74 Ангстрема взято из [экспериментальных данных](#). Если бы задали неправильные координаты, то рассчитанная энергия окажется неверной. Точнее, она соответствовала бы нефизичной ситуации, когда неведомая сила “удерживает” ядра водорода на месте.

В `psi4` есть метод для оптимизации геометрии молекулы `psi4.optimize`; он не фиксирует положение ядер и возвращает минимальную возможную энергию с учетом вариации положения атомов.

Вычисления с оптимизацией геометрии занимают значительно больше времени.

```
psi4.core.clean()

h_mol_bad = psi4.geometry("""
H 0 0 0
H 0 0 1.5
""")
# 1.5 – неверное расстояние в ангстремах, верное – 0.74

psi4.set_options({
    'basis': 'STO-3G',
    'reference': 'rohf',
})

# рассчитываем энергию "в точке" с неправильной геометрией
e_in_ht_h_bad = psi4.energy('scf', molecule=h_mol_bad)

# рассчитываем энергию, оптимизируя по ходу геометрии
e_in_ht_h_optimized = psi4.optimize('scf', molecule=h_mol_bad)

print(f"Hydrogen molecule, incorrect ground state energy: {e_in_ev(e_in_ht_h_bad)} eV
")
print(f"Hydrogen molecule, optimized ground state energy: {e_in_ev(e_in_ht_h_
optimized)} eV")
```

```
Optimizer: Optimization complete!
Hydrogen molecule, incorrect ground state energy: -24.786132109551904 eV
Hydrogen molecule, optimized ground state energy: -30.40888422818135 eV
```

Для некорректной геометрии получилась завышенная энергия, а после оптимизации – почти что такая же энергия, как при вычислении с фиксированным расстоянием 0.74. В оптимальном состоянии энергия системы должна быть минимальна, так что результаты вполне разумны.

Подобный метод можно использовать и для поиска геометрии куда более сложных молекул.

57.2.6 Молекула этилового спирта

Молекула водорода – это все еще почти игрушечный пример. Давайте попробуем обсчитать молекулу этанола.

Задавать руками геометрию молекулы C_2H_5OH можно, но будет явно сложнее, чем для молекулы водорода. К счастью, это необязательно: `psi4` умеет скачивать геометрию из базы данных PubChem по номенклатурному имени либо уникальному ChemId.

```
psi4.core.clean()

eth = psi4.geometry("pubchem:ethanol")

psi4.set_options({
    'basis': 'STO-3G',
    'reference': 'rohf',
})

e_in_ht_eth = psi4.energy('scf', molecule=eth)

print(f"Ethanol ground state energy: {e_in_ev(e_in_ht_eth)} eV")
```

Searching PubChem database for ethanol (single best match returned)

Found 1 result(s)

Ethanol ground state energy: -4139.645908265889 eV

57.3 Что мы узнали?

Разобрались с базовой теорией квантовой химии:

- как записать уравнение Шредингера для атома;
- какое получается аналитическое решение для атома водорода;
- как устроен метод Self-Consistent Field для вычисления волновых функций и энергии для задачи многих тел;
- как использовать SCF в python пакете `psi4`.

В примерах везде вычисляли ground state энергию, но, конечно, зная волновые функции, можно посчитать много чего еще. Например, можно вычислить спектр поглощения и энергию ионизации (энергии возбужденных состояний), моделировать взаимодействие молекул (найти равновесное состояние для двух систем), с помощью плагинов можно смоделировать рассеяние рентгена на молекуле... Квантовая механика постулирует, что любая измеримая величина является усреднением определенного оператора по волновой функции, поэтому в теории любую физическую величину можно посчитать, зная волновую функцию. Возможности ограничены в основном вычислительной сложностью, а не теорией.

Для более глубокого погружения в практику квантовой химии можно пройти лабораторные работы `psi4`: [раз](#), [два](#).

CHAPTER
FIFTYEIGHT

МОДЕЛЬ ИЗИНГА

В этой лекции познакомимся с моделью Изинга, которая изначально была разработана для описания магнетизма, но оказалась настолько удачной и универсальной, что сегодня к решению именно этой задачи стараются свести многие проблемы реального мира, причем не только из физики. В следующем блоке подробно покажем, как к гамильтонианам типа Изинга, или, по-другому, “спиновым стеклам” могут быть сведены задачи комбинаторной оптимизации и квантовой химии. Так что знакомство с этой удивительной моделью, а также описывающим ее гамильтонианом нам просто необходимо!

Note: Специальные квантовые компьютеры компании D-Wave сконструированы так, что они могут решать вообще только одну задачу – нахождения основного состояния гамильтонианов типа Изинга. Но эта задача настолько распространена и важна, что эти компьютеры стали первыми в мире коммерческими квантовыми компьютерами! Кстати, далее этим компьютерам у нас посвящена [отдельная лекция](#).

Ближайшее время посвятим довольно много времени объяснению этой модели. Это может показаться скучным и занудным, но это важно для понимания того, как это все работает и как решать с помощью вариационных квантовых алгоритмов реальные задачи!

58.1 Задача Изинга в одномерном случае

Note: Ниже попробуем на пальцах объяснить модель Изинга. Пробовать будем через цепочку атомов антиферромагнетика во внешнем магнитном поле. Если вы плохо помните физику и вам это объяснение покажется сложным, то не расстраивайтесь – дальше также объясним задачу Изинга как задачу о поиске максимального разреза в графе – известную задачу комбинаторной оптимизации.

Пусть у нас есть, например, цепочка атомов, которые обладают магнитным моментом. Например, цепочка атомов антиферромагнетика. И мы прикладываем к этой цепочке внешнее магнитное поле.

Тогда, если поле маленькое, наши атомы будут стараться выстроиться в антиферромагнитный порядок, когда соседние из них имеют моменты, направленные в разные стороны. Но если поле уже большое, то оно будет стремиться “поворнуть” моменты по своему направлению. А если еще вспомнить, что магнитный момент атома является квантовой величиной и может быть в суперпозиции состояний в одну сторону и в противоположную, то не очень маленькое, но и не слишком большое поле будет переводить часть атомов именно в такие суперпозиции.

Reminder о квантовой физике

В квантовой механике есть фундаментальное уравнение, которое описывает динамику квантовых систем. Оно называется уравнением Шредингера: $i\hbar \frac{\partial \Psi}{\partial t} = \hat{H}\Psi$, где \hat{H} – это оператор Гамильтона, или гамильтониан. Также его называют оператором полной энергии системы, так как в общем случае он равен сумме операторов

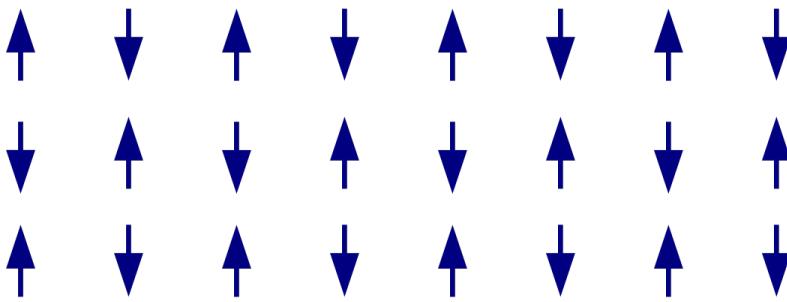


Fig. 58.1: Иллюстрация антиферромагнитного порядка

кинетической и потенциальной энергии. Начиная с этой лекции будем очень часто обращаться к этому оператору, но в целом в нем нет ничего принципиально сложного. Это такой же эрмитов оператор, как и другие. А наблюдаемая величина, которую получаем при [измерении](#) этого оператора – это энергия системы.

Давайте теперь запишем гамильтониан такой системы. Для представления магнитных моментов будем использовать оператор σ^z – другими словами, спин в направлении оси Z . Если кто-то забыл, как выглядит оператор σ^z , то рекомендуем еще раз просмотреть раздел про [операторы Паули](#) первой лекции. Далее будем очень активно использовать эти матрицы для представления задач реального мира!

Для начала, в случае если внешнего поля нет, мы должны записать взаимодействие соседних атомов. Так как у нас антиферромагнетик, минимальная энергия достигается в случае, если каждый спин направлен с соседними. Это просто оператор $\sigma_j^z \sigma_{j+1}^z$, который действует на все пары соседних спинов. Ну и сразу введем некоторую константу обменного взаимодействия J , чтобы потом нам было удобно сравнивать ее с внешним полем. В итоге, для цепочки из N спинов, получаем:

$$\hat{H}_{h=0} = J \sum_{i=0}^{N-1} \sigma_i^z \sigma_{i+1}^z$$

А теперь давайте добавим внешнее поле h . В этом случае поле просто действует на все спины и пытается выстроить их в зависимости от своего направления, например, вниз. Тогда полный гамильтониан такой системы можно записать в виде:

$$\hat{H}_{h \neq 0} = J \sum_{i=0}^{N-1} \hat{\sigma}_i^z \hat{\sigma}_{i+1}^z - h \sum_{i=0}^N \hat{\sigma}_i^z$$

58.2 Задача Изинга как задача о максимальном разрезе в графе

Задача о максимальном разрезе в графе – это очень известная задача комбинаторики. Она относится к классу NP -трудных, и к ней можно свести все другие NP задачи. При этом ее формулировка одна из самых простых среди всего класса задач. Формулируется она следующим образом.

Нам дан граф – набор вершин V и связывающих их ребер E . Нам надо найти такое разделение вершин V на два непересекающихся набора V_1, V_2 , что число ребер между вершинами из разных наборов будет максимально.

Теперь давайте представим, что каждой вершине нашего графа сопоставили кубит. Для этих кубитов можем производить измерения по оси Z , чтобы понять, как направлен тот или иной спин. И давайте запишем вот такой

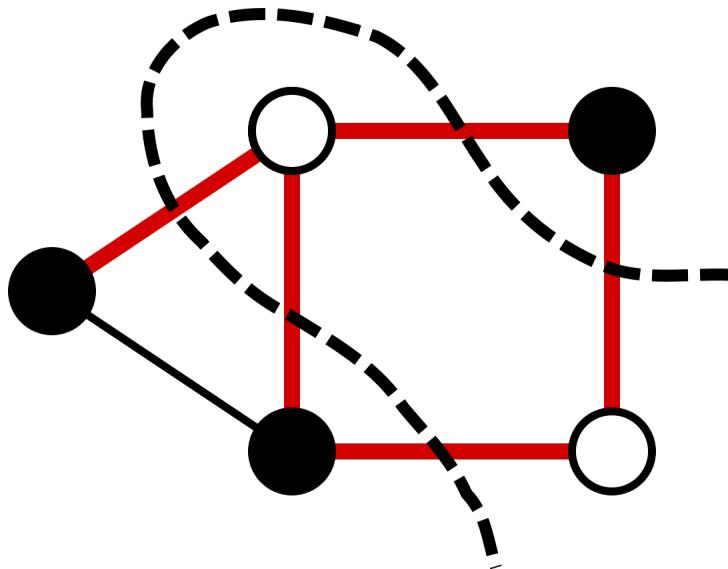


Fig. 58.2: Иллюстрация задачи о максимальном разрезе в графе

гамильтониан и внимательно на него посмотрим:

$$\hat{H} = \sum_{u,v \in E} \hat{\sigma}_u^z \hat{\sigma}_v^z$$

Тут суммирование $u, v \in E$ идет по всем ребрам графа, а u, v – вершины инцидентные ребрам. Если вспомнить, что собственные значения σ^z это ± 1 для, соответственно, спина “вверх” и спина “вниз”, то не трудно понять, в каком случае у нас будет минимум энергии этого гамильтониана. А будет он тогда, когда максимальное число пар вершин u, v имеют разную ориентацию своих спинов. Ведь если они имеют одинаковую направленность (причем не важно, $+1$ или -1), их произведение будет равно 1, но если направленность разная, то их произведение даст нам -1 . Таким образом, минимум энергии такого гамильтониана достигается тогда, когда мы разбили наши вершины на две группы – спин “вверх” и спин “вниз” – причем число ребер между этими группами максимальное. А это в чистом виде формулировка задачи о максимальном разрезе в графе!

Note: Тематика квантовой физики мало обсуждалась в первых лекциях, но нам пока достаточно знать лишь то, что для любой физической системы (включая квантовую) стремится в состояние с минимальной энергией. Например, тело, подброшенное вверх, стремится упасть на землю, а возбужденный атом стремится релаксировать в невозбужденное состояние.

При этом из квантовой физики помним, что для реальных физических систем наиболее вероятными являются состояния с минимальной энергией и системы стремятся в эти состояния прийти. Теперь для простоты предположим, что наш график – это просто цепочка, то есть ребра есть лишь между соседними в одномерном пространстве вершинами. Ну и теперь давайте сформулируем нашу задачу о максимальном разрезе чуточку сложнее – нам надо найти не просто максимальный разрез, а такой разрез, который самый большой при наименьшем числе вершин в наборе V_1 . И поскольку теперь у нас два вклада в стоимость, то нам нужны коэффициенты, которые покажут, что важнее. Пусть это будут J и h . Тогда гамильтониан соответствующей модели Изинга можно записать так:

$$\hat{H} = J \sum_{i=0}^{N-1} \hat{\sigma}_i^z \hat{\sigma}_{i+1}^z - h \sum_{i=0}^N \hat{\sigma}_i^z$$

Как видно, это тот же самый гамильтониан, который получили и для моделирования антиферромагнетиков. То есть задача об основном состоянии цепочки антиферромагнитных частиц во внешнем поле эквивалентна задаче о максимальном разрезе в графе-цепочке при некотором штрафе за одно из выделенных направлений спинов. Эквивалентность в данном случае значит, что:

- решив задачу о максимальном разрезе, можно найти и основное состояние физической системы;
- как-то смоделировав физическую систему, подождав пока она релаксирует, после чего измерив ее, получим конфигурацию, отвечающую решению задачи о максимальном разрезе.

Note: Одномерная цепочка атомов, или поиск максимального разреза в графе-цепочке, является простым случаем и не является NP -задачей. Однако уже в двумерном случае эта задача становится сильно сложнее, как и, например, если в цепочке атомов ферромагнетика добавим взаимодействие не только соседних спинов, но и взаимодействие с соседями соседа. Аналогично, модель вида Изинга сильно усложняется при добавлении недиагональных (off-diagonal elements) элементов гамильтониана, например, когда внешнее поле направлено в другом направлении и второй член гамильтониана принимает вид $h \sum_{i=N} \sigma_i^x$. Более подробное исследование данной модели приводится в [этой продвинутой лекции](#).

58.3 Модель Изинга на чистом NumPy

Давайте попробуем реализовать одномерный гамильтониан Изинга на чистом NumPy/SciPy в виде разреженной матрицы. Для этого вспомним, что действуя оператором σ_z на i -й кубит, одновременно действуем единичным оператором на все остальные, а потом перемножаем все операторы произведением Кронекера. Из [лекций по линейной алгебре](#) помним также об ассоциативности произведения Кронекера, чем и воспользуемся:

```
import numpy as np
from scipy import sparse
from scipy.sparse import linalg as sl

def sigmaz_k(k: int, n: int) -> (sparse.csr_matrix):
    left_part = sparse.eye(2 ** k)
    right_part = sparse.eye(2 ** (n - 1 - k))

    return sparse.kron(
        sparse.kron(
            left_part,
            sparse.csr_matrix(np.array([[1, 0], [0, -1], ])))
        ),
        right_part
    )
```

А теперь можем реализовать и сам оператор Изинга:

```
def ising(j: float, h: float, n: int) -> (sparse.csr_matrix):
    res = sparse.csr_matrix((2 ** n, 2 ** n), dtype=np.complex64)

    for i in range(n - 1):
        res += j * sigmaz_k(i, n) * sigmaz_k(i + 1, n)
        res -= h * sigmaz_k(i, n)

    res -= h * sigmaz_k(n - 1, n)

    return res
```

Если внешнего поля нет, спины выстраиваются в полный антиферромагнитный порядок, в чем легко убедиться. Создадим оператор для такой модели и, например, 10 спинов (или 10 вершин в графе, если говорим в терминах Max-Cut):

```
op = ising(1, 0, 10)
solution = sl.eigs(op, which="SR", k=1, return_eigenvectors=True)
print(f"Energy: {solution[0][0]}")
```

```
Energy: (-9.000000000000018-2.208468631860285e-16j)
```

Note: Тут пользуемся функциями из ARPACK – набором рутин для линейной алгебры разреженных систем. Более подробно о способах и алгоритмах классических решений задачи о собственных значениях расскажем в *одной из следующих лекций*, полностью посвященной этой теме. Пока же просто используем эту рутину как “черный ящик”. Более подробное описание этой функции и ее аргументов можно посмотреть в [документации библиотеки SciPy](#).

Эта энергия соответствует антиферромагнитному порядку, в этом легко убедиться, нарисовав спины и формулу на бумажке. Внимательный читатель заметил, что в этот раз вернули также и первый собственный вектор, который в нашем случае является волновой функцией основного состояния. А как знаем, квадраты элементов вектора волновой функции дают нам вероятности соответствующих битовых строк (если для вас это все звучит дико, то очень рекомендуем вернуться к [лекции про кубит](#)). Давайте посмотрим на эту битовую строку, иначе на порядок наших спинов в решении (или на разбиение вершин графа на два подмножества в терминах Max-Cut):

```
def probs2bit_str(probs: np.array) -> (str):
    size = int(np.log2(probs.shape[0]))
    bit_s_num = np.where(probs == probs.max())[0][0]

    s = f"{bit_s_num:b}"
    s = "0" * (size - len(s)) + s

    return s

probs = solution[1] * solution[1].conj()
print(probs2bit_str(probs))
```

```
0101010101
```

Теперь давайте попробуем добавить внешнее поле с коэффициентом, равным удвоенному значению константы обменного взаимодействия. В терминах комбинаторной задачи, добавляем штраф, равный 2 умножить на число спинов, направленных вверх.

```
def external_field(j: float, h: float, n: int) -> (None):
    op = ising(j, h, n)
    solution = sl.eigs(op, which="SR", k=1, return_eigenvectors=True)
    print(f"Energy: {solution[0][0]}")

    probs = solution[1] * solution[1].conj()
    print(probs2bit_str(probs))

external_field(1, 2, 10)
```

```
Energy: (-10.99999999999999-1.0571225056728771e-16j)
0101010010
```

Видим, что теперь наш антиферромагнитный порядок уже не полный. В целом, данная модель довольно интересная, так как при некотором отношении $\frac{h}{J}$ у нас происходит фазовый переход от полной упорядоченности, а при дальнейшем росте h приходим к одинаковой ориентации всех спинов, в чем легко убедиться, взяв, например, $h = 100$:

```
external_field(1, 100, 10)
```

```
Energy: (-991.0000000000039-3.4882612571137096e-14j)
0000000000
```

58.4 Заключение

В этой лекции на базовом уровне познакомились с моделью Изинга – очень важным концептом в квантовом машинном обучении. Узнали, что:

- модель Изинга изначально была создана для объяснения магнетизма;
- нахождение решений для модели Изинга в общем случае – *NP-полная* задача;
- модель Изинга также может быть сформулирована в терминах задачи о максимальном разрезе в графе (и наоборот);
- в классической модели Изинга существуют интересные фазовые переходы;
- модель Изинга легко реализовать в коде, используя SciPy, но размерность задачи растет очень быстро.

ТОЧНОЕ АНАЛИТИЧЕСКОЕ РЕШЕНИЕ ОДНОМЕРНОЙ МОДЕЛИ ИЗИНГА. МЕТОД ТРАНСФЕР-МАТРИЦЫ.

В этой лекции более подробно обсудим модель Изинга, *рассмотренную ранее*. Чуть глубже погрузимся в физику этой модели, а также рассмотрим ее частные случаи, которые имеют точные аналитические решения. Для этого также познакомимся с методом *трансфер-матрицы*.

59.1 Модель Изинга

Модель Изинга представляет собой широко распространенную математическую модель статистической физики, которая может применяться в различных областях человеческой деятельности.

Note: Несмотря на свою известность, многие не знают, что модель названа в честь Эрнеста Изинга – автора работы, опубликованной в 1925 г. [Izi25], в которой он впервые представил данную модель и которая стала его первой и по сути *единственной* публикацией.



Fig. 59.1: Эрнст Изинг, 1900-1998

Суть самой модели можно постичь, например, при стремлении описать свойства намагничивания материала.

Представим себе, что имеется магнитная решетка (она может быть одномерной, двумерной или трехмерной) каждой вершине которой сопоставляется число называемое *спином* (σ) и принимающее значение $+1$ или -1 (соответственно поле « \uparrow » или поле « \downarrow »). Каждому из 2^N (где N – число вершин решетки) возможных вариантов расположения спинов (или намагниченности) приписывается энергия, величина которой определяется условиями попарного взаимодействия спинов соседних вершин. Например, энергия одномерной модели Изинга, в которой спины взаимодействуют только с ближайшими соседями (посредством параметра обменного взаимодействия J) и с однородным внешним полем (H), имеет вид:

$$E(\{\sigma\}) = -J \sum_{i=1}^N \sigma_i \sigma_{i+1} - H \sum_{i=1}^N \sigma_i. \quad (59.1)$$

Это выражение соответствует периодическим граничным условиям, при которых первый и $N + 1$ -й спины отождествляются, то есть

$$\sigma_{N+1} \equiv \sigma_1.$$

В этом случае можно считать, что система спинов находится на окружности, то есть первый и N -й спины являются ближайшими соседями и взаимодействуют между собой. Для описания такой системы необходимо найти статистическую сумму Z , в которой содержится полная информация о всех состояниях системы. Тогда другие характеристики системы, такие как удельная энергия, свободная энергия, энтропия, намагниченность, магнитная восприимчивость и др. вычисляются через статистическую сумму или ее производные.

Статистическая сумма рассматриваемой модели при температуре θ есть сумма по всем спиновым конфигурациям $\{\sigma\}$:

$$Z(\theta, H, N) = \sum_{\{\sigma\}} e^{-\frac{E(\{\sigma\})}{\theta}}. \quad (59.2)$$

Сумму по состояниям $\{\sigma\}$ можно понимать как последовательное суммирование по значениям всех переменных σ_i :

$$\sum_{\{\sigma\}} \equiv \sum_{\sigma_1=\pm 1} \sum_{\sigma_2=\pm 1} \dots \sum_{\sigma_N=\pm 1}.$$

Следует заметить, что количество членов статистической суммы экспоненциально растет с увеличением числа вершин, соответственно увеличивается и время расчета термодинамических параметров для большого количества частиц.

59.2 Матричный подход

Для аналитического решения модели Изинга как правило применяется матричный подход. В рамках такого подхода статистическая сумма многих решеточных моделей Изинга может быть записана в виде следа некоторой степени *трансфер-матрицы*. Чтобы выразить статистическую сумму (59.2) через трансфер-матрицу, сначала представим гиббсовскую экспоненту в виде произведения N множителей

$$e^{-\frac{E(\{\sigma\})}{\theta}} = T_{\sigma_1 \sigma_2} T_{\sigma_2 \sigma_3} \dots T_{\sigma_{N-1} \sigma_N} T_{\sigma_N \sigma_1} = \prod_{j=1}^N T_{\sigma_j, \sigma_{j+1}}, \quad (59.3)$$

где

$$T_{\sigma \sigma'} = \langle \sigma | T | \sigma' \rangle = e^{t \sigma \sigma' + \frac{h}{2} (\sigma + \sigma')}, \quad (59.4)$$

а t и h введены для более компактного написания формул и выражаются через J , H и θ следующим образом

$$t = \frac{J}{\theta}, \quad h = \frac{H}{\theta}.$$

Числа $T_{\sigma\sigma'}$ (59.4) будем дальше рассматривать как матричные элементы матрицы T , строки и столбцы которой вместо натуральных чисел нумеруются изинговскими переменными σ и σ' . Так как изинговские переменные принимают два значения: $\sigma, \sigma' = \pm 1$, то матрица T является квадратной матрицей размера 2×2 . Договоримся дальше считать, что значение $\sigma = 1$ ($\sigma' = 1$) соответствует первой строке (первому столбцу), а значение $\sigma = -1$ ($\sigma' = -1$) соответствует второй строке (второму столбцу). Матрица T с элементами (59.4) в таком случае имеет следующий явный вид:

$$T = \begin{vmatrix} T_{1,1} & T_{1,-1} \\ T_{-1,1} & T_{-1,-1} \end{vmatrix} = \begin{vmatrix} e^{t+h} & e^{-t} \\ e^{-t} & e^{t-h} \end{vmatrix} \quad (59.5)$$

Матрица T называется *трансфер-матрицей Крамерса–Ваннье*.

Равенство (59.3) позволяет записать статистическую сумму (59.2) в следующем виде

$$Z(\theta, H, N) = \sum_{\sigma_1=\pm 1} \sum_{\sigma_2=\pm 1} \dots \sum_{\sigma_{N-1}=\pm 1} \sum_{\sigma_N=\pm 1} T_{\sigma_1\sigma_2} T_{\sigma_2\sigma_3} \dots T_{\sigma_{N-1}\sigma_N} T_{\sigma_N\sigma_1}. \quad (59.6)$$

Структура слагаемых позволяет переписать сумму в правой части последнего равенства следующим образом:

$$\begin{aligned} & \sum_{\sigma_1=\pm 1} \sum_{\sigma_2=\pm 1} \dots \sum_{\sigma_{N-1}=\pm 1} \sum_{\sigma_N=\pm 1} T_{\sigma_1\sigma_2} T_{\sigma_2\sigma_3} \dots T_{\sigma_{N-1}\sigma_N} T_{\sigma_N\sigma_1} = \\ &= \sum_{\sigma_1=\pm 1} \sum_{\sigma_2=\pm 1} \dots \sum_{\sigma_{N-1}=\pm 1} T_{\sigma_1\sigma_2} T_{\sigma_2\sigma_3} \dots T_{\sigma_{N-2}\sigma_{N-1}} \sum_{\sigma_N=\pm 1} T_{\sigma_{N-1}\sigma_N} T_{\sigma_N\sigma_1} \end{aligned} \quad (59.7)$$

Теперь, поскольку $T_{\sigma\sigma'}$ являются матричными элементами матрицы T , заметим, что сумма

$$\sum_{\sigma_N=\pm 1} T_{\sigma_{N-1}\sigma_N} T_{\sigma_N\sigma_1}$$

равна $T_{\sigma_{N-1}\sigma_1}^2$, то есть матричному элементу матрицы T^2 , которая получается в результате (матричного) умножения двух матриц T (является квадратом матрицы T). Тогда сумма в правой части равенства (59.7) равна

$$\sum_{\sigma_1=\pm 1} \sum_{\sigma_2=\pm 1} \dots \sum_{\sigma_{N-1}} T_{\sigma_1\sigma_2} T_{\sigma_2\sigma_3} \dots T_{\sigma_{N-2}\sigma_{N-1}} T_{\sigma_{N-1}\sigma_1}^2. \quad (59.8)$$

Дальше используем равенство

$$\sum_{\sigma_{N-1}} T_{\sigma_{N-2}\sigma_{N-1}} T_{\sigma_{N-1}\sigma_1}^2 = T_{\sigma_{N-2}\sigma_1}^3,$$

где $T^3 = T \cdot T^2$ – куб матрицы T , и перепишем выражение (59.8) в виде

$$\sum_{\sigma_1=\pm 1} \sum_{\sigma_2=\pm 1} \dots \sum_{\sigma_{N-2}} T_{\sigma_1\sigma_2} T_{\sigma_2\sigma_3} \dots T_{\sigma_{N-3}\sigma_{N-2}} T_{\sigma_{N-2}\sigma_1}^3.$$

Продолжим этот процесс и просуммируем последовательно по всем σ_i , $i = 2, \dots, N$, кроме σ_1 , тогда равенство (59.6) преобразуется к виду

$$Z(\theta, H, N) = \sum_{\sigma_1=\pm 1} T_{\sigma_1\sigma_1}^N$$

В последнем равенстве суммируются диагональные элементы матрицы T^N , которая есть матрица T в степени N . Сумма диагональных элементов матрицы – это след данной матрицы. Таким образом, окончательно получаем для статистической суммы одномерной модели Изинга с взаимодействием ближайших соседей и внешним полем, при периодических граничных условиях, следующую формулу:

$$Z(\theta, H, N) = \text{Tr } T^N. \quad (59.9)$$

Отметим несколько обстоятельств, связанных с полученным выражением для статистической суммы:

1. формула (59.9) является следствием трансляционной инвариантности рассматриваемой системы. Трансляционная инвариантность в данном случае означает, что энергия системы спинов не изменяется при пространственном сдвиге, то есть спиновые конфигурации $\{\sigma\}$ и $\{\sigma'\}$, где $\sigma'_i = \sigma_{i+1}$, имеют одинаковые энергии;
2. представление гиббсовской экспоненты в виде произведения (59.3), где множители $T_{\sigma\sigma'}$ имеют вид (59.4), не является единственным возможным. Можно выбрать трансфер-матрицу по-другому. Например, если взять $\tilde{T}_{\sigma\sigma'} = e^{t\sigma\sigma' + h\sigma}$, то тогда: $e^{-\frac{E(\{\sigma\})}{\theta}} = \prod_{j=1}^N \tilde{T}_{\sigma_j\sigma_{j+1}}$. Таким образом, в выборе трансфер-матрицы существует некоторый произвол. Значение статистической суммы не зависит от этого произвола, поэтому им естественно распорядиться исходя из каких-то дополнительных соображений, например так, чтобы матрица обладала какой-то симметрией, что упрощает вычисления. Для рассмотренной выше одномерной модели Изинга трансфер-матрица была выбрана вещественной симметричной (то есть эрмитовой);
3. если выбранная нами трансфер-матрица T является эрмитовой, то с помощью унитарного преобразования U ее можно преобразовать к диагональной форме:

$$\dot{T} = U^{-1}TU = dgn\{\lambda_1, \lambda_2, \dots, \lambda_r\},$$

где символом dgn обозначена диагональная матрица, на главной диагонали которой расположены ее собственные значения. В общем случае размерность такой матрицы равна величине $r = s^n$, где s – число состояний спина в вершине, а n – число обменных взаимодействий спинов цепочки (в рассматриваемой нами одномерной модели Изинга $s = 2$ и $n = 1$). В этом случае статистическая сумма (59.9) переписывается к следующему виду:

$$Z(\theta, H, N) = \text{Tr } \dot{T}^N = \sum_i^r \lambda_i^N, \quad (59.10)$$

При этом величина статистической суммы не меняется, поскольку след матрицы является ее инвариантом. Отметим, что процедуру диагонализации трансфер-матрицы можно провести множеством способов. В зависимости от подбора диагонализирующей матрицы U её можно провести таким образом, что на первом месте в матрице \dot{T} будет стоять её главное (единственное максимальное вещественное) собственное значение, которое всегда существует по теореме Фробениуса–Перрона. Если на первое место в спектре трансфер-матрицы поставить ее главное собственное значение, то статистическую сумму (59.10) можно переписать как:

$$Z(\theta, H, N) = \lambda_1^N \left[1 + \left(\frac{\lambda_2}{\lambda_1} \right)^N + \dots + \left(\frac{\lambda_r}{\lambda_1} \right)^N \right] \quad (59.11)$$

Тогда при переходе к термодинамическому пределу ($N \rightarrow \infty$) в выражении (59.11) все слагаемые, кроме первого, стремятся к нулю,

$$\lim_{N \rightarrow \infty} \left(\frac{\lambda_i}{\lambda_1} \right)^N = 0, \quad i = 2, 3, \dots, r \quad (59.12)$$

и выражение для статистической суммы преобразуется к виду $Z = \lambda_1^N$.

В результате все термодинамические и магнитные функции системы – свободная энергия системы, приходящаяся на один спин,

$$F = -\frac{\theta}{N} \ln Z = -\theta \ln \lambda_1, \quad (59.13)$$

энтропия

$$S = -\frac{\partial F}{\partial \theta} = \ln \lambda_1 + \frac{\theta}{\lambda_1} \frac{\partial \lambda_1}{\partial \theta}, \quad (59.14)$$

теплоемкость

$$C = -\theta \frac{\partial^2 F}{\partial \theta^2} = 2 \frac{\theta}{\lambda_1} \frac{\partial \lambda_1}{\partial \theta} + \frac{\theta^2}{\lambda_1} \frac{\partial^2 \lambda_1}{\partial \theta^2} - \frac{\theta^2}{\lambda_1^2} \left(\frac{\partial \lambda_1}{\partial \theta} \right)^2, \quad (59.15)$$

намагниченность

$$M = -\frac{\partial F}{\partial H} = \frac{\theta}{\lambda_1} \frac{\partial \lambda_1}{\partial H}, \quad (59.16)$$

и магнитная восприимчивость

$$\chi = -\frac{\partial^2 F}{\partial H^2} = \frac{\partial M}{\partial H} = -\frac{\theta}{\lambda_1^2} \left(\frac{\partial \lambda_1}{\partial H} \right)^2 + \frac{\theta}{\lambda_1} \frac{\partial^2 \lambda_1}{\partial H^2} \quad (59.17)$$

определяются только через главное собственное значение трансфер-матрицы.

59.3 Решение одномерной модели

Теперь вернемся к аналитическому решению рассматриваемой нами классической одномерной модели Изинга ($s = 2$ и $n = 1$). Как показали выше, задача сводится к сведению трансфер-матрицы (59.5) диагональному виду. Для этого воспользуемся правилами линейной алгебры и решим характеристическое уравнение для трансфер-матрицы и найдем её собственные значения:

$$\det(T - \lambda I) = \det \left(T - \lambda \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right) = 0, \quad (59.18)$$

где I – единичная матрица, λ – действительная переменная. Раскрывая определитель (59.18), находим искомое характеристическое уравнение:

$$\lambda^2 - \lambda(e^{t+h} + e^{t-h}) + e^{2t} - e^{-2t} = 0 \quad (59.19)$$

решением которого являются два собственных значения:

$$\lambda_{\pm} = \frac{1}{2} e^t (e^h + e^{-h}) \pm \frac{1}{2} \sqrt{e^{2t} (e^h + e^{-h})^2 - 4e^{2t} + 4e^{-2t}} = e^t ch(h) \pm \sqrt{e^{2t} sh^2(h) + e^{-2t}}, \quad (59.20)$$

где $ch(h) = \frac{e^h + e^{-h}}{2}$ – гиперболический косинус и $sh(h) = \frac{e^h - e^{-h}}{2}$ – гиперболический синус. Следовательно статистическая сумма рассматриваемой модели записывается в следующем виде:

$$Z = Tr T^N = \lambda_+^N + \lambda_-^N.$$

Здесь следует отметить, что для всех физических значений параметров t и h ($\in \mathbb{R}, < \infty$) выполняется условие: $|\lambda_+| > |\lambda_-|$. Тогда в термодинамическом пределе ($N \rightarrow \infty$) значение λ_+ преобладает над λ_- и удовлетворяет (59.12), а статистическая сумма преобразуется к виду: $Z = \lambda_+^N + \lambda_-^N \sim \lambda_+^N$. Отсюда можно получить выражение для удельной свободной энергии системы, приходящейся на один спин (59.13) (с точностью до множителя $(-\theta)$):

$$f(t, h) = \lim_{N \rightarrow \infty} \frac{\ln Z}{N} = \ln \lambda_+ = t + \ln \left(ch(h) + \sqrt{sh^2(h) + e^{-4t}} \right)$$

и для намагниченности (59.16):

$$m = -\frac{\partial f(t, h)}{\partial h} = \frac{sh(h) + \frac{sh(h)ch(h)}{\sqrt{sh^2(h)+e^{-4t}}}}{ch(h) + \sqrt{sh^2(h) + e^{-4t}}} = \frac{sh(h)}{\sqrt{sh^2(h) + e^{-4t}}},$$

которая является аналитической функцией h и исчезает при $h \rightarrow 0$ (так как $sh(h) \rightarrow 0$). Таким образом рассматриваемая модель не допускает фазового перехода при любой положительной температуре ($\theta > 0$). Она претерпевает *paramagnитный-ферромагнитный* фазовый переход только при $\theta = 0$, в этом случае спонтанная намагниченность равна $|m| = 1$.

59.4 Заключение

В этой лекции ознакомились с аналитическим решением одномерной модели Изинга методом *трансфер-матрицы*. В заключении хотелось бы сказать, что модель Изинга поддается многим обобщениям, будь то более высокие измерения, различные решетки или модифицированные взаимодействия.

ЗАДАЧИ КОМБИНАТОРНОЙ ОПТИМИЗАЦИИ

В обзоре квантовых алгоритмов мы уже касались темы задач, для которых не существует эффективных классических алгоритмов, а сегодня разберем еще больше примеров. В конце кратко обсудим пару способов, как решать такие задачи на классических компьютерах, а также в чем их принципиальные проблемы и ограничения.

Лекция будет построена так:

- задача о максимальном разрезе в графе;
- задача о выделении сообществ в графе;
- задача о Гамильтоновых циклах и коммивояжере;
- задача о рюкзаке;
- жадные алгоритмы;
- метод имитации отжига.

Какого-то сложного кода или больших формул с дираковскими значками из квантовой механики тут не будет, так что можно немного передохнуть, расслабиться и насладиться чтением. Для тех, кто хорошо знаком с задачами целочисленной оптимизации, в этой лекции точно не будет ничего нового, и ее можно будет пропускать, ну или пролистать по диагонали.

60.1 Задача о максимальном разрезе в графе

Мы уже немного говорили об этой задаче в лекции про модель Изинга из-за их очень большого сходства. Но давайте еще раз вспомним, что это за модель такая. Ну и сразу рассмотрим ее максимально общий случай. Итак, у нас есть граф на множестве вершин V , связанных множеством ребер E . Каждое ребро имеет две инцидентных вершины u, v ; в общем случае порядок u, v важен, тогда мы говорим о направленном (*directed*) графе. Каждому ребру можно также сопоставить действительное число w , тогда у нас будет так называемый *взвешенный* граф. Наша цель – разбить множество вершин V на два непересекающихся сообщества V_1, V_2 . Давайте сформулируем функцию стоимости:

$$C = \sum_{u,v,w \in E} w(\mathbf{1}(u \in V_1, v \in V_2) + \mathbf{1}(u \in V_2, v \in V_1))$$

То есть в общем случае это просто сумма всех весов ребер между двумя сообществами. В общем случае эта задача является NP -полной. В теории к этой задаче можно также свести любую другую NP задачу за полиномиальное время.

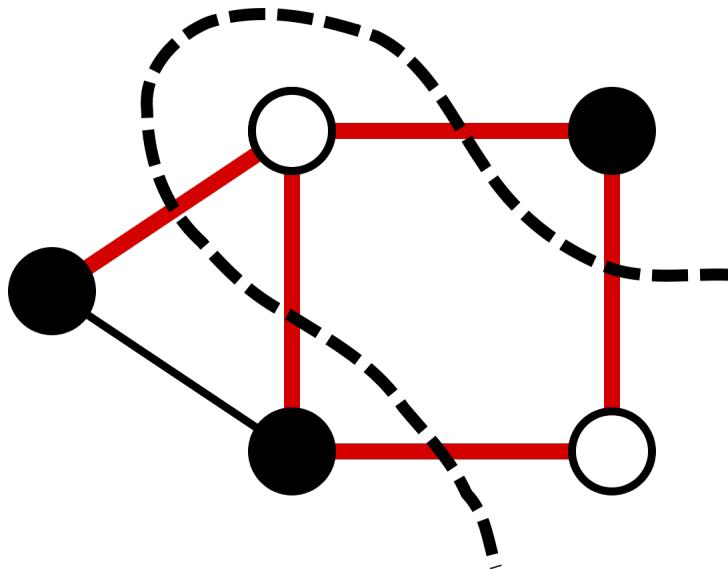


Fig. 60.1: Иллюстрация задачи о максимальном разрезе в графе

60.2 Задача о выделении сообществ в графах

Задача о выделении сообществ в графах это уже более практическая и понятная задача. Она находит применение во многих областях, но одно из самых очевидных применений – это социология (в том числе анализ социальных сетей), когда мы хотим, анализируя контакты людей, выделить из них сообщества для дальнейшего анализа. Эта задача также является NP -трудной, так как существует экспоненциально много способов разбить вершины на множества и при этом для общего случая не существует полиномиального алгоритма решения. Ну и даже если мы найдем какое-то решение, то проверка того, что оно лучшее возможна также лишь за экспоненциальное время. Хотя для этой задачи известны относительно быстрые приближенные алгоритмы, нам очень трудно понять, насколько хорошее решение они дают для действительно больших графов.

Note: Для работы с графиками мы будем пользоваться библиотекой NetworkX. Она написана на чистом Python и плохо подходит для работы с большими графиками, зато имеет простой интерфейс и легко устанавливается на любую систему. Ее можно установить из репозитория PyPI, используя команду

```
pip install networkx
```

Подробнее о пакетах Python можно почитать в [одной из вводных лекций](#) про этот язык программирования.

Одним из первых известных наборов данных для задачи выделения сообществ является “Клуб каратэ Захари” (Zachary’s Karate Club) [Zac77]. Для этого набора данных точно известно, к какому из двух сообществ принадлежит каждая из вершин. В этом клубе карате был внутренний конфликт, и одна часть людей была в группе одного из инструкторов (Mr. Hi), а другая – в группе администратора (Officer).

```
import networkx as nx
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'

zachary = nx.generators.social.karate_club_graph()
```

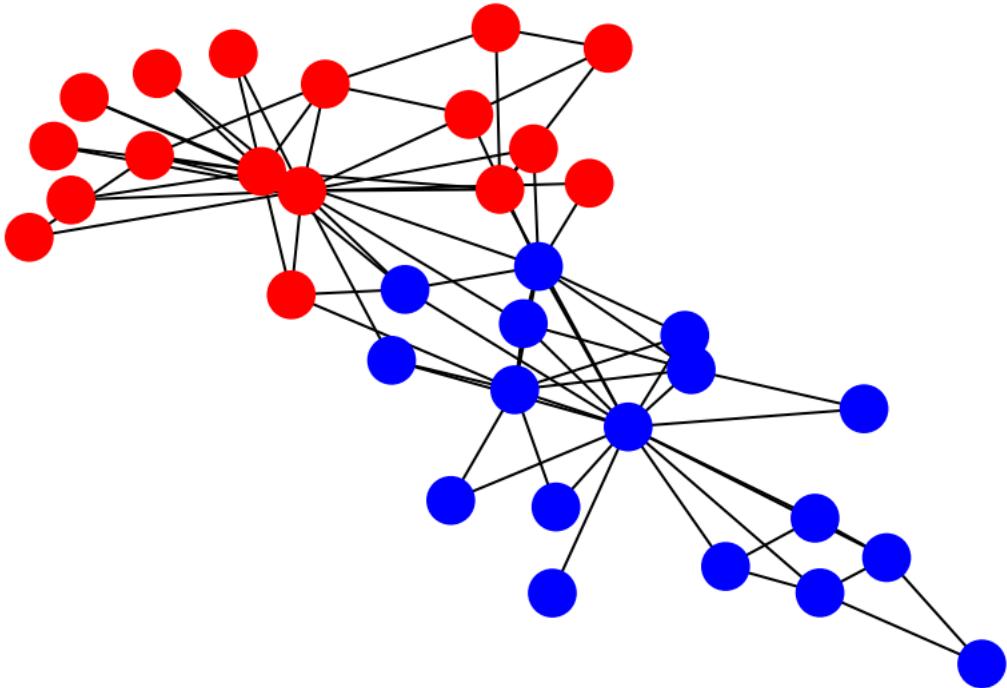
(continues on next page)

(continued from previous page)

```

nx.draw(
    zachary,
    node_color=[
        {"Officer": "r", "Mr. Hi": "b"} .get(dt["club"]) for _, dt
        in zachary.nodes(data=True)
    ],
)
plt.show()

```



Задачу о выделении сообществ в тривиальном случае разбиения графа на два подмножества можно свести к знакомой нам задаче о максимальном разрезе. Правда, в отличие от задачи о максимальном разрезе, в случае с сообществами мы хотим иметь минимальное число ребер между ними. Но это можно сделать просто поменяв пару символов в выражении для стоимости:

$$C = \sum_{u,v,w \in E} w(\mathbf{1}(u \in V_1, v \in V_1) + \mathbf{1}(u \in V_2, v \in V_2))$$

Но на самом деле мы только что свели более простую задачу о минимальном разрезе с неотрицательными весами к более сложной задаче Max-Cut. А еще выбранная нами метрика (количество ребер) – не самый лучший вариант для этой задачи. Гораздо лучше подойдет модулярность (modularity), предложенная физиком Марком Ньюманом [New06]:

$$Q(C) = \frac{1}{2|E|} \sum_{e \in E} B_{e_{src}, e_{dst}} \delta(c_{e_{src}}, c_{e_{dst}})$$

Тут B – это матрица модулярности (modularity matrix). Ее элементы определяются через степени d_i соответствующих вершин графа (степень вершины – это число ребер, связанных с данной вершиной) и матрицу смежности A графа:

$$B_{ij} = A_{ij} - \frac{d_i d_j}{2|E|}$$

Условно, модулярность – это разница между числом ребер внутри сообществ в нашем графе и числом ребер внутри сообществ в графе с таким же числом ребер, но сгенерированным случайным образом. Это довольно сложное понятие, которое выходит за рамки нашего курса, но все равно потребуется нам, чтобы показать, что задача оптимизации модулярности может быть сформулирована как задача Изинга.

Note: Это интересно, но одним из первых алгоритмов для решения задачи о выделении сообществ в графах был алгоритм имитации отжига, который изначально был создан именно для решения проблемы гамильтонианов типа Изинга. Причина заключается в том, что модулярность очень схожа по виду с выражением энергии для магнетиков.

Мы тут пока описали лишь простой случай модулярности для не взвешенного и ненаправленного графа. Но даже в таком случае для задачи точной оптимизации модулярности не известно полиномиального алгоритма решения. Поэтому обычно применяют приближенные или жадные алгоритмы, и они вроде даже неплохо работают. Но мы почти не знаем, насколько действительно далеко они от самых оптимальных решений, особенно для больших графов.

60.3 Задача о Гамильтоновых циклах

Перед тем как перейти к интересной и важной задаче поиска Гамильтоновых циклов, мы вспомним задачу о мостах Кенигсберга (Калининграда). Ведь именно гуляя по этому городу и пытаясь решить эту задачу, Леонард Эйлер изобрел теорию графов. Суть задачи: нужно обойти все острова города, пройдя по каждому мосту лишь один раз, и вернуться на тот остров, откуда стартовал. Эйлер, создав математический аппарат теории графов, сумел доказать, что это невозможно, ну а дальше завертелось, и вот мы с вами тут.

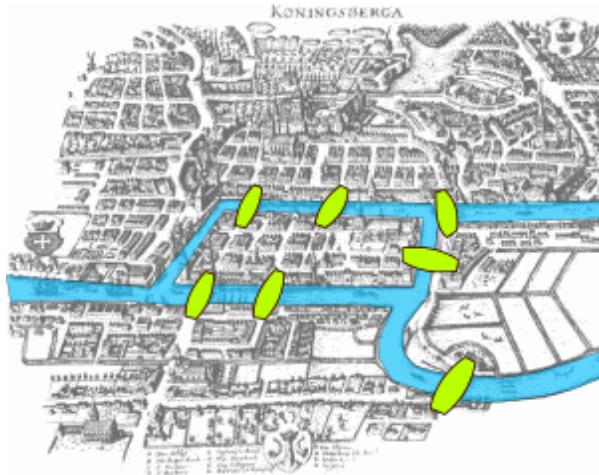


Fig. 60.2: Мосты Кенингсберга, думая о которых, Эйлер изобрел теорию графов

Такой путь в графе, когда мы проходим по каждому ребру лишь один раз, называется Эйлеров цикл. Но нам будет более интересен схожий класс циклов – Гамильтоновы циклы. Это такие циклы, которые проходят через каждую вершину графа ровно один раз.

Для Гамильтонова цикла мы можем ввести N^2 бинарных переменных $x_{i,p}$. Каждая переменная $x_{i,p}$ равна 1, если i -я вершина находится на p -м шаге пути и 0 если нет. Тогда легко ввести условия существования такого цикла:

$$\begin{cases} \sum_p x_{i,p} = 1 & \forall i \\ \sum_i x_{i,p} = 1 & \forall p \\ (x_{i,p} = 1) \wedge (x_{j,p+1} = 0) & \forall i, j \notin E \end{cases}$$

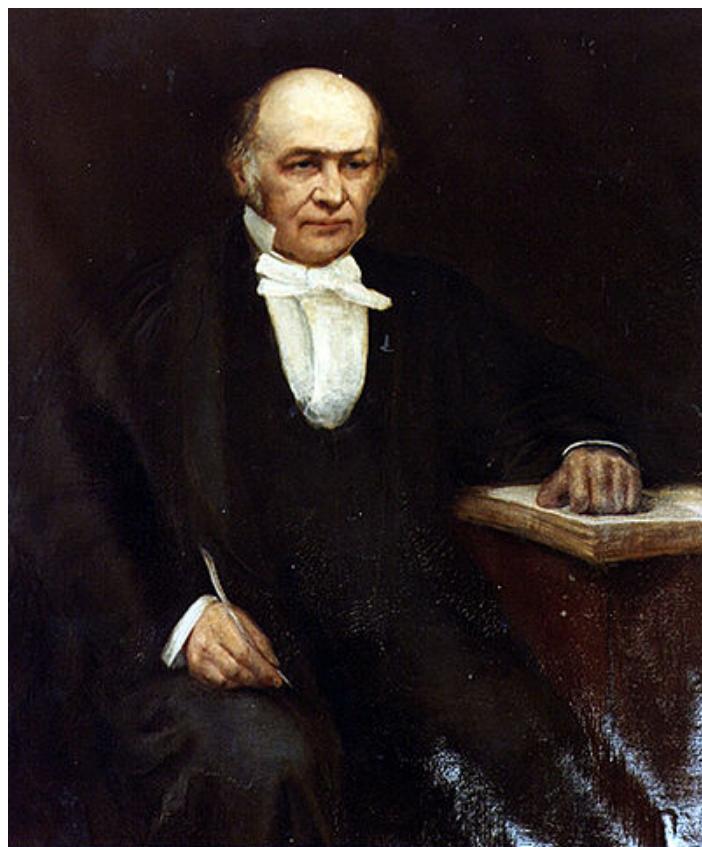


Fig. 60.3: Сер Уильям Роэн Гамильтон, 1805 - 1865

Тут первое условие говорит нам о том, что каждая вершина должна попасть в путь. Второе условие – каждый шаг пути содержит строго одну вершину. Ну а третий шаг – просто утверждение о том, что между вершинами соседних шагов пути должно быть ребро. На самом деле, эти три условия можно переписать в единую функцию стоимости:

$$C = \left(1 - \sum_i x_{i,p}\right)^2 + \left(1 - \sum_p x_{i,p}\right)^2 + \sum_{u,v \notin E} x_{u,p}x_{v,p+1}$$

Правда, в этом случае мы должны минимизировать, а не максимизировать эту величину.

60.4 Задача коммивояжера

Задачу коммивояжера мы (а точнее наш смартфон) решаем каждый раз, когда строим в Google Maps маршрут, включающий несколько точек. Зная, как формулируется задача о гамильтоновых циклах, сформулировать задачу коммивояжера очень легко.



Fig. 60.4: Иллюстрация задачи коммивояжера

По сути нам требуется взять все Гамильтоновы циклы и выбрать из них тот, для которого сумма весов по содержащимся в нем ребрам будет минимальной. Но надо помнить, что цикл обязательно должен быть в первую очередь Гамильтоновым, поэтому мы добавим веса слагаемых в выражение для стоимости, причем веса,

отвечающие за сам цикл, будут больше:

$$C = A(1 - \sum_i x_{i,p})^2 + A(1 - \sum_p x_{i,p})^2 + A \sum_{u,v \notin E} x_{u,p} x_{v,p+1} + B \sum_{u,v,w \in E} w x_{u,p} x_{v,p+1}$$

Тут A, B это веса, которые лучше выбирать так, что $0 < Bw < A \quad \forall u, v, w \in E$.

Note: Функции стоимости для задач и Гамильтоновых циклах и задачи коммивояжера являются все же приближением исходной формулировки, так как формально мы переходим от задачи с ограничениями к задаче без ограничений. В общем случае нет гарантии, что решение задачи без ограничений будет эквивалентно решению задачи с ограничениями. Но для целей нашей лекции и курса в целом мы опустим сложный момент исследования эквивалентности этих задач, потому что обычно оно работает вполне корректно.

60.5 Задача о рюкзаке

Довольно простая для понимания задача, к которой, однако, сводится огромное число куда более сложных задач. Например, именно задачей о рюкзаке является проблема оптимизации портфеля биржевых акций, или, например, оптимизации графиков работы сотрудников McDonald's, учитывая ожидаемое число посетителей и т.д.

Суть задачи заключается в том, что у нас есть рюкзак ограниченного объема. А еще есть набор разных предметов – каждый из них имеет свой объем и стоимость. Наша цель в данном случае заключается в том, чтобы найти оптимальный набор предметов так, чтобы они влезали в наш рюкзак, при этом их стоимость была бы максимально возможной.

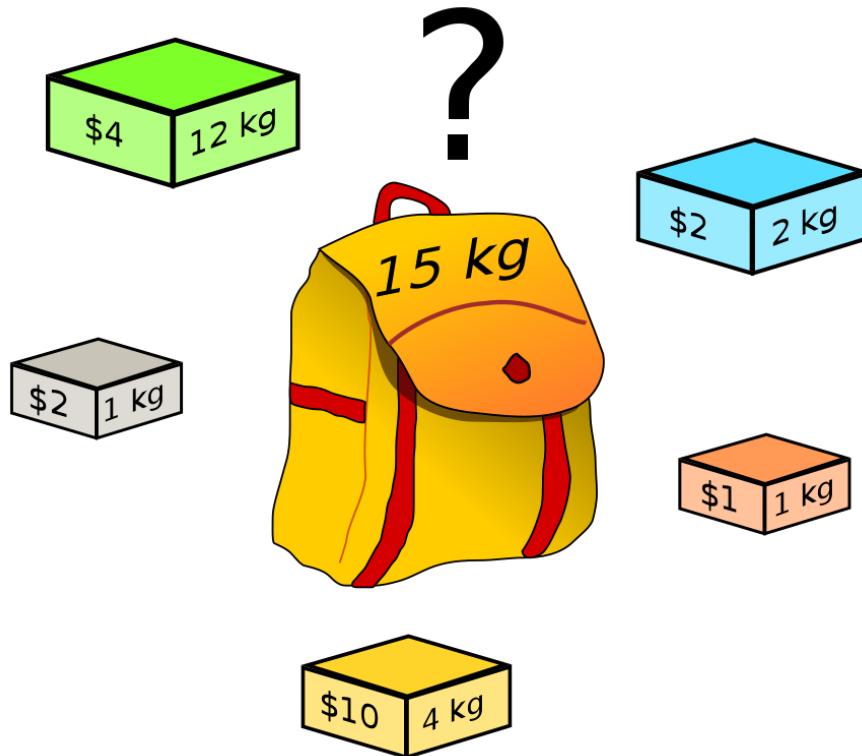


Fig. 60.5: Иллюстрация задачи о рюкзаке.

Формально это можно записать следующим образом. Пускай у нас есть рюкзак объема V . Давайте обозначим количество экземпляров i -й вещи в рюкзаке переменной x_i , а ее объем v_i . Ценность i -го предмета обозначим как s_i . Решением задачи будет вектор X из целых чисел x_i , которые при этом могут быть равны нулю – это будет значить, что i -я вещь не входит в оптимальный набор. Тогда нам необходимо решить следующую задачу:

$$\begin{cases} X = \operatorname{argmax}\{\sum_i x_i \cdot s_i\} \\ \sum_i x_i \cdot v_i \leq W \end{cases}$$

60.6 Классические алгоритмы

Самый простой и очевидный вариант решения таких проблем – это просто перебор всех возможных комбинаций. Но так как сложность задач растет экспоненциально ($O(N) \sim e^N$), то очень быстро прямое решение (еще иногда говорят *brute force* решение, или *brute force* алгоритм) становится невозможным. Понятно, что можно воспользоваться какими-то эвристиками для конкретной задачи. Это приводит нас к алгоритмам типа *ветвей и границ* (*branch and bound*) [LD10], в которых мы обычно строим дерево возможных решений и пытаемся перебирать не все подряд, а лишь “перспективные” ветви этого дерева, отсекая те ветки, где хороших решений точно не будет. Но сложность таких алгоритмов все равно остается экспоненциальной. Другой вариант – это искать не обязательно самое лучшее решение, а хотя бы какое-то подходящее решение (*feasible solution*, решение, которое удовлетворяет ограничениям). Это приводит нас к аппроксимационным алгоритмам, которые каким-то образом пытаются свести экспоненциальную задачу к полиномиальной, пусть и теряя в гарантированном качестве итогового решения.

Важное замечание

Задачи комбинаторной оптимизации нельзя точно решить на квантовом компьютере! Квантовые компьютеры лишь дают нам потенциально очень эффективные *аппроксимационные* алгоритмы. Причем в основе этих алгоритмов лежит обычно как раз квантовое машинное обучение!

60.6.1 Общий вид алгоритма ветвей и границ

Давайте запишем как в общем виде выглядит алгоритм ветвей и границ ([источник](#)).

1. Найти любое решение x_h проблемы, которое бы удовлетворяло ограничениям. Сохраним значение целевой функции для этого решения: $B = f(x_h)$. Оно будет определять лучшее из известных на данный момент решений.
2. Инициализировать очередь для хранения частных решений. В этом случае разбиваем пространство возможных решений на маленькие подпространства и строим дерево потенциальных решений. Эта часть индивидуальная для каждой задачи. Например, в случае задачи о рюкзаке, можно взять начальное решение и построить две его ветви: одна будет содержать все решения, где больше одного предмета, а другая – где его будет меньше.
3. Выполняем итерации, пока наша очередь не пустая.
 - взять узел N из очереди;
 - если узел N представляет собой терминальный узел x дерева и $f(x) < B$, то это новое лучшее известное решение; сохраняем его $B = f(x)$;
 - если нет, то разбиваем N для создания новых узлов N_i :
 - если верхняя оценочная граница решений для этой ветки $\text{bound}(N_i)$ больше B , то ничего не делаем;

- в противном случае добавляем N_i в очередь.

В данном случае нам необходимо реализовать для конкретной задачи следующие процедуры:

- построения дерева решения
- оценки верхней границы для ветви

В целом, данный алгоритм сегодня это, пожалуй, лучшее, что можем использовать, если нам необходимо точное решение. Также всегда есть опция остановить поиск до того, как мы достигнем оптимума – в этом варианте алгоритм становится аппроксимационным.

60.6.2 Жадные алгоритмы

Теперь давайте вернемся к аппроксимационным алгоритмам, которые работают относительно быстро, но, к сожалению, часто дают довольно плохие результаты. Ну и самым простым вариантом таких алгоритмов будет жадный алгоритм. Рассматривать его будем на примере задачи о рюкзаке. Пусть у нас есть рюкзак объема 30 и набор предметов с их стоимостью и объемом:

$$items = \{(4, 370), (9, 1950), (10, 3500), (21, 6700), (17, 6100), (3, 800), (27, 8300)\}$$

```
capacity = 30
items = [(4, 370), (9, 1950), (10, 3500), (21, 6700), (17, 6100), (3, 800), (27, 8300), ]
```

Наше решение будет максимально простым:

1. считаем удельную стоимость предметов, то есть стоимость единицы их массы;
2. набираем сначала самых дорогих предметов;
3. по остаточному принципу набираем остальные предметы, отдавая предпочтения тем, которые дороже по удельной стоимости.

```
items_and_score = sorted(
    [(it[0], it[1], it[1] / it[0]) for it in items],
    key=lambda x: x[2],
    reverse=True
)

print("Items, sorted by relative cost:")
for it in items_and_score:
    print(f"Weight: {it[0]}\tCost: {it[1]}\tRelative Cost: {it[2]}")
```

Items, sorted by relative cost:		
Weight: 17	Cost: 6100	Relative Cost: 358.8235294117647
Weight: 10	Cost: 3500	Relative Cost: 350.0
Weight: 21	Cost: 6700	Relative Cost: 319.04761904761904
Weight: 27	Cost: 8300	Relative Cost: 307.4074074074074
Weight: 3	Cost: 800	Relative Cost: 266.66666666666667
Weight: 9	Cost: 1950	Relative Cost: 216.66666666666666
Weight: 4	Cost: 370	Relative Cost: 92.5

“Заполняем” рюкзак:

```

solution = []
w = capacity
min_weight = min([it[0] for it in items_and_score])

while True:
    if w < min_weight:
        break
    else:
        cand = [it for it in items_and_score if it[0] <= w][0]
        solution.append(cand)
        w -= cand[0]

final_score = sum([it[1] for it in solution])
final_weight = sum([it[0] for it in solution])

print(f"Final score: {final_score}")
print(f"Total weight of items: {final_weight}")

```

```

Final score: 10400
Total weight of items: 30

```

В данном случае это сработало неплохо – мы нашли действительно хорошее решение. Однако можно заметить, что решение из трех элементов (10, 3500) было бы более выгодным, так как итоговая ценность была бы 10500, что на 100 единиц больше жадного решения. Но в силу своей *жадности* наш алгоритм не смог найти это решение, хотя найденное им и лежит довольно близко к лучшему. Так что очень часто жадные алгоритмы находят очень плохие решения, причем качество решений сильно падает с ростом размерности проблемы.

60.6.3 Метод имитации отжига

Это итеративный алгоритм, который очень часто способен найти действительно неплохое решение. Причем, в отличие от жадных алгоритмов, отжиг специально сделан так, чтобы не “застревать” в каких-то локально-хороших точках пространства решений, а наоборот, искать самое лучшее. Мы разберем работу отжига на примере задачи о максимальном разрезе в графе, а именно, реализуем подобие алгоритма Метрополиса-Гастингса [Has70]. Но для начала давайте сгенерируем случайный граф:

```

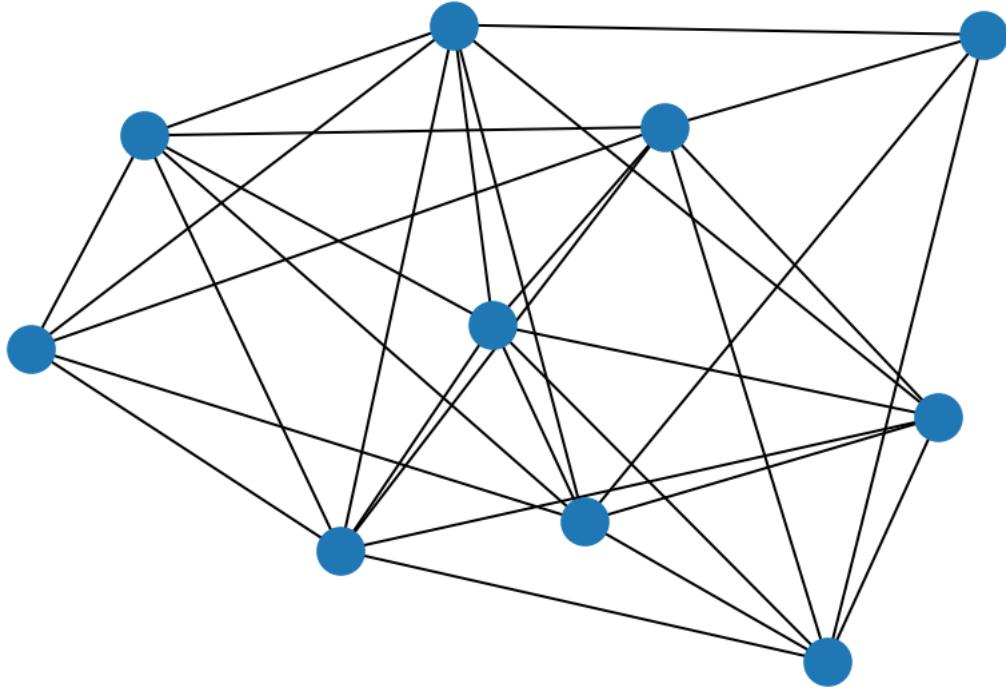
import numpy as np
from typing import List
from typing import Union

np.random.seed(42)
rand_mat = np.random.rand(10, 10)
rand_adj = (rand_mat + rand_mat.T) / 2
rand_adj[rand_adj < 0.35] = 0.0
np.fill_diagonal(rand_adj, 0)

g = nx.Graph(rand_adj)
nx.draw(g)

plt.show()

```



Напомню, что цель состоит в том, чтобы разбить множество вершин на два подмножества так, чтобы сумма весов ребер между двумя подмножествами была максимальной. Для этого понадобится функция, которая считает целевое значение для любого разбиения на два подмножества.

```
def score(g: nx.classes.graph.Graph, x: List[int]) -> (float):
    score = 0
    for e in g.edges(data=True):
        if x[e[0]] != x[e[1]]:
            score += e[2]["weight"]
    return score
```

Разобьем вершины на две группы случайным образом несколько раз и проверим, что функция работает корректно:

```
def split_random_node(random_seed: int) -> (List[int]):
    np.random.seed(random_seed)
    random_x = [
        1 if np.random.random() <= 0.5 else -1
        for _ in range(g.number_of_nodes())
    ]
    print(f"Random seed {random_seed}\nScore: {score(g, random_x):.2f}")

    return random_x

for random_seed in [2019, 2020, 2021]:
    random_x = split_random_node(random_seed)
```

Random seed 2019	Score: 7.78
------------------	-------------

(continues on next page)

(continued from previous page)

Random seed 2020	Score: 7.23
Random seed 2021	Score: 9.38

Суть процесса отжига заключается в следующем:

1. генерируем начальное случайное решение (или получаем *feasible* при помощи эвристик);
2. задаем начальную “температуру” – некий глобальный метапараметр, суть которого станет ясна далее;
3. выполняем отжиг заданное число итераций;
 - выполняем случайную модификацию решения;
 - если значение функции стоимости для нового решения лучше, чем для старого, то принимаем его;
 - если нет, то все равно можем принять новое решение, но лишь с некоторой вероятностью, которая тем больше, чем выше температура и чем ближе друг к другу по оценке старое и новое решение.

Давайте реализуем это. В качестве функции, которая дает нам вероятность принять/отклонить новое решение, будем использовать [распределение Больцмана](#):

$$P_{acceptance} = e^{\frac{E_{new}-E_{old}}{T}}$$

Видно, что эта величина может быть больше единицы в случае, когда новое решение лучше старого, но для нас это не проблема – это просто будет значить, что мы точно принимаем новое решение!

```
from copy import copy

e_history = []
x = random_x
T = T_0 = 100
e = score(g, x)
e_history.append(e)

def bolzman(e_old: float, e_new: float, T: Union[int, float]) -> (float):
    return np.exp((e_new - e_old) / T)

def permute(x: List[int]) -> (List[int]):
    i = np.random.randint(0, len(x) - 1)
    x_new = copy(x)
    x_new[i] *= -1

    return x_new

for i in range(1500):
    new_state = permute(x)
    new_e = score(g, new_state)

    if new_e > e:
        e = new_e
        x = new_state
    else:
        prob = bolzman(e, new_e, T)
        if np.random.rand() <= prob:
            e = new_e
            x = new_state

    e_history.append(e)
T = T_0 / (i + 1)
```

(continues on next page)

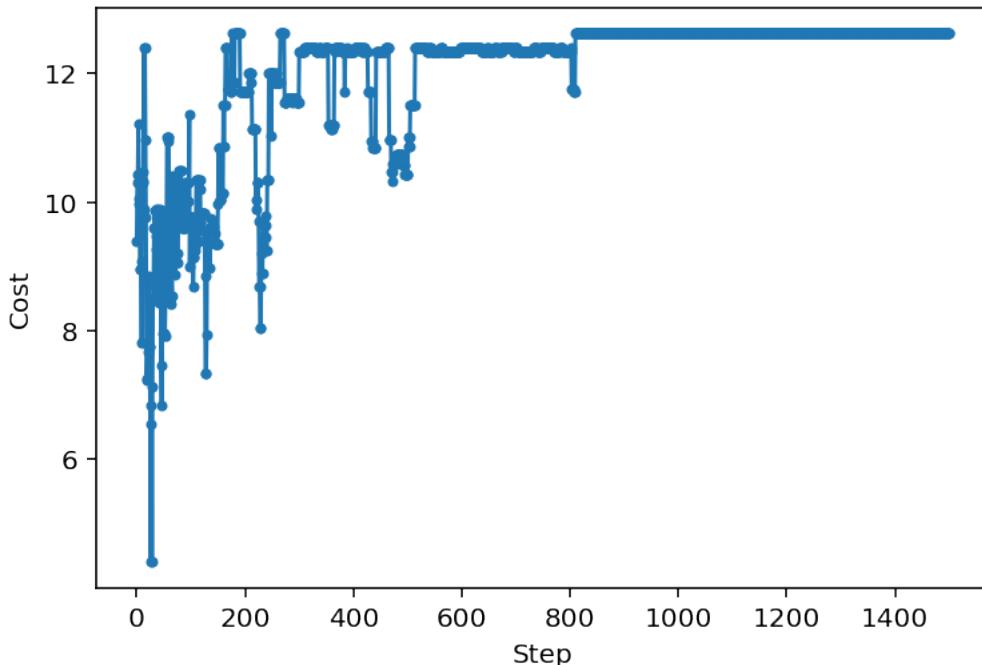
(continued from previous page)

```
print(f"Final energy: {e:.2f}")
```

```
Final energy: 12.62
```

И посмотрим, как оно сходилось:

```
plt.plot(np.arange(len(e_history)), e_history, ".-")
plt.xlabel("Step")
plt.ylabel("Cost")
plt.show()
```



Видим, что в начале у нас был активный поиск решения, а ближе к концу переходы становились возможны, лишь если новое решение лучше. Этот процесс чем-то похож на кристаллизацию расплавов, когда изначально все частицы плавают в жидкости, но с уменьшением температуры они все более точно становятся в те позиции, которые обеспечивают минимум свободной энергии кристалла. Именно поэтому данный алгоритм называется *имитацией отжига* (*simulated annealing*).

Этот алгоритм неплохо находит оптимальные решения, но с ростом размерности задачи, а также пространства решений процесс “отжига” должен длиться все дольше и дольше, а температура уменьшаться все медленней.

60.7 Заключение

В этой лекции мы узнали, что же это за такие NP -задачи, а также познакомились с примерами некоторых из них. Посмотрели, как эти задачи можно решать на классическом компьютере, и какие при этом есть ограничения. В следующих лекциях мы узнаем:

- как задачи комбинаторной оптимизации можно свести к решению модели Изинга (подробнее об этой модели – в [отдельной лекции](#))
- как задача Изинга переходит в задачу об основном состоянии квантовой системы
- как задачу об основном состоянии можно решать на квантовом компьютере разными способами:
 - через алгоритм *VQE*, основанном на [вариационных квантовых схемах](#)
 - через алгоритм *QAOA*, который является квантовым аналогом алгоритма имитации отжига
 - на специальных квантовых компьютерах – [квантовых аннилерах](#)

CHAPTER
SIXTYONE

О БЛОКЕ “ОТ ЗАДАЧ К КВАНТОВЫМ АЛГОРИТМАМ”

Этот блок посвящен тому, как перейти от задач оптимизации или квантовой химии к квантовым алгоритмам. Блок содержит лекции о:

- преобразовании Жордана-Вигнера, где показывается, как перейти от фермионных операторов из квантовой химии к спиновым операторам и модели Изинга;
- формулировке задач комбинаторной оптимизации в терминах спиновых операторов и модели Изинга.

В заключении блока кратко затрагивается тема классических алгоритмов для решения задачи об основном состоянии модели Изинга и квантовых гамильтонианах – алгоритмы Арнольди/Ланкзоша, а также вариационный Монте-Карло и его модификации.

ФОРМУЛИРОВКА ЗАДАЧ ОПТИМИЗАЦИИ В ТЕРМИНАХ МОДЕЛИ ИЗИНГА

Не зря мы так глубоко изучали [модель Изинга](#) и [анализировали](#) ее решения – сегодня увидим, как в терминах этой модели можно сформулировать большинство важных задач [комбинаторной оптимизации](#).

62.1 QUBO матрица

Но сначала кратко обсудим так называемую QUBO матрицу [GKD19] – это еще один способ записать задачу модели Изинга, и дальше по курсу иногда будем этим пользоваться.

QUBO – это сокращение от **Quadratic Unconstrained Binary Optimization**, или, если переводить, то это *задача квадратичной оптимизации без ограничений*. То есть это такие задачи, для которых нет отдельных ограничений, например, равенств или неравенств, а функция стоимости представима в виде многочлена второй степени от входных переменных. Решением такой задачи является бинарный вектор $X = \{x_1, x_2, \dots, x_n\}$ такой, что $x_1, \dots, x_n \in \{0, 1\}$. В этом случае функцию стоимости C можно записать в виде:

$$C = a_1 x_1^2 + a_2 x_2^2 + \dots + a_n x_n^2 + b_1 x_1 + \dots + b_n x_n + c_{1,2} x_1 x_2 + c_{1,3} x_1 x_3 + \dots + c_{n,n-1} x_n x_{n-1}$$

Но так как переменные бинарные, то разницы между x_i^2 и x_i нет – дальше будем использовать a_i как единственные коэффициенты, считая, что b_i уже и так включены в a_i . В этом случае можно представить функцию стоимости как матрицу размера $|X| \times |X|$, на диагонали которой стоят коэффициенты a_1, \dots, a_n , а вне диагонали стоят коэффициенты, с которыми в стоимость входят пары элементов:

$$Q = \begin{pmatrix} a_1 & c_{1,2} & \dots & c_{1,n} \\ \dots & \dots & \dots & \dots \\ c_{n,1} & \dots & \dots & a_n \end{pmatrix}$$

Сама оптимизационная задача в этом случае формулируется следующим образом:

$$\arg \min_X X^T Q X$$

Note: Тут рассматриваем именно минимизацию функции стоимости. Но если исходная задача формулировалась в терминах максимизации чего-то, например, как задача о рюкзаке или максимальном разрезе в графе, то очевидно, что просто домножив стоимость на -1, перейдем от максимизации к минимизации. Далее это будет важно, так как будем рассматривать задачу поиска *основного состояния* квантомеханической системы, а такое состояние – это по определению состояние именно с *минимальной* энергией.

62.2 QUBO как квантовый гамильтониан

Если вспомнить, как выглядит модель Изинга, то легко заметить, что там, как и в QUBO-проблемах, есть лишь члены первой и второй степени. Вот только спиновые операторы (матрицы Паули) имеют собственные значения ± 1 , а не $\{0, 1\}$, которые фигурируют в QUBO. Это проблема, так как квадраты этих значений не равны им самим. Но это легко можно исправить введя “бинарный” оператор \hat{x} :

$$\hat{x} = \frac{1 + \hat{\sigma}^z}{2} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$$

Введя “промежуточные” переменные sz_i которые связаны с x_i через это выражение ($x_i = \frac{1+sz}{2}$), всегда можно свести выражение для C к следующему виду:

$$C = \sum_{i,j \in \{1, \dots, |X|\}} J_{i,j} sz_i sz_j + \sum_{i \in \{1, \dots, |X|\}} h_i sz_i$$

А это уже почти модель Изинга! Заменяем sz на операторы σ^z и получаем гамильтониан системы. Причем такой, что минимум его энергии будет совпадать с минимальным значением функции стоимости C :

$$\hat{H} = \sum_{i,j \in \{1, \dots, |X|\}} J_{i,j} \sigma_i^z \sigma_j^z + \sum_{i \in \{1, \dots, |X|\}} h_i \sigma_i^z$$

$$\Psi |\hat{H}| \Psi = \min X^T Q X = \min C, \quad \text{если } \Psi \text{ -- это основное состояние системы}$$

Причем в силу того, что у в выражении для \hat{H} фигурируют лишь σ^z , основное состояние гамильтониана является “классическим”, то есть все спины “ориентированы” либо вверх, либо вниз, но не находятся в суперпозиции. Другими словами, гамильтониан диагонализируется в \mathbf{Z} -базисе.

62.3 Поиск основного состояния

Как увидели, задачи, сформулированные в терминах QUBO, можно свести к задаче нахождения основного состояния квантовой системы, которая описывается некоторым оператором. Но что это вообще значит? И что это за гамильтониан такой, о котором все время говорим? Вообще говоря, гамильтониан – это оператор энергии:

$$\hat{H} = \hat{T} + \hat{U},$$

где T – это кинетическая энергия системы, а U – потенциальная энергия. Именно этот оператор определяет квантовую динамику в уравнении Шредингера. То есть это оператор, зная который (а также начальное состояние системы), можно посчитать волновую функцию системы ($\Psi(t)$) в любой момент времени в будущем. Правда, это лишь в теории, так как сложность интегрирования уравнения Шредингера очень быстро растет с ростом размера системы. Этой темы немного уже касались в [лекции про виды квантового машинного обучения](#), когда рассматривали, как нейросети помогают решать это самое уравнение.

Еще коснемся этой темы в следующих лекциях, но пока достаточно понимания того, что гамильтониан полностью описывает квантовую систему. С другой стороны, зная гамильтониан, всегда можем построить “в железе” квантовую систему, которую он описывает. Например, конфигурируя магнитные поля и двухчастичные обменные взаимодействия.

Таким образом, решение проблемы QUBO сводится к поиску Ψ_{GS} – волновой функции, отвечающей основному состоянию:

$$\arg \min_{\Psi} \hat{H}$$

А финальное решение получается просто как наиболее вероятная конфигурация спинов в состоянии Ψ_{GS} . Ну и останется лишь перейти от полученных собственных значений операторов σ_i^z к собственным значениям операторов \hat{x}_i .

Note: Кстати, справедлив и обратный переход, то есть решив проблему QUBO каким-нибудь из [классических алгоритмов](#) для решения комбинаторных задач, автоматически получим конфигурацию основного состояния соответствующей физической системы! Этот подход впервые был предложен аж в 1988-м году в работе [BGrotschelJungerR88]. Таким образом, приходим к пониманию **тесной связи задач квантовой физики и комбинаторной оптимизации**. Именно на этом и строится огромное число перспективных квантовых алгоритмов для решения задач реального мира. Особенно в *NISQ* эпоху!

Важное преимущество квантового представления QUBO-задач – в том, что наш мир устроен таким образом, что любая квантовая система всегда стремится в состояние с минимальной энергией, то есть основное. На этом построен даже целый класс квантовых аннилиров. Но и для вариационных квантовых алгоритмов это также дает свои преимущества.

62.4 Статья “Ising formulations of many NP problems”

Основным источником информации для нас будет статья “*Ising formulations of many NP problems*” [Luc14], вышедшая в 2014-м году (версия 3). В данной лекции рассмотрим лишь часть примеров из этой работы, хотя наше рассмотрение будет чуть более подробным. В целом эта статья может быть использована как прекрасный справочник.

62.5 Задача о максимальном разрезе в графе (повторение)

Уже рассматривали эту задачу в [лекции о модели Изинга](#) и в [лекции про задачи комбинаторной оптимизации](#), но теперь повторим еще раз. Итак, есть граф на множестве вершин V , связанных множеством ребер E . Каждое ребро соединяет вершины u, v . Для простоты будем рассматривать случай ненаправленного графа. Каждое ребро имеет вес w . Цель – разбить множество вершин V на два непересекающихся сообщества V_1, V_2 таким образом, чтобы суммарный вес ребер, соединяющих вершины из разных сообществ, был максимальен:

$$\arg \max_{V_1, V_2} \sum_{u, v, w \in E} w(\mathbf{1}(u \in V_1, v \in V_2) + \mathbf{1}(u \in V_2, v \in V_1))$$

Эта задача уже является задачей без ограничений и может быть сразу сформулирована в терминах QUBO и модели Изинга.

62.5.1 QUBO матрица

QUBO матрица для этой задачи имеет размер $|V| \times |V|$, а вектор решения X это, соответственно, будет бинарный вектор длины $|V|$. Для простоты обозначим сообщество V_1 как вершины, для которых $x_i = 0$, а V_2 это будут вершины с $x_i = 1$. Еще хотим решать задачу минимизации вместо задачи максимизации. Запишем новую целевую функцию:

$$C = - \sum_{i, j \in \{1, \dots, |V|\}} w_{i,j} (x_i + x_j - 2x_i x_j)$$

Чтобы записать QUBO матрицу, будет удобнее работать с матрицей смежности графа, а не списком его ребер. Матрица смежности A (*adjacency matrix*) – это матрица размера $|V| \times |V|$, элементы которой это веса $w_{i,j}$, если в

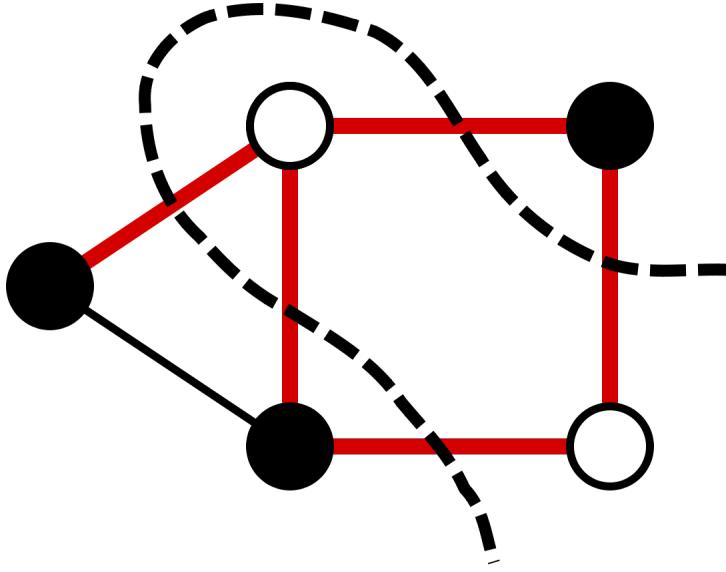


Fig. 62.1: Иллюстрация задачи о максимальном разрезе в графе

графе есть ребро между вершинами i и j , и 0, если ребра нет. Тогда матрица QUBO будет иметь следующий вид:

$$QUBO = \begin{pmatrix} \sum_{j=0}^{|V|-1} A_{0,j} & -2A_{0,1} & \dots & -2A_{0,|V|} \\ \dots & \dots & \dots & \dots \\ -2A_{|V|,0} & \dots & \dots & \sum_{j=0}^{|V|-1} A_{|V|,j} \end{pmatrix}$$

62.5.2 Гамильтониан Изинга

В случае этой задачи можно сказать, что она изначально имеет вид модели Изинга. И действительно, наиболее вероятная конфигурация спинов для основного состояния системы с гамильтонианом такого вида:

$$\hat{H} = \sum_{i,j \in \{1, \dots, |V|\}} (1 - \sigma_i^z \sigma_j^z)$$

будет в точности соответствовать решению задачи о максимальном разрезе. При этом численное значение энергии будет отличаться ровно на величину $|E|$, то есть на число ребер, так как величина $(1 - \sigma_i^z \sigma_j^z)$ будет равна нулю, если вершины находятся в разных сообществах, или единице, если они находятся в одном.

62.6 Задача коммивояжера

Задача коммивояжера обсуждалась в [лекции по комбинаторной оптимизации](#), где были получены выражения для представления данной задачи в виде “без ограничений”. Напомним, что удалось добиться этого, “внеся” ограничения в выражения для целевой функции в виде штрафа за отклонение от ограничений. А также добавили соответствующие коэффициенты. Полученное в той лекции выражение имеет вид:

$$C = A(1 - \sum_i x_{i,p})^2 + A(1 - \sum_p x_{i,p})^2 + A \sum_{u,v \notin E} x_{u,p} x_{v,p+1} + B \sum_{u,v,w \in E} w x_{u,p} x_{v,p+1}$$

Для нас удобно, что это уже задача минимизации. В данном случае QUBO-матрица получается при помощи явного раскрытия скобок в выражении для стоимости. Можно заметить, что в этом случае получаем также элементы 0-й степени, но формат QUBO-матрицы такого не предусматривает. Но во-первых, в данном случае легко можем определить разницу между $X^T Q X$ и минимумом C , а во-вторых, для нас это не столь важно – нам нужно решение, а значение энергии/функции стоимости получается без каких-либо проблем за полиномиальное время.

Note: Это довольно важное замечание, так как часто можно найти относительно простое представление задачи в виде QUBO, а вот учет всех констант может сильно усложнить ее вид. Более того, как увидим далее, не все представления QUBO одинаково эффективны, особенно когда переходят к решению на квантовом компьютере: для каких-то видов QUBO будет одна величина энергетической щели между основным и возбужденным состоянием в процессе решения, а для других QUBO-представлений той же задачи оно уже может стать больше!

62.7 Задача о выделении сообществ в графе

Уже говорили об этой задаче в лекции про комбинаторную оптимизацию. Для этой задачи было разработано немало эвристических алгоритмов поиска [SCBR14], а в этой части сосредоточимся на формулировке как задачи QUBO.

62.7.1 $k = 2$ сообществ

В случае всего двух сообществ будем использовать двоичные спиновые переменные $s_i \in \{-1, 1\}$ для кодирования того, к какому именно сообществу принадлежит вершина i . $\frac{1+s_i s_j}{2}$ будет равняться 1, если i и j принадлежат одному сообществу, и 0 – в противном случае. Модулярность при этом будет следующей

$$M = \frac{1}{4m} s^T B s \quad (62.1)$$

$$B_{ij} = (A_{ij} - \frac{g_i g_j}{2m}) \quad (62.2)$$

При $s_i = 2x_i - 1$ между спиновыми $s_i \in \{-1, 1\}$ и битовыми $x_i \in \{0, 1\}$ переменными соответственно, а также $\sum_{i,j} B_{i,j} = 0$, максимизация модулярности (62.2) может быть эквивалентна выражению задачи минимизации в форме QUBO с гамильтонианом $H = -\frac{1}{m} x^T B x$ и с QUBO матрицей $Q = -\frac{B}{m}$.

62.7.2 Множественные сообщества ($k > 2$)

Чтобы сформулировать задачу выделения k сообществ в канонической форме QUBO, сначала одновременно кодируем двоичные переменные x_i , а затем строим QUBO-гамильтониан.

Используем схему однократного кодирования (one-hot encoding), в которой задаем $x_{i,c} = 1$, если вершина $i \in c$, и $x_{i,c} = 0$ в противном случае, то есть,

$$x_{i,c} = \begin{cases} 1, & i \in c \\ 0, & \end{cases} \quad (62.3)$$

Тогда потребуется k переменных на логическую вершину, и размер двоичного вектора решений x увеличится с вектора длины N для случая $k = 2$ до $k \times N$ для k -сообществ. В частности, задаем $x = (x_{1,1}, x_{2,1}, \dots, x_{N,1}, \dots, x_{1,k}, x_{2,k}, \dots, x_{N,k})$.

$k > 2$ формулирует проблему как бинарную задачу минимизации, тогда возможно построить гамильтониан QUBO для k -сообществ с $H_M = -\frac{1}{m} \sum_{c=1}^k x_c^T B x_c$, в котором каждый член в сумме описывает бинарную задачу обнаружения сообщества для данного сообщества c .

Представляя обобщенную матрицу модулярности B размером $kN \times kN$ и блочно-диагональной формы с B вдоль диагонали (62.4), можем переписать задачу обнаружения k -сообщества как задачу двоичной минимизации в каноническом формате QUBO (где исходные многоклассовые переменные вложены в большее число двоичных переменных):

$$H_M = -\frac{1}{m} x^T B x$$

$$B = \begin{bmatrix} B & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & B \end{bmatrix} \quad (62.4)$$

Поскольку каждый узел $i = 1, \dots, N$ должен находиться ровно в одном сообществе $c = 1, \dots, k$, необходимо добавить штрафной член, чтобы ограничить решение для распределения сообществ. Формально это ограничение можно записать как $\sum_{c=1}^k x_{i,c} = 1$ при $i = 1, \dots, N$.

Это линейное ограничение можно добавить в задачу QUBO в виде квадратичного штрафного члена:

$$H_P = P \sum_{i=1}^N \left(\sum_{c=1}^k x_{i,c} - 1 \right)^2 \quad (62.5)$$

с положительным предварительным коэффициентом $P > 0$, обеспечивающим выполнение ограничений. Чтобы сформулировать штрафной член в (62.5) в QUBO-гамильтониан $H_P = x^T Q_P x$, перенумеруем двоичный вектор решений, используя один подстрочный индекс от 1 до kN , то есть $x = (x_{1,1}, x_{2,1}, \dots, x_{N,1}, \dots, x_{N,k}) = (x_1, x_2, \dots, x_N, \dots, x_{kN})$. Тогда штрафной член (62.5) может быть переписан как

$$H_P = P(Vx - b)^T(Vx - b) \quad (62.6)$$

где b – вектор всех единиц, а $V = [I_N, \dots, I_N]$ – высокоструктурированная матрица размера $N \times k_N$ с $N \times N$ матрицами тождества, сложенными горизонтально рядом друг с другом. Если вес P достаточно велик, например, находится в диапазоне от $1/N$ до $10/N$, то штрафной гамильтониан H_P будет стремиться к решению, в котором все ограничения удовлетворены $Vx = b$. Поскольку гамильтониан H_P квадратичен с точностью до константы, его можно переписать в канонической форме QUBO как $H_P = x^T Q_P x$, с $Q_P = P(V^T V - 2 \operatorname{diag}(V^T b))$.

Объединив модульности в уравнении (62.4) и штрафной гамильтониан в уравнении (62.6), получаем окончательный QUBO-гамильтониан для общего решения задачи о выделении k сообществ в графе

$$H = H_M + H_P$$

62.8 Заключение

Из этой лекции узнали, что такое QUBO матрица, а также как от такой формулировки оптимизационных задач можно перейти к задаче поиска основного состояния квантомеханической системы. В будущих лекциях познакомимся уже непосредственно с тремя основными методами решения этой задачи:

1. квантовый отжиг В этом случае реализуем модель Изинга буквально в железе. Это сразу дает ряд ограничений и особенностей, но с другой стороны мы получаем возможность хорошо масштабировать систему. И действительно, сегодня компьютеры фирмы D-Wave имеют порядка нескольких тысяч кубитов. Многие специалисты считают, что до появления универсальных квантовых компьютеров именно аналоговые машины D-Wave, решающие задачу Изинга, станут основным коммерческим инструментом в квантовых вычислениях.

2. вариационно-градиентные методы В этом подходе попытаемся закодировать волновую функцию основного состояния системы при помощи *вариационной квантовой схемы*, а дальше найти такие параметры, которые минимизируют результат измерения гамильтониана в таком состоянии. Этот метод называется *Variational Quantum Eigensolver*.
3. третий популярный подход соединяет идеи первых двух В этом случае также делаем квантовый отжиг, но для этого не строим целевую систему в железе, а производим симулацию квантовой динамики на кубитах при помощи специальных приближений из области квантовой механики. А параметры “отжига”, как и в VQE, подбираем при помощи градиентных методов. Такой подход носит название *Quantum Approximate Optimization Algorithm*.

Именно этим темам будет посвящена большая часть оставшихся лекций.

ПРЕОБРАЗОВАНИЕ ЖОРДАНА-ВИГНЕРА

63.1 Описание лекции

В этой лекции мы узнаем как реально существующей системе (например, электронам на решетке) сопоставить систему состоящую из спинов (кубитов), для того чтобы просимулировать ее на квантовом компьютере. Для этого мы сначала изучим альтернативный, зачастую более удобный подход к описанию многочастичных квантовых систем, который называется “вторичное квантование”.

63.2 Введение

Для того чтобы просимулировать квантовую систему на квантовом же компьютере (например, посчитать энергию основного состояния какого-нибудь атома), нам необходимо закодировать состояние системы и действующие на нее операторы: найти соответствие между состояниями системы и состояниями нашего компьютера (т.е., кубитов).

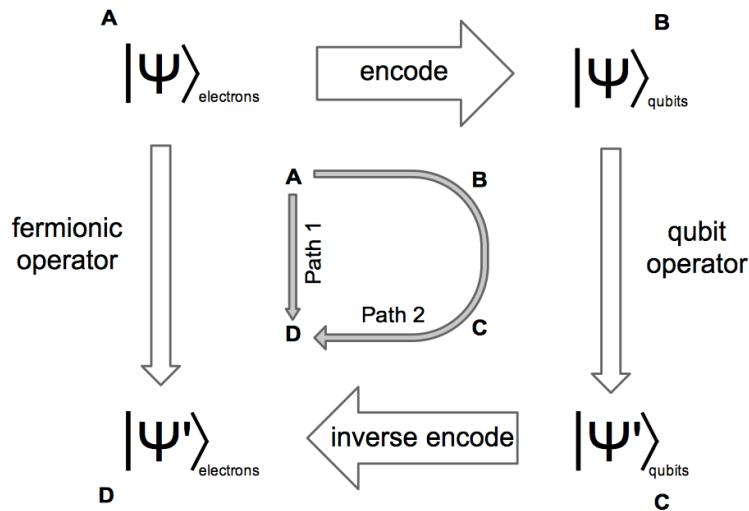


Fig. 63.1: Общая схема симуляции квантовой системы на квантовом компьютере (взято из [SRL12]).

63.3 Спины, фермионы и бозоны

Как мы помним из [лекции по квантовой химии](#), квантовые частицы могут быть либо бозонами, либо фермионами. Одно из различий между ними состоит в том, что при перестановке двух бозонов волновая функция не меняется, а при перестановке двух фермионов – меняет знак. Эта классификация следует из того, что квантовые частицы неотличимы.

Несмотря на то, что наши кубиты состоят из каких-то частиц (бозонов или фермионов), обычно они отличимы (например, зафиксированы на своих позициях) и, следовательно, не являются ни бозонами, ни фермионами. Так как мы используем два состояния кубита ($|0\rangle$ и $|1\rangle$), каждый кубит может быть описан как система со спином 1/2.

63.4 Вторичное квантование

В квантовой механике мы [можем описать](#) состояние нескольких частиц как тензорное произведение состояний каждой из частиц. Например, если у нас есть две частицы, и их квантовое состояние описывается положением частицы в пространстве, r_i , мы можем записать состояние частиц как

$$|\psi\rangle = |r_1\rangle \otimes |r_2\rangle.$$

У этого подхода есть два главных недостатка – во первых, работать с системами в которых разное количество частиц, или где это количество может меняться, не очень удобно. Во-вторых, не учитывается неразличимость квантовых частиц.

Обе эти проблемы решаются вторичным квантованием, где вместо использования состояния каждой частицы мы описываем систему количеством частиц в каждом из возможных состояний. Так как частицы неразличимы, такое описание достаточно для полного описания системы. Вместо Гильбертова пространства фиксированной размерности наши состояния теперь являются элементами пространства Фока (суммы Гильбертовых пространств для всех возможных значений количества частиц). Основное состояние, или вакуум, теперь это состояние с 0 частиц.

Note: Несмотря на название, мы не квантуем систему повторно, а просто избавляемся от избыточности в описании.

Мы так же можем определить операторы, которые добавляют или убирают частицу в определенном состоянии из системы. На языке первой квантизации:

$$\begin{aligned} b_\alpha^\dagger |\Psi\rangle &= \frac{1}{\sqrt{N+1}} |\psi_\alpha\rangle \otimes |\Psi\rangle \\ b_\alpha (|\psi_\alpha\rangle \otimes |\Psi\rangle) &= \frac{1}{\sqrt{N}} |\Psi\rangle. \end{aligned}$$

Note: До сих пор операторы которые мы видели действовали из гильбертова пространства с определенным количеством частиц в него самого. Оператор создания действует из гильбертова пространства с n частиц в гильбертово пространства с $n + 1$ частиц (оба этих пространства являются частью пространства Фока).

Тут b_α^\dagger называется оператором создания (creation), а b_α – уничтожения (annihilation). Заметим, что эти операторы не эрмитовы, т.е., $b_\alpha^\dagger \neq b_\alpha$. Нормализация операторов выбрана таким образом, что собственные значения (эрмитова) оператора $b_\alpha^\dagger b_\alpha$ – количество частиц в состоянии α .

Note: Наблюдаемые величины соответствуют эрмитовым операторам, следовательно b_α не соответствует никакой наблюдаемой величине. В частности, такой член сам по себе не может быть частью гамильтониана (но может быть частью эрмитова члена, например $b_\alpha + b_\alpha^\dagger$.

Если в системе нет ни одной частицы в состоянии α , то оператор уничтожения уничтожает состояние:

$$b_\alpha |\Psi\rangle = 0.$$

0 тут это нулевой вектор, в отличии от $|0\rangle$, вакуума. Так как 0 это не физическое состояние (например, если бы мы захотели измерить какую-нибудь величину X , то вероятность найти систему в любом собственном состоянии $|x\rangle$ будет равно нулю), мы можем интерпретировать это как тот факт, что нельзя уничтожить частицу которой нет.

Note: Хотя стандартная конвенция говорит о том, что векторы состояния нормализованы ($\langle\psi|\psi\rangle = 1$), после применения оператора (эрмитова или нет) мы не обязаны получить нормализованное состояние. Можно считать что после каждого применения оператора мы нормализуем состояние, либо, чуть более математически, что мы рассматриваем проективное гильбертово пространство, где все параллельные векторы эквивалентны.

В зависимости от типа частиц, операторы создания и уничтожения подчиняются определенным отношениям. В случае бозонов, разные операторы коммутируют:

$$\begin{aligned} [b_i^\dagger, b_j] &= \delta_{ij} \\ [b_i^\dagger, b_j^\dagger] &= [b_i, b_j] = 0. \end{aligned}$$

В случае фермионов, операторы антисимметричны:

$$\begin{aligned} \{c_i^\dagger, c_j\} &= \delta_{ij} \\ \{c_i^\dagger, c_j^\dagger\} &= \{c_i, c_j\} = 0. \end{aligned}$$

В частности,

$$\{c_j^\dagger, c_j^\dagger\} = 2(c_j^\dagger)^2 = 0,$$

и, следовательно, в системе не может быть больше одного фермиона в одном состоянии (в соответствии с принципом запрета Паули). Из этого так же следует что $c_j^2 = 0$, так как независимо от состояния системы, после первого оператора в ней не останется фермиона который можно было бы уничтожить.

63.5 Переход от спинов к фермионам

Пронумеруем возможные состояния частиц индексом i : например, это могут быть электроны в состояниях, отвечающих гауссианам из STO-3G или находящиеся в разных узлах цепочки атомов. Мы можем попробовать сопоставить спину фермион: если спин с индексом i направлен вниз то фермион с индексом i существует, а если спин направлен вверх – то фермион не существует (количество фермионов с определенным индексом может быть только 1 или 0). Другими словами, используя оператор количества частиц $\hat{n}_i = c_i^\dagger c_i$, где c_i^\dagger и c_i это операторы создания и уничтожения соответственно, мы хотели бы сопоставить

$$\hat{\sigma}_i^z = 1 - 2\hat{n}_i$$

Тогда лестничные (ladder) операторы $\sigma^- = (\hat{\sigma}_i^x - i\hat{\sigma}_i^y)/2 = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$ и $\sigma^+ = (\hat{\sigma}_i^x + i\hat{\sigma}_i^y)/2 = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$, соответствуют операторам создания и уничтожения.

Note: Заметим что оператор σ^- переводит состояние “спин вверх” в состоянии “спин вниз”, а состояние “спин вниз” уничтожает. σ^+ переводит состояние “спин вниз” в состоянии “спин вверх” (а состояние “спин вверх” уничтожает). Оба этих оператора неэрмитовы.

Действительно, в одном состоянии (если индексы двух операторов равны) эти операторы выполняют фермионное антисимметрическое отношение

$$\{\sigma_j^+, \sigma_j^-\} = 1.$$

К сожалению, на разных узлах эти операторы коммутируют, а не антисимметрически. Чтобы это исправить, мы “прикрепляем” к каждому фермиону “нить” (string):

$$\begin{aligned}\sigma_i^+ &= \left[\prod_{j < i} (1 - 2c_j^\dagger c_j) \right] c_i \\ \sigma_i^- &= \left[\prod_{j < i} (1 - 2c_j^\dagger c_j) \right] c_i^\dagger.\end{aligned}$$

Оператор $\prod_{j_i} (1 - 2c_j^\dagger c_j)$ равен ± 1 в зависимости от четности количества фермионов слева от узла i .

Заметим, что c_k антисимметрически коммутирует с $(1 - 2c_k^\dagger c_k)$:

$$\begin{aligned}\{c_j, (1 - 2c_j^\dagger c_j)\} &= c_j(1 - 2c_j^\dagger c_j) + c_j(1 - 2c_j^\dagger c_j)c_j = \\ &\quad c_j - 2 \underbrace{c_j c_j^\dagger}_{1 - c_j^\dagger c_j} c_j + c_j - 2c_j^\dagger \underbrace{c_j c_j}_{0} = \\ &\quad c_j - 2c_j + 2c_j^\dagger \underbrace{c_j c_j}_{0} + c_j = 0,\end{aligned}$$

и, следовательно, с нитью $\prod_{j_i} (1 - 2c_j^\dagger c_j)$ (интуитивно, если сначала разрушить фермион, то четность изменится).

Пусть, без ограничения общности, $\ell > k$:

$$\sigma_k^+ \sigma_\ell^- = \left[\prod_{j < k} (1 - 2c_j^\dagger c_j) \right] c_k \left[\prod_{m < \ell} (1 - 2c_m^\dagger c_m) \right] c_\ell^\dagger.$$

Если мы перенесем c_k вправо, то выражение умножится на -1 дважды (один раз из-за изменения четности ℓ -нити, и один раз из-за обмена с c_ℓ^\dagger). В то же время, i -нить коммутирует со всем, поэтому ее мы можем перенести вправо без изменений:

$$\sigma_k^+ \sigma_\ell^- = \left[\prod_{j < k} (1 - 2c_j^\dagger c_j) \right] c_k \left[\prod_{m < \ell} (1 - 2c_m^\dagger c_m) \right] c_\ell^\dagger = \left[\prod_{m < \ell} (1 - 2c_m^\dagger c_m) \right] c_\ell^\dagger \left[\prod_{j < k} (1 - 2c_j^\dagger c_j) \right] c_k = \sigma_\ell^- \sigma_k^+,$$

как и требовалось. Мы так же можем записать обратное отношение:

$$\begin{aligned}c_i &= \prod_{j < i} \sigma_j^z \sigma_i^+ \\ c_i^\dagger &= \prod_{j < i} \sigma_j^z \sigma_i^-.\end{aligned}$$

Проверка антисимметрического отношения оставляется читателю в качестве упражнения.

Таким образом мы установили соответствие между фермионной и спиновой системами, но операторы в обоих случаях очень нелокальны. В частности, один фермионный оператор соответствует произведению $\mathcal{O}(N)$ спиновых операторов (и наоборот).

Note: Существуют и другие способы сопоставления фермионной и спиновой систем, например, Bravyi–Kitaev transform [BK02]. Также существует способ сопоставить бозоны и фермионы – бозонизация [VDS98].

63.5.1 Пример. Модель Хаббарда

Одной из важных моделей в физике твёрдого тела является модель Хаббарда, описывающая взаимодействие электронов на решётке. В соответствии с принципом Паули, на каждом узле может быть максимум два электрона (один со спином вверх и один со спином вниз).

Гамильтониан модели Хаббарда состоит из двух членов. Первый, кинетический, описывает движение электрона по решётке:

$$H_k = -t \sum_{\langle i,j \rangle, \sigma} c_{i,\sigma}^\dagger c_{j,\sigma} + \text{h.c.},$$

где $\langle i,j \rangle$ означает что i и j индексы соседних узлов решётки, σ обозначает спин электрона, а *h.c.* (hermtian conjugate) обозначает слагаемое, эрмитово сопряжённое написанному.

Оператор $c_{i,\sigma}^\dagger c_{j,\sigma}$ уничтожает электрон на узле j и создает электрон с таким же спином на узле i , то есть, фактически, перемещает электрон с узла j на узел i .

Note: Эрмитово сопряжённый оператор $(c_{i,\sigma}^\dagger c_{j,\sigma})^\dagger = c_{j,\sigma}^\dagger c_{i,\sigma}$ перемещает электрон с узла i на узел j .

Так как этот член квадратичный, мы можем точно решить гамильтониан содержащий только его.

Второй член содержит отталкивающее взаимодействие двух электронов на одном узле, соответствующее отталкиванию двух зарядов:

$$H_U = U \sum_i n_{i,\uparrow} n_{i,\downarrow} = U \sum_i c_{i,\uparrow}^\dagger c_{i,\uparrow} c_{i,\downarrow}^\dagger c_{i,\downarrow}$$

Полный гамильтониан

$$H = -t \sum_{\langle i,j \rangle, \sigma} c_{i,\sigma}^\dagger c_{j,\sigma} + \text{h.c.} + U \sum_i n_{i,\uparrow} n_{i,\downarrow}.$$

Для того чтобы описать соответствующий спиновый Гамильтониан, каждому узлу мы сопоставим два кубита (один для каждого фермиона). Для простоты, рассмотрим одномерную цепочку:

$$\begin{aligned} c_{i,\downarrow} &= \prod_{k<2i} \sigma_k^z \sigma_{2i}^+ \\ c_{i,\uparrow} &= \prod_{k<2i+1} \sigma_k^z \sigma_{2i+1}^+ \end{aligned}$$

Следовательно, кинетическая часть равна

$$H_k = -t \sum_{\langle i,j \rangle, \sigma} c_{i,\sigma}^\dagger c_{j,\sigma} + \text{h.c.} = -t \sum_{\langle i,j \rangle} \prod_{\ell<2j} \sigma_\ell^z \sigma_{2j}^- \prod_{k<2i} \sigma_k^z \sigma_{2i}^+ + \text{h.c.}$$

Заметим, что хоть каждый оператор создания/уничтожения сильно не локален, в их произведении большая часть произведения отменяется (так как $(\sigma_j^z)^2 = 1$):

$$\begin{aligned} H_k &= -t \sum_j \sigma_{2j}^- \sigma_{2j}^z \sigma_{2j+1}^z \sigma_{2j+2}^+ + \sigma_{2j}^+ \sigma_{2j}^z \sigma_{2j+1}^z \sigma_{2j+2}^- = \\ &= -t \sum_j (\sigma_{2j}^- \sigma_{2j+2}^+ + \sigma_{2j}^+ \sigma_{2j+2}^-) \sigma_{2j}^z \sigma_{2j+1}^z = \\ &= -t \sum_j (\sigma_{2j}^x \sigma_{2j+2}^x + \sigma_{2j}^y \sigma_{2j+2}^y) \sigma_{2j}^z \sigma_{2j+1}^z \end{aligned}$$

Второй член дает нам (используя $\hat{\sigma}_i^z = 1 - 2\hat{n}_i$)

$$H_U = \frac{U}{4} \sum_j (1 - \sigma_{2j}^z)(1 - \sigma_{2j+1}^z)$$

Таким образом, мы получили спиновый гамильтониан:

$$H_k - t \sum_j (\sigma_{2j}^x \sigma_{2j+2}^x + \sigma_{2j}^y \sigma_{2j+2}^y) \sigma_{2j}^z \sigma_{2j+1}^z + \frac{U}{4} \sum_j (1 - \sigma_{2j}^z)(1 - \sigma_{2j+1}^z)$$

ПРОБЛЕМА СОБСТВЕННЫХ ЗНАЧЕНИЙ

64.1 Введение

Мы с вами узнали, что задачи комбинаторной оптимизации и квантовой химии могут быть сведены к решению проблемы поиска минимального собственного значения большого эрмитова оператора – гамильтониана. Для оптимизационных задач это *осуществляется* при помощи сведения к *QUBO*-матрице и гамильтониану типа Изинга. А для электронных орбиталей из квантовой химии можно *применить* преобразование Жордана-Вигнера и также перейти к спиновому гамильтониану.

Теперь перед нами встает вопрос, а как же искать основное состояние этого гамильтониана? В этой лекции рассмотрим **классические** методы решения этой проблемы, то есть без квантовых компьютеров. Рассмотрение этих методов и их недостатков покажет то, зачем тут так нужен будет квантовый компьютер.

64.2 О проблеме (повторение)

Эта тема обсуждалась во вводных лекциях по линейной алгебре, в части про собственные вектора и собственные значения.

Итак, пусть у имеется диагонализируемая матрица A размерности $n \times n$, она же является линейным оператором \hat{A} . Из линейной алгебры знаем, что у этой матрицы есть n таких чисел e_i и векторов Ψ_i , что для них выполняется условие:

$$A\Psi_i = e_i\Psi_i$$

или в нотации Дирака, которая используется в области квантовых вычислений:

$$\hat{A}\Psi_i = e_i\Psi_i$$

Таким образом, собственные вектора – это такие вектора, которые при применении оператора не меняют свое направление. Например, в примере ниже собственный вектор – это ось симметрии оператора:

64.3 Итеративные алгоритмы

В целом, задача нахождения собственных значений является очень трудной с вычислительной точки зрения, особенно для больших матриц. Для матриц размера более, чем 3×3 в общем случае не существует алгоритма нахождения собственных значений и собственных векторов. Однако существует несколько итеративных алгоритмов. Рассмотрим лишь два из них, причем без особых деталей, так как эти алгоритмы, а также доказательство их сходимости являются достаточно сложными.

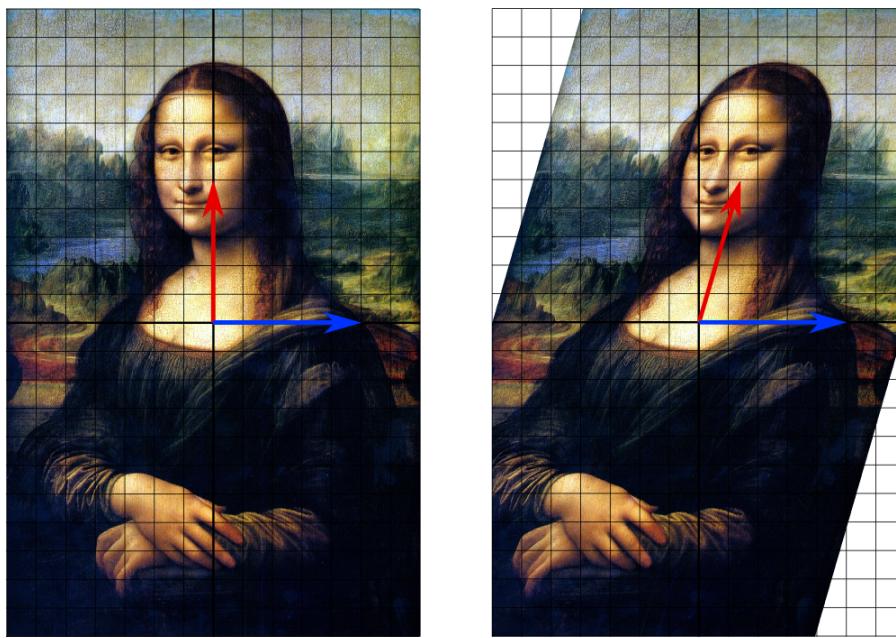


Fig. 64.1: Синий вектор, в отличии от красного, при применении оператора не меняет направление так как является его собственным вектором.

64.3.1 Степенной метод

Один из самых простых для понимания алгоритмов, который, тем не менее находит интересные применения. Суть его в том, что берем некоторый случайный вектор Ψ и начинаем последовательно действовать на него оператором \hat{A} (другими словами умножать, на нашу матрицу), при этом нормируя:

$$\Psi_{i+1} = \frac{\hat{A}\Psi_i}{\|\hat{A}\|}$$

И так повторяем до тех пор, пока изменение вектора не будет меньше, чем некоторое заданное маленькое значение ϵ . Когда достигли этого условия, это значит что нашли первый собственный вектор, который соответствует наибольшему собственному значению. В частном случае интересных нам эрмитовых операторов, можно так же последовательно находить все собственные вектора и собственные значения.

Note: На самом деле, сеть интернета является графом – множеством связанных между собой вершин. А любой граф можно представить в виде большой-большой, но очень разреженной матрицы, каждый элемент которой это 1 если между соответствующими вершинами есть ребро и 0, если нет. Например, элемент L_{ij} будет 1, если между вершинами i и j есть ребро. В 1998-м году, Ларри Пейдж и Сергей Брин нашли очень эффективный способ подсчета первого собственного вектора этой матрицы, используя именно модификацию степенного метода. Этот алгоритм получил название PageRank, причем *Page* это фамилия автора, а не отсылка к веб-страницам, как можно было бы подумать. Этот алгоритм лег в основу поисковика *Google*, который в дальнейшем вырос в транснациональную корпорацию!

64.3.2 Итерация Арнольди

Это гораздо более сложный метод, который, однако, является одним из самых эффективных применительно к разреженным матрицам [Arn51]. Объяснить его легко, к сожалению, не получится, так как алгоритм требует понимания Крыловских подпространств и других концептов из области линейной алгебры разреженных систем. Но пока достаточно лишь того, что этот алгоритм имеет очень эффективную реализацию – [ARPACK](#), написанную в середине 90-х годов на языке FORTRAN77. Именно эта библиотека используется “под капотом” у SciPy, а также во многих других научных пакетах. Давайте посмотрим, как она работает.

Сгенерируем большую разреженную матрицу.

```
import numpy as np
from scipy import sparse

np.random.seed(42)
x = np.random.random(10000)
np.random.seed(42)
y = np.random.random(10000)
px = np.where(x > 0.2)
py = np.where(y > 0.2)
num_elements = max([px[0].shape[0], py[0].shape[0]])
spmat = sparse.coo_matrix(
    (
        (np.ones(num_elements),
         (px[0][:num_elements], py[0][:num_elements]))
    )
)

print(spmat.__repr__())
```

```
<10000x10000 sparse matrix of type '<class 'numpy.float64'>'  
with 7957 stored elements in COOrdinate format>
```

Матрица размера 10000×10000 это большая матрица и работать с ней в “плотном” (dense) представлении было бы очень трудно. Но ARPACK позволяет найти минимальное собственное значение за доли секунд, используя разреженность матрицы:

```
from scipy.sparse import linalg as sl

max_eigval = sl.eigs(spmat, k=1, which="LR", return_eigenvectors=False)[0]
min_eigval = sl.eigs(spmat, k=1, which="SR", return_eigenvectors=False)[0]

print(f"Min E: {min_eigval}\nMax E: {max_eigval}")
```

```
Min E: (-1.1102230246251565e-16+0j)
Max E: (1.0000000000000007+0j)
```

Для тех кто забыл, какие параметры принимает функция `eigs` из `scipy.linalg.spare` напомним, что первый параметр это разреженная матрица, `k` – сколько именно собственных значений хотим получить, `which` указывает на собственные значения:

- SM – *smallest magnitude* – наименьшие по модулю числа
- LM – *largest magnitude* – наибольшие по модулю числа
- SR – *smallers real* – числа с наименьшей действительной частью
- LR – *largest real* – числа с наибольшей действительной частью

- SI – *smallest image* – числа с наименьшей мнимой частью
- LI – *largest image* – числа с наибольшей мнимой частью

Наконец, параметр `return_eigenvectors` – хотим ли получить только собственные значения, или еще и собственные вектора.

Более подробна работа с `scipy.sparse`, а также с `scipy.sparse.linalg` разбирается в [вводном блоке по линейной алгебре](пока пусто).

Note: Не у всех матриц все собственные значения являются действительными, поэтому ARPACK по умолчанию считает комплексные значения, хотя в этом конкретном случае видим, что мнимая часть равна нулю.

64.3.3 Алгоритм Ланкзоша

Итерация Ланкзоша (англ. Lanzos) [Lan50] – это модификация итерации Арнольди, которая работает с эрмитовыми матрицами и находит максимально широкое применение в том числе для квантовых гамильтонианов. Этот алгоритм по умолчанию включен в большинство математических пакетов, включая ARPACK и, соответственно, SciPy:

```
max_eigval = sl.eigsh(spmat, k=1, which="LM", return_eigenvectors=False) [0]
min_eigval = sl.eigsh(spmat, k=1, which="SM", return_eigenvectors=False) [0]

print(f"Min E: {min_eigval}\nMax E: {max_eigval}")
```

```
Min E: -8.323011768995762e-25
Max E: 1.0000000000000002
```

У этой процедуры из ARPACK немного другие варианты параметра `which`, так как мы помним, что у эрмитовых матриц собственные значения вещественны:

- LM – *largest magnitude* – наибольшие по модулю
- SM – *smallest magnitude* – наименьшие по модулю
- LA – *largest algebraic* – алгебраически наибольшие, т.е. с учетом знака
- SA – *smallest algebraic* – алгебраически наименьшие, т.е. с учетом знака

64.4 Вариационные алгоритмы

В этом разделе поговорим о существующих алгоритмах решения задачи об основном состоянии уже в контексте квантовой механики. Хотя, как помним, задачи оптимизации и квантовой физики [тесно связаны](#). В каком-то смысле, вариационные алгоритмы, а в особенности, квантовый Монте-Карло и различные его модификации в чем-то сильно похожи на классический [алгоритм имитации отжига](#).

64.4.1 Вариационный Монте-Карло

Variational Monte-Carlo, или просто **VMC** это очень простой и в тоже время эффективный алгоритм нахождения основного состояния квантомеханической системы.

Note: Замечание – в классическом **VMC** обычно работают при нулевой температуре. Хотя в общем случае, температура оказывает значительное влияние на то, в каком состоянии находится физическая система.

Давайте еще раз запишем ожидаемое значение энергии гамильтониана в состоянии Ψ :

$$E = \frac{\Psi|\hat{H}|\Psi}{\Psi|\Psi|}$$

Если ввести вектор X , который описывает конфигурацию системы (например, ориентации спинов), то выражение для энергии можно переписать в интегральной форме:

$$E = \frac{\int |\Psi(X)|^2 \frac{\hat{H}\Psi(X)}{\Psi(X)} dX}{\int |\Psi(X)|^2 dX}$$

В данном случае, выражение

$$\frac{|\Psi(X)|^2}{\int |\Psi(X)|^2 dX}$$

дает распределение вероятностей, а значит можно из него семплировать, используя методы Монте-Карло. Это очень похоже на то, как ранее семплировали из распределения Больцмана в [классическом методе Монте-Карло](#). Вопрос лишь в том, как представить волновую функцию Ψ ? В этом помогут так называемые *trial wave functions* – параметризованные функции от X . В этом случае меняем или *варьируем* параметры *trial wave function* в процессе:

- семплируем из $\frac{|\Psi(X)|^2}{\int |\Psi(X)|^2 dX}$ конфигурации;
- обновляем параметризацию *trial function* так, чтобы минимизировать энергию.

Повторяем до сходимости. Ну а дальше посмотрим на некоторые примеры *trial wave functions*.

Jastrow Function

Когда есть задача из N квантовых частиц, каждая из которых описывается координатой или радиус вектором, то можно построить *trial wave function* в виде суммы попарных функций двухчастичных взаимодействий:

$$\Psi(X) = e^{-\sum_{i,j} u(r_i, r_j)},$$

где r_i, r_j – это радиус-векторы частиц, а $u(r_i, r_j)$ – симметричная функция, описывающая двухчастичное взаимодействия. Такая функция называется **Jastrow function** [Jas55]. В этом случае, в процессе работы **VMC** будем просто варьировать радиус-векторы частиц также, как варьировали вершины графа в [обычном отжиге](#), когда решали задачу комбинаторной оптимизации. Только теперь есть еще и параметризация обменных взаимодействий, которую “варьируем”.

Hartree-Fock (SCF)

Для задач квантовой химии, когда работаем с фермионами, существует вид *trial wave function* на основе Слэтеровского детерминанта, о котором писали в [продвинутой лекции по квантовой химии](#):

$$\Psi(R) = D^\dagger D^\downarrow,$$

где D это матрица из одноэлектронных орбиталей:

$$D = \begin{bmatrix} \psi_1(r_1) & \psi_1(r_2) & \dots & \psi_1(r_{N/2}) \\ \dots & \dots & \dots & \dots \\ \psi_{N/2}(r_1) & \psi_{N/2}(r_2) & \dots & \psi_{N/2}(r_{N/2}) \end{bmatrix}$$

Jastrow Function для спинов

Дальше нас будут интересовать как раз модели Изинга и спины, а не частицы в пространстве или орбитали из вторичного квантования. Для спинов можем записать Jastrow function следующим образом:

$$\Psi(s) = e^{\sum_{i,j} s_i W_{i,j} s_j},$$

где матрица W будет играть роль параметризации и отражать парные спиновые корреляции. Давайте посмотрим это на практике при помощи библиотеки NetKet [CCH+19].

```
import netket as nk
```

Моделировать будем простую [модель Изинга](#) для цепочки из 10 спинов (чтобы быстро считалось):

$$\hat{H} = -h \sum_i \sigma_i^x + J \sum_{i,j} \sigma_i^z \sigma_j^z$$

Параметры возьмем такими:

- $J = 0.5$
- $h = 1.321$

```
g = nk.graph.Hypercube(length=10, n_dim=1, pbc=True)
hi = nk.hilbert.Spin(s=0.5, N=g.n_nodes)
op = nk.operator.Ising(h=1.321, hilbert=hi, J=0.5, graph=g)
```

Поскольку модель относительно небольшая по числу частиц, то сразу можем получить точное решение методом Ланкзоша.

```
exact = nk.exact.lanczos_ed(op) [0]
```

Создадим модель на основе Jastrow и VMC:

```
sampler = nk.sampler.MetropolisLocal(hi)
model = nk.models.Jastrow()
optimizer = nk.optimizer.Sgd(learning_rate=0.05)
sr = nk.optimizer.SR(diag_shift=0.01)
vmc = nk.driver.VMC(op, optimizer, sampler, model, n_samples=1000, preconditioner=sr)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0 and
rerun for more info.)
```

Note: Изучение документации библиотеки NetKet оставляем вам самим, так как объяснение абстракций графа и гильбертова пространства, а также использование метода stochastic reconfiguration для вычисления градиентов выходит за рамки лекции. Документация представлена на [сайте NetKet](#).

Запустим оптимизацию:

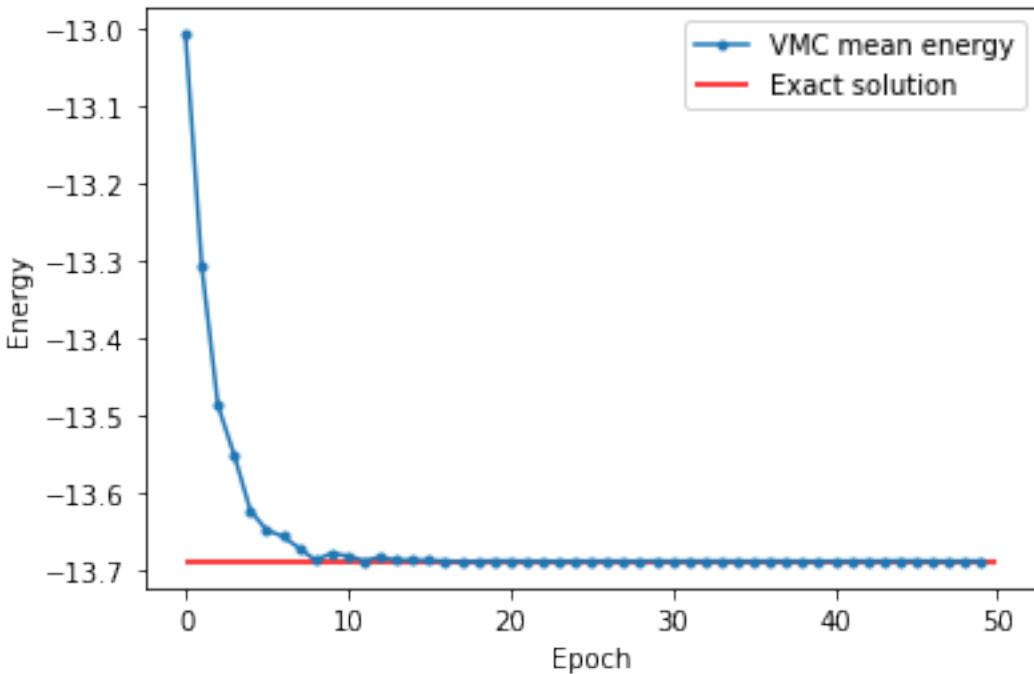
```
logger = nk.logging.RuntimeLog()
vmc.run(50, out=logger, show_progress=False)
```

```
(<netket.logging.runtime_log.RuntimeLog at 0x7f9cfce09fa0>,)
```

Посмотрим на результат:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(6, 4))
plt.plot(list(range(50)), np.real(logger.data["Energy"]["Mean"]), ".-", label="VMC mean energy")
plt.xlabel("Epoch")
plt.ylabel("Energy")
plt.hlines(exact, 0, 50, label="Exact solution", color="red")
plt.legend()
plt.show()
```



Neural Network Quantum States

Еще более интересный подход к выбору *trial wave function* – это использование в качестве $\Psi(X)$ нейронной сети [CT17]. Уже немного касались этой темы, когда речь шла о видах квантового машинного обучения. Хороший вариант, это использовать, например, полностью связную сеть – ограниченную машину Больцмана:

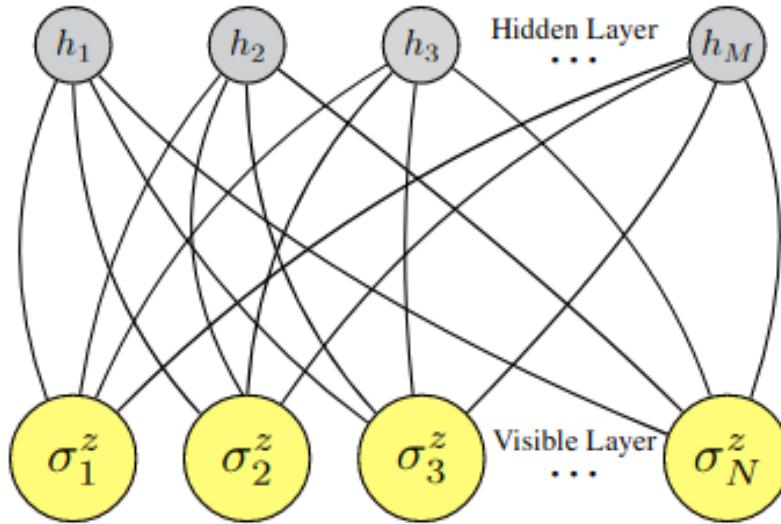


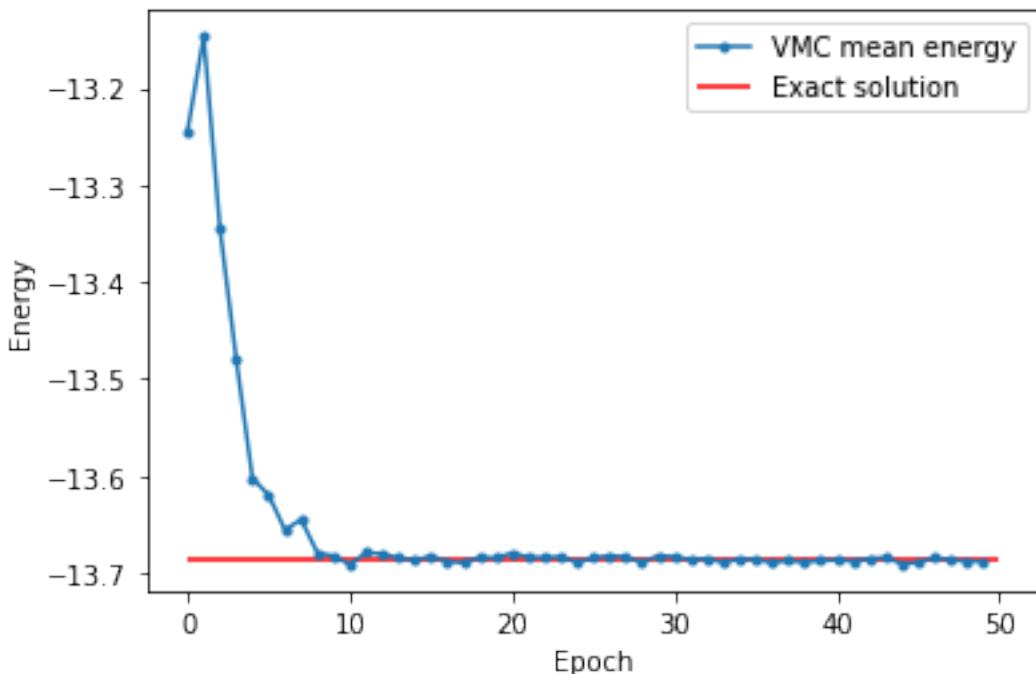
Fig. 64.2: Нейронная сеть в качестве *trial wave function* из работы [CT17].

Это также легко может быть реализовано с использованием библиотеки NetKet:

```
model = nk.models.RBM()
optimizer = nk.optimizer.Sgd(learning_rate=0.05)
sr = nk.optimizer.SR(diag_shift=0.01)
vmc = nk.driver.VMC(op, optimizer, sampler, model, n_samples=1000, preconditioner=sr)

logger = nk.logging.RuntimeLog()
vmc.run(50, out=logger, show_progress=False)

plt.figure(figsize=(6, 4))
plt.plot(list(range(50)), np.real(logger.data["Energy"] ["Mean"]), ".-", label="VMC\u2014mean energy")
plt.xlabel("Epoch")
plt.ylabel("Energy")
plt.hlines(exact, 0, 50, label="Exact solution", color="red")
plt.legend()
plt.show()
```



Преимущества использования нейронной сети трудно показать на таком небольшом примере с моделью Изинга и 10-ю спинами, но они полностью раскрываются, если нужно анализировать более сложные модели.

Note: Это интересно, но при помощи библиотеки NetKet можно по сути решать проблемы комбинаторной оптимизации [SB19] с помощью методов *deep learning*.

64.4.2 Проблемы с VMC

К сожалению, у метода **VMC** есть свои проблемы. Это относительно плохая масштабируемость – при росте размерности проблемы для того, чтобы подобрать реально хорошую аппроксимацию потребуется все больше итераций и семплов на каждой из них. Также у **VMC** есть ряд фундаментальных проблем, например, так называемая *sign problem* [LJGS+90].

64.5 Заключение

В этой лекции рассмотрены известные подходы к решению задачи о минимальном собственном значении на классическом компьютере. Как увидели, все эти методы не могут быть масштабированы на реально большие операторы. Так что для решения этих проблем действительно нужен квантовый компьютер.

CHAPTER
SIXTYFIVE

О БЛОКЕ “VQE”

Этот блок будет полностью посвящен вариационному алгоритму нахождения собственных значений операторов – Variational Quantum Eigensolver.

Базовая лекция расскажет верхнеуровневую интуицию алгоритма и в ней будет пример решения простой задачи Изинга. В продвинутой лекции будет практика решения на квантовом компьютере реальной комбинаторной задачи от начала и до конца.

VARIATIONAL QUANTUM EIGENSOLVER

В этой лекции рассмотрим такой классный и очень перспективный алгоритм как Variational Quantum Eigensolver. Это алгоритм создан для аппроксимационного решения задачи о собственных значениях матриц. Но, как помним, к этой задаче *можно свести* большое число реально интересных задач, например, *NP*-трудные *задачи комбинаторной оптимизации*. Также к решению проблемы о собственных значениях спиновых систем *сводятся* задачи квантовой химии, которые *обсуждались* в отдельной лекции.

Задача о собственных значениях, на решение которой направлен рассматриваемый алгоритм *обсуждалась* ранее, поэтому не будем на ней останавливаться подробно.

66.1 VQE

Теперь давайте посмотрим, как эта задача связана с квантовым компьютером, а также как ее можно свести к градиентному спуску в пространстве параметров VQE, содержащей лишь гейты вращения. Напомню, что наличие лишь гейтов вращения *позволяет эффективно считать градиенты*.

66.1.1 Собственные значения как результат измерения

Итак, есть эрмитова матрица, для которой хотим найти минимальное собственное значение. То есть сделать примерно то, что делает ARPACK или VMC, но на квантовом компьютере.

Note: Если матрица A не эрмитова, то всегда можно ее дополнить до эрмитовой. Поэтому далее, без потери общности, будем считать эту матрицу именно эрмитовой. Между тем не будем рассматривать как именно это делать, пока достаточно знать что это возможно.

Из первых лекций помним, что любое измерение квантового оператора в каком-то состоянии есть проекция этого состояния на пространство *его собственных векторов*, а результатом измерения является одно из его собственных значений. При этом распределение вероятностей получения этих собственных значений определяется волновой функцией Ψ измеряемого состояния. А значит, то, какое собственное значение получим, будет определяться тем, какое состояние приготовим! Более того, минимальное значение энергии, или минимальное собственное значение в терминах этой задачи является наиболее вероятным результатом измерения основного состояния. Напомним, что результат измерения оператора \hat{A} в состоянии Ψ записывается так:

$$e = \Psi \hat{A} \Psi$$

А, как помним из *лекции по параметризованным квантовым схемам*, приготовленное состояние задается собственно самой схемой (еще говорят *ansatz*), а также набором ее классических параметров, которые вариируются в процессе обучения.

Так приходим к алгоритму VQE – Variational Quantum Eigensolver:

1. выбираем *ansatz* – параметризованный оператор $\hat{B}(q)$, который представляет из себя последовательность гейтов вращения; вектор q – это вектор из действительных чисел – параметров;
2. генерируем случайные начальные параметры q_0 ;
3. цикл обучения, повторяем n раз:
 1. готовим состояние $\Psi = \hat{B}(q_n)0$;
 2. измеряем $e = \Psi \hat{A} \Psi$;
 3. вычисляем градиент $g = \frac{\partial e}{\partial q}$;
 4. обновляем параметры $q_{n+1} = q_n - \alpha g$;
4. результатом обучения являются:
 - набор параметров q такой, что $\hat{B}(q)0$ является вектором, близким к первому собственному вектору \hat{A} ;
 - результат измерения e , близкий к первому собственному значению \hat{A} .

66.2 Интуиция алгоритма

Если вспомнить подход **VMC** для решения задачи о собственных значениях, то там использовался *trial wave function* – некоторое, параметризованное кодирование состояния в его вероятность. С некоторой долей условности можно рассматривать *ansatz* в **VQE** как квантовый аналог классических *trial wave functions*. Только вместо семплирования при помощи методов Монте-Карло, условно говоря, сэмплируем из реальной волновой функции, когда считаем математическое ожидание результата измерения. Как [помним из первой лекции](#) измерение есть проекция на один из собственных векторов, то есть можно рассматривать это как выбор сэмпла из пространства всех собственных векторов. Причем для определения вероятности используется состояние после применения параметризованного *ansatz*.

66.3 Пример

Решим простую задачку с тривиальной моделью Изинга, который [рассматривали](#) в посвященной этому лекции. Будем использовать тот же код:

```
import numpy as np
from scipy import sparse
from scipy.sparse import linalg as sl

def sigmaz_k(k: int, n: int) -> (sparse.csr_matrix):
    left_part = sparse.eye(2 ** k)
    right_part = sparse.eye(2 ** (n - 1 - k))

    return sparse.kron(
        sparse.kron(
            left_part,
            sparse.csr_matrix(np.array([[1, 0], [0, -1], ])))
        ),
        right_part
    )
```

(continues on next page)

(continued from previous page)

```
def ising(j: float, h: float, n: int) -> (sparse.csr_matrix):
    res = sparse.csr_matrix((2 ** n, 2 ** n), dtype=np.complex64)

    for i in range(n - 1):
        res += j * sigmaz_k(i, n) * sigmaz_k(i + 1, n)
        res -= h * sigmaz_k(i, n)

    res -= h * sigmaz_k(n - 1, n)

    return res
```

Создадим оператор со значениями $j = 1.0, h = 2.0$:

```
op = sparse.coo_matrix(ising(j=1.0, h=2.0, n=10))
```

66.3.1 Ansatz

Воспользуемся относительно простым вариантом кодирования с использованием $CNOT$ и нескольких слоев.

Note: Более подробно причина выбора именно такого кодирования, а также подробности внутренней реализации на PennyLane рассматривается в практической лекции по VQE.

```
import pennylane as qml

N_SPINS = 10
N_LAYERS = 2

device = qml.device("default.qubit", wires=N_SPINS)
ham = qml.SparseHamiltonian(op, wires=list(range(N_SPINS)), id="Hamiltonian")

def layer(params):
    for i in range(N_SPINS):
        qml.RY(params[i][0], wires=i)
        qml.RZ(params[i][1], wires=i)

    for i in range(N_SPINS - 1):
        qml.CZ(wires=[i, i + 1])

@qml.qnode(device, diff_method="parameter-shift")
def feed_forward(params):
    for k in range(N_LAYERS):
        layer(params[k])

    for i in range(N_SPINS):
        qml.RY(params[-1][i][0], wires=i)
        qml.RZ(params[-1][i][1], wires=i)

    return qml.expval(ham)

params = tuple(
    tuple(
        (
            np.random.rand() * 2 * np.pi,
```

(continues on next page)

(continued from previous page)

```

        np.random.rand() * 2 * np.pi,
    )
    for _ in range(N_SPINS)
)
for _ in range(N_LAYERS)
)

print(f"Result for random weights: {feed_forward(params):.4f}")

```

Result for random weights: -1.4923

66.3.2 Обучение

Возьмем точное решение из лекции по модели Изинга:

```
exact = -11.0
```

Запустим обучение и посмотрим, как оно сходится:

```

import matplotlib.pyplot as plt
%config InlineBackend.figure_format = "retina"

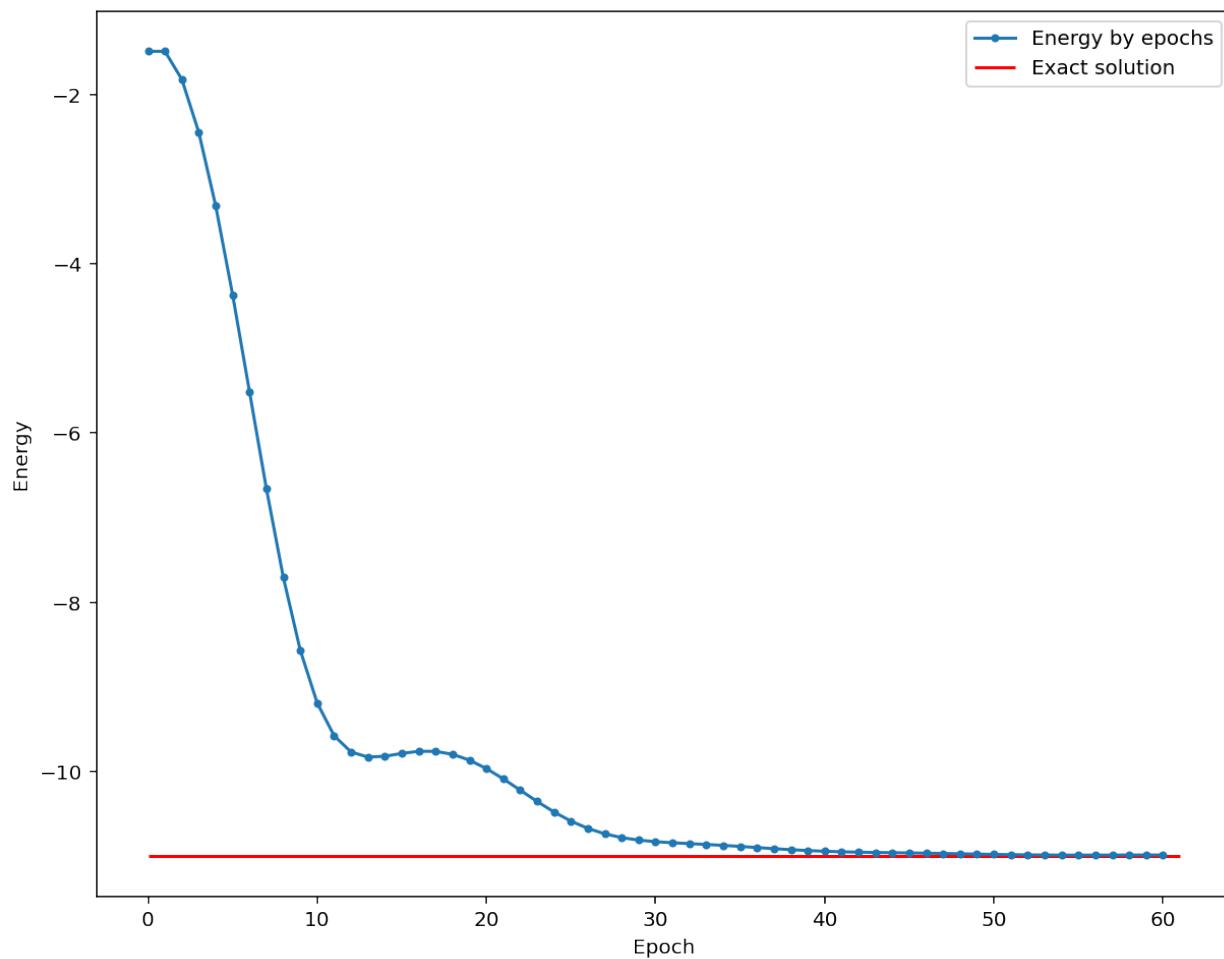
opt = qml.MomentumOptimizer()

energy = [
    feed_forward(params),
]

for epoch in range(60):
    params, e = opt.step_and_cost(feed_forward, params)
    energy.append(e)

plt.figure(figsize=(10, 8))
plt.plot(np.arange(len(energy)), energy, ".-", label="Energy by epochs")
plt.hlines(
    exact,
    0,
    len(energy),
    color="red",
    label="Exact solution",
)
plt.xlabel("Epoch")
plt.ylabel("Energy")
plt.legend()
plt.show()

```



66.4 Заключение

В этой лекции познакомились с **VQE** – очень перспективным, красивым и мощным алгоритмом для решения задач о собственных значениях на квантовых компьютерах. В следующей лекции более детально посмотрим на реализацию этого алгоритма в коде, а также обсудим выбранный *ansatz*.

ПРАКТИКА VQE

Эта лекция будет посвящена практике с **VQE**, а именно решению задачи коммивояжера. Будет много кода и мало теории. Рекомендуется заранее повторить следующие лекции:

- Описание задачи коммивояжера в лекции про задачи комбинаторной оптимизации.
- Сведение этой задачи к модели **Изинга** в лекции о переходе от комбинаторных задач к квантовым алгоритмам.

Нам потребуется определить бинарный оператор $\sigma^{z_{mod}}$, который активно **использовали** при определении гамильтониана. Нужно будет расписать полный оператор, отвечающий данной задаче и нужен случайный взвешенный граф, по которому будет “путешествовать” коммивояжер. Начнем с импорта библиотек и генерации графа.

67.1 Импорт библиотек

Для графов будем использовать структуры данных из уже знакомой нам библиотеки **NetworkX**, а для работы с разреженными операторами **SciPy**.

```
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = "retina"

import numpy as np
import networkx as nx
from scipy import sparse
from scipy.sparse import linalg as sl
```

67.2 Случайный взвешенный граф

Случайный взвешенный граф проще всего сгенерировать из случайной матрицы весов. Давайте сделаем такую:

Note: Тут будем работать с матрицей полного графа, то есть такого, в котором каждая вершина связана с каждой. Это вполне корректное допущение для задачи коммивояжера, так как, например, из каждого города РФ обычно можно попасть во все другие, пусть и за разное время в пути.

```
np.random.seed(42)
rand_mat = np.random.randint(0, 100, (4, 4))
```

Только эта матрица не является симметричной и имеет ненулевые элементы на диагонали. Элементы на диагонали означают петли (loop или self-loop), что не очень разумно в терминах задачи коммивояжера: не ясно, что значит, например, путь из Москвы в Москву. Аналогично и про то, что матрица не симметричная: если путь из Москвы в Санкт-Петербург займет 6 часов, то и путь обратно тоже займет столько же. Так что хотим симметричную матрицу. Давайте это исправим:

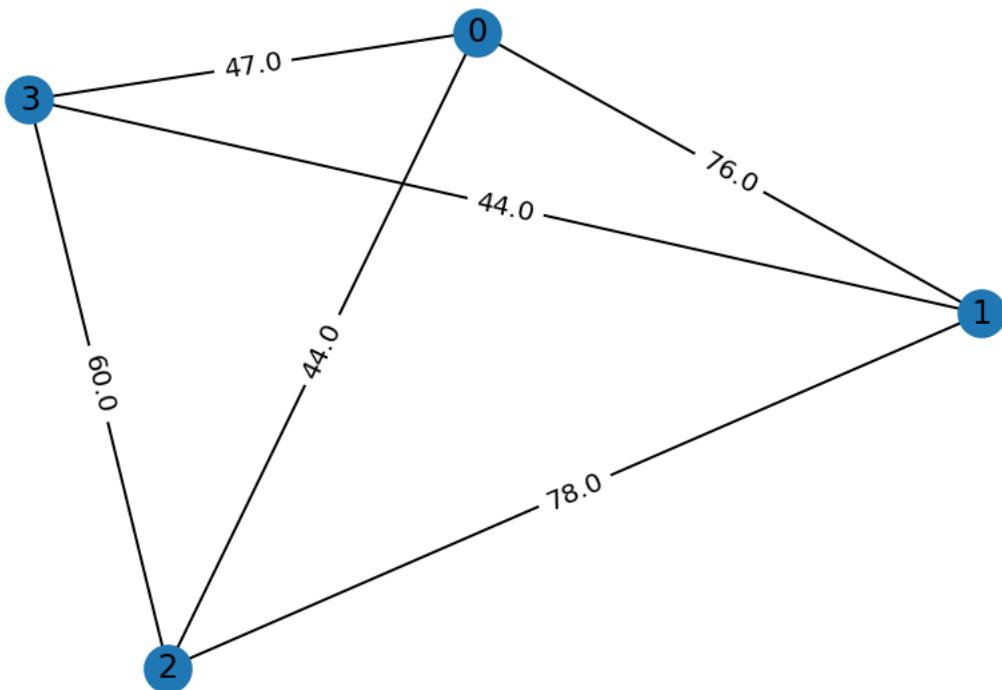
```
rand_adj = (rand_mat + rand_mat.T) / 2
np.fill_diagonal(rand_adj, 0)
```

Наконец, сделаем из этого граф и визуализируем его:

```
g = nx.Graph(rand_adj)

pos = nx.drawing.layout.random_layout(g, seed=42)
nx.draw(g, pos=pos)
edge_weights = nx.get_edge_attributes(g, "weight")
nx.draw_networkx_edge_labels(g, pos=pos, edge_labels=edge_weights)
nx.draw_networkx_labels(g, pos=pos)

plt.show()
```



67.3 Решение задачи TSP в NetworkX

Для того, чтобы потом понять, что все правильно сделали, давайте найдем приближенное решение задачи средствами NetworkX.

```
nx_solution = nx.algorithms.approximation.traveling_salesman_problem(g)
print(f"NetworkX solution (nodes order): {nx_solution}")

nx_solution_cost = 0
for i in range(len(nx_solution) - 1):
    nx_solution_cost += g.get_edge_data(nx_solution[i], nx_solution[i + 1])["weight"]

print(f"NetworkX solution (total cost): {nx_solution_cost}")
```

```
NetworkX solution (nodes order): [0, 3, 1, 2, 0]
NetworkX solution (total cost): 213.0
```

67.4 Матрица σ_{mod}^z

Это бинарный оператор, который ранее [вводили](#) в лекции о представлении комбинаторных задач в виде моделей Изинга. Соотношение между σ_{mod}^z и σ^z таково:

$$\hat{\sigma}_{mod}^z = \frac{1 + \hat{\sigma}^z}{2}$$

```
def sigma_z_mod(k: int, size: int) -> sparse.csr_matrix:
    sz_mod = sparse.coo_matrix(
        [
            [1, 0],
            [0, 0]
        ]
    )

    left_ith = sparse.eye(2 ** k, format="coo")
    right_ith = sparse.eye(2 ** (size - k - 1), format="coo")

    return sparse.kron(
        sparse.kron(left_ith, sz_mod),
        right_ith,
    )
```

Для произвольного состояния ψ справедливо

$$\hat{\sigma}_{mod}^z \psi = \frac{1}{2} \psi + \frac{1}{2} \hat{\sigma}^z \psi$$

Пусть ϕ – собственный вектор (собственное состояние) оператора $\hat{\sigma}^z$, соответствующий собственному значению λ :

$$\hat{\sigma}^z \phi = \lambda \phi$$

Отсюда следует, что

$$\hat{\sigma}_{mod}^z \phi = \frac{1}{2} \phi + \frac{1}{2} \lambda \phi = \frac{1 + \lambda}{2} \phi$$

Значит,

- собственные векторы $\hat{\sigma}_{mod}^z$ и $\hat{\sigma}^z$ совпадают;
- соответствующие собственные значения связаны соотношением $\lambda_{mod} = \frac{1+\lambda}{2}$.

Поскольку $\lambda \in \{-1, +1\}$, получаем $\lambda_{mod} \in \{0, 1\}$. На всякий случай проверим это, создав оператор маленькой размерности и вычислив его собственные значения:

```
eigs = np.linalg.eig(sigma_z_mod(1, 3).toarray())
print(np.unique(eigs[0]))
```

```
[0. 1.]
```

Напомню, что у нас zero-based индексация, поэтому `sigma_z_mod(1, 3)` значит что-то типа “построй мне матрицу $\hat{\sigma}_{mod}^z$, действующую на второй спин (ну или второй кубит)”.

Легко убедиться, что

- для $\lambda_{mod} = 1$ собственный вектор представляется битовой строкой, в которой спин с индексом 1 имеет значение 0;
- для $\lambda_{mod} = 0$ собственный вектор представляется битовой строкой, в которой спин с индексом 1 имеет значение 1.

```
def sigmaz_k(k: int, n: int) -> sparse.csr_matrix:
    left_part = sparse.eye(2 ** k)
    right_part = sparse.eye(2 ** (n - 1 - k))

    return sparse.kron(
        sparse.kron(
            left_part,
            sparse.csr_matrix(np.array([[1, 0], [0, -1]])))
        ,
        right_part
    )

def probs2bit_str(probs: np.array) -> str:
    size = int(np.log2(probs.shape[0]))
    bit_s_num = np.where(probs == probs.max())[0][0]
    s = f"{bit_s_num:b}"
    s = "0" * (size - len(s)) + s

    return s

for i, psi in enumerate(eigs[1]):
    probs = psi * psi.conj()
    bit_s = probs2bit_str(probs)
    print(f"Eigenvalue: {eigs[0][i]}\tBit string: {bit_s}")
```

```
Eigenvalue: 1.0      Bit string: 000
Eigenvalue: 1.0      Bit string: 001
Eigenvalue: 0.0      Bit string: 010
Eigenvalue: 0.0      Bit string: 011
Eigenvalue: 1.0      Bit string: 100
Eigenvalue: 1.0      Bit string: 101
Eigenvalue: 0.0      Bit string: 110
Eigenvalue: 0.0      Bit string: 111
```

Убедимся также в совпадении собственных векторов с матрицей σ^z :

Note: Для чистых состояний без запутывания матрицы собственных векторов могут быть приведены к виду единичных матриц. И действительно, каждый собственный вектор отвечает строго одной конфигурации спинов, а значит его волновая функция имеет вид $\Psi = 0, 0, \dots, 1, \dots, 0, 0$, где 1 соответствует как раз номеру той битовой строки, которая отвечает этой конфигурации.

```
print(f"Sigma z_mod:\n{eigs[1]}")
print(f"\nSigma z:\n{np.linalg.eig(sigmax_k(1, 3).toarray())[1]}")
```

```
Sigma z_mod:
[[1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]]
```

```
Sigma z:
[[1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1.]]
```

Note: В общем случае, даже если у двух операторов совпадают собственные вектора, то их порядок (по возрастанию собственных значений) совпадать совсем не обязан. В этом легко убедиться, например, если поменять местами $\hat{1}$ и $\hat{\sigma}^z$ в $\sigma^{z_{mod}}$. Поэтому, чтобы избежать ошибок в будущем, лучше не привыкать сравнивать матрицы собственных векторов при помощи `np.allclose`, а сравнивать их глазами построчно.

67.5 Гамильтониан задачи TSP

Давайте еще раз выпишем полный гамильтониан системы. Напомню, что задача будет параметризована при помощи N^2 кубит, которые для удобства выстроены в виде матрицы, где i -я строка соответствует i -му шагу в цикле по графу, а j -й элемент строки соответствует номеру вершины и является бинарным оператором:

$$\hat{H} = a \sum_{i=1}^N \left(1 - \sum_{j=1}^N \sigma_{i,j}^{z_{mod}} \right)^2 + a \sum_{j=1}^N \left(1 - \sum_{i=1}^N \sigma_{i,j}^{z_{mod}} \right)^2 + a \sum_{u,v \notin E} \sum_{j=1}^{N-1} \sigma_{u,j}^{z_{mod}} \sigma_{v,j+1}^{z_{mod}} + b \sum_{u,v \in E} \sum_{j=1}^{N-1} \sigma_{u,j}^{z_{mod}} \sigma_{v,j+1}^{z_{mod}}$$

Note: Еще раз вспомним, что первые два слагаемых “говорят”: каждая вершина должна входить только в один столбец, а каждая строка должна содержать только одну вершину. Третье слагаемое представляет собой штраф за соединение вершин, между которыми нет ребра, а четвертое дает стоимость всего пути. Первые три слагаемых идут с большим коэффициентом, чем последнее, так как нам важно в первую очередь получить реальное решение, а не получить самое дешевое.

Сделаем финт ушами: будем всегда считать, что путь коммивояжера начинается в вершине с индексом 0. Так как путь коммивояжера это гамильтонов цикл, то делаем это упрощение без какой-либо потери общности, так как вершина с индексом 0 все равно должна будет войти в путь. При этом такой хитрый трюк позволит снизить число необходимых кубит с N^2 до $(N - 1)^2$, что довольно существенно (хотя асимптотическая сложность остается такой же). Работать с графом будем при помощи уже знакомой библиотеки NetworkX:

```
def convert_tsp_to_ising(g: nx.Graph) -> sparse.coo_matrix:
    # Имеется два элемента гамильтониана:
    # - часть, дающая нам Гамильтонов цикл
    # - часть, считающая стоимость пути
    # Для нас важно, чтобы вес первой части превалировал.
    # Поэтому, в качестве коэффициента берем
    # пятикратное значение максимального веса ребер в графе.

    a = 5 * max([e[2]["weight"] for e in g.edges(data=True)])
    size = g.number_of_nodes() - 1

    H = sparse.coo_matrix((2 ** (size ** 2), 2 ** (size ** 2)), dtype=np.complex128)

    # Вычисляем сумму по колонкам:
    # - каждая вершина должна быть в каждой колонке строго один раз
    for i in range(size):
        # Считаем выражение 1 - sum
        col_sum = sparse.eye(2 ** (size ** 2), format="coo")
        for j in range(size):
            col_sum -= sigma_z_mod(i + j * size, size ** 2)

        # Возводим в квадрат и добавляем к гамильтониану
        H += a * col_sum * col_sum

    # Вычисляем сумму по строкам:
    # - каждая вершина должна быть в каждой строке строго один раз
    for j in range(size):
        row_sum = sparse.eye(2 ** (size ** 2), format="coo")
        for i in range(size):
            row_sum -= sigma_z_mod(i + j * size, size ** 2)

        # Аналогично, в квадрат и добавляем
        H += a * row_sum * row_sum

    # Считаем сумму по тем ребрам, которых в графе нет и тем, которые есть.
    for i in range(size):
        for j in range(size):
            if g.has_edge(i + 1, j + 1):
                # Сумма по всем ребрам графа
                coef = g.get_edge_data(i + 1, j + 1)["weight"]
            else:
                # Сумма по всем отсутствующим ребрам графа
                coef = a

            for k in range(size - 1):
                # Дальше просто используем полученный коэффициент и считаем сумму
                H += (
                    coef
                    * sigma_z_mod(i + k * size, size ** 2)
                    * sigma_z_mod(j + (k + 1) * size, size ** 2)
                )
```

(continues on next page)

(continued from previous page)

```

H += (
    coef
    * sigma_z_mod(i + (size - 1) * size, size ** 2)
    * sigma_z_mod(j, size ** 2)
)

# Начинаем всегда из нулевой вершины, это позволяет снизить число необходимых кубитов.
# Теперь должны добавить в гамильтониан слагаемое, отвечающее за нулевую вершину в пути.
for j in range(size):
    if g.has_edge(0, j + 1):
        coef = g.get_edge_data(0, j + 1)["weight"]
    else:
        coef = a

    H += coef * sigma_z_mod(j, size ** 2)
    H += coef * sigma_z_mod(j + (size - 1) * size, size ** 2)

# PennyLane работает с матрицами только в координатном формате, поэтому сразу делаем приведение.
return sparse.coo_matrix(H)

```

Убедимся, что точное решение даст тот же путь (с той же стоимостью), что и решение из NetworkX:

```

op = convert_tsp_to_ising(g)
scipy_solution = sl.eigs(op, k=1, which="SR", return_eigenvectors=True)
print(f"Ground state energy: {scipy_solution[0][0]:.4f}")

probs = scipy_solution[1] * scipy_solution[1].conj()
bit_s = probs2bit_str(probs)

# Нулевая вершина всегда первая
scipy_path = [0,]

for i in range(3):
    k = 0
    for j in range(3):
        if bit_s[j + i * 3] == "0":
            k = j + 1
    scipy_path.append(k)
scipy_path.append(0)

scipy_solution_cost = 0
for i in range(len(scipy_path) - 1):
    scipy_solution_cost += g.get_edge_data(scipy_path[i], scipy_path[i + 1])["weight"]

print(f"SciPy solution (nodes order): {scipy_path}")
print(f"SciPy solution (total cost): {scipy_solution_cost}")

```

```

Ground state energy: 273.0000-0.0000j
SciPy solution (nodes order): [0, 2, 1, 3, 0]
SciPy solution (total cost): 213.0

```

Видим, что решение через матрицу совпадает с тем, что выдал NetworkX (с точностью до инверсии: [0, 2, 1, 3, 0] и [0, 3, 1, 2, 0]), как и общая стоимость пути. Энергия основного состояния отличается по причине того, что использовали оптимизацию и магию с коэффициентами. Но это не так важно: по энергиям

важно лишь отношение порядка, потому что зная битовую строку всегда можем проверить стоимость пути за линейное время!

Теперь можно наконец-то, приступать к симуляции квантовых вычислений!

67.6 VQE на Pennylane

Для начала давайте преобразуем матрицу к тому объекту, с которым умеет работать квантовый фреймворк.

Note: Pennylane поддерживает также работу с представлением гамильтонианов в виде так называемых Паули-строк, но тут не будем еще раз определять весь гамильтониан, а просто перериспользуем разреженную матрицу.

Веса графа имеют разброс значений от 0 до 100. Из-за этого, при некоторых конфигурациях можем получать очень большое собственное значение гамильтониана, вплоть до нескольких тысяч. Это не очень хорошо для градиентного спуска методом `paramter-shift`, поэтому для численной устойчивости давайте умножим все значения весов на 0.1. Это сделает градиенты не слишком большими и улучшит численную устойчивость. Делать это можно совершенно спокойно, так как не критична численная величина минимального собственного, а важно лишь отношение порядка собственных значений.

```
import pennylane as qml

numer_stability_const = 0.1
ham = qml.SparseHamiltonian(op * numer_stability_const, wires=list(range(9)))
```

Далее необходимо выбрать *ansatz*. Современные квантовые процессоры обычно имеют вид решетки, типа такого:

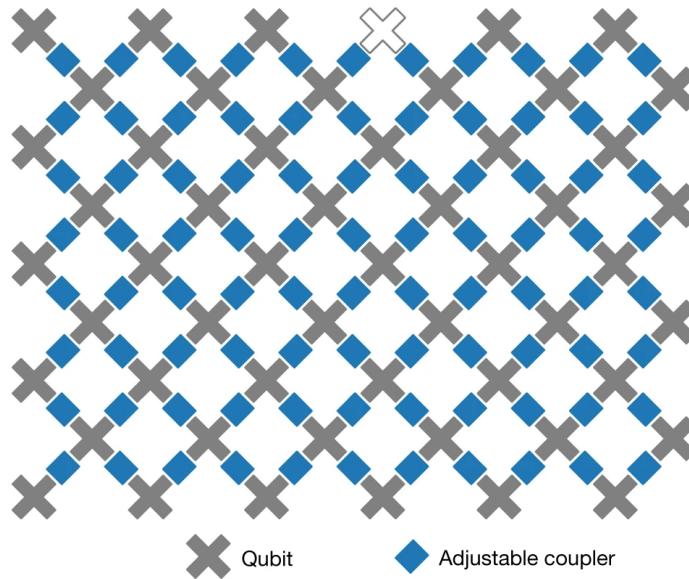


Fig. 67.1: Схематичное изображение квантового компьютера *Sycamore* от компании *Google* [AAB+19]

Поэтому, в идеале, хотелось бы делать либо однокубитные операции, либо двухкубитные операции, но над “соседними” кубитами. Так появилась идея *hardware efficient ansatz* [KMT+17]. Он заключается в том, что формируется несколько “слоев” VQC, где каждый такой слой содержит несколько операций вращений, а также попарные “запутывающие” операции (например, *CZ*, или гейт *CNOT*). Выглядит это примерно так:

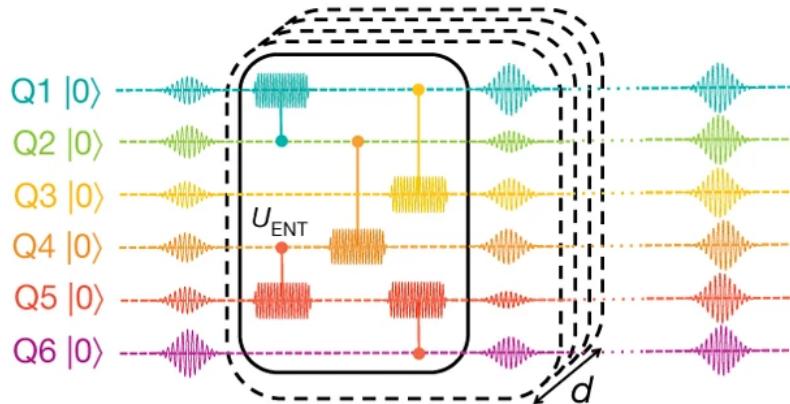


Fig. 67.2: Иллюстрация *hardware efficient ansatz* (*HWEA*) из [KMT+17]

Давайте реализуем такой слой:

```
size = len(ham.wires)

def layer(params):
    for i in range(size):
        qml.RY(params[i][0], wires=i)
        qml.RZ(params[i][1], wires=i)

    for i in range(size - 1):
        qml.CZ(wires=[i, i + 1])
```

Здесь специально не повторяем основ параметризованных квантовых схем. Для тех, кто забыл, что это такое, рекомендуем вернуться к *отдельной лекции* по этой теме.

Теперь создадим устройство, которое будет квантовым симулятором, а также напишем код, который будет на этом устройстве запускаться. Такой код должен последовательно применить несколько *HWEA* слоев, после чего измерить ожидаемое значение гамильтониана в полученном состоянии.

```
device = qml.device("default.qubit", wires=size)
num_layers = 3

@qml.qnode(device, diff_method="parameter-shift")
def feed_forward(params, wires=size):
    for k in range(3):
        layer(params[k])

    for i in range(size):
        qml.RY(params[-1][i][0], wires=i)
        qml.RZ(params[-1][i][1], wires=i)

    return qml.expval(ham)
```

Теперь надо сгенерировать параметры и выбрать метод их оптимизации. В этот раз давайте воспользуемся алгоритмом Momentum, так называемым “ускоренным” градиентным спуском:

```
opt = qml.MomentumOptimizer()
params = tuple(
    tuple(
```

(continues on next page)

(continued from previous page)

```

(
    np.random.rand() * 2 * np.pi,
    np.random.rand() * 2 * np.pi,
)
for _ in range(size)
)
for _ in range(num_layers + 1)
)

```

Ну и наконец можно приступать к оптимизации параметров. Все примерно также, как в *базовой лекции про градиенты*. То есть ничего нового тут по сути не делается.

67.7 Симуляция

Warning: Осторожно, следующий код работает несколько минут, так как обучаем довольно тяжелую схему на 9-и кубитах! Кстати, при желании можно добиться гораздо больше точности по энергии относительно точного решения. Для этого достаточно установить число эпох ~ 150 . Но поскольку этот код исполняется при сборке данных лекций, то мы не можем себе позволить считать отдельные страницы по 7-10 минут. Читателю рекомендуется самостоятельно определить то число эпох, которого достаточно, чтобы получить точность в 3-м знаке после запятой.

```

energy = [
    feed_forward(params),
]

for epoch in range(85):
    params, e = opt.step_and_cost(feed_forward, params)
    energy.append(e)

    if epoch % 5 == 0:
        print(f"epoch: {epoch}\tenergy: {e:.6f}")

```

epoch: 0 energy: 360.638506

epoch: 5 energy: 163.172963

epoch: 10 energy: 96.375950

epoch: 15 energy: 56.524352

epoch: 20 energy: 43.497589

epoch: 25 energy: 35.938162

epoch: 30 energy: 31.882151

epoch: 35 energy: 29.457526

```
epoch: 40          energy: 29.232270
```

```
epoch: 45          energy: 28.058306
```

```
epoch: 50          energy: 27.814023
```

```
epoch: 55          energy: 27.495886
```

```
epoch: 60          energy: 27.479515
```

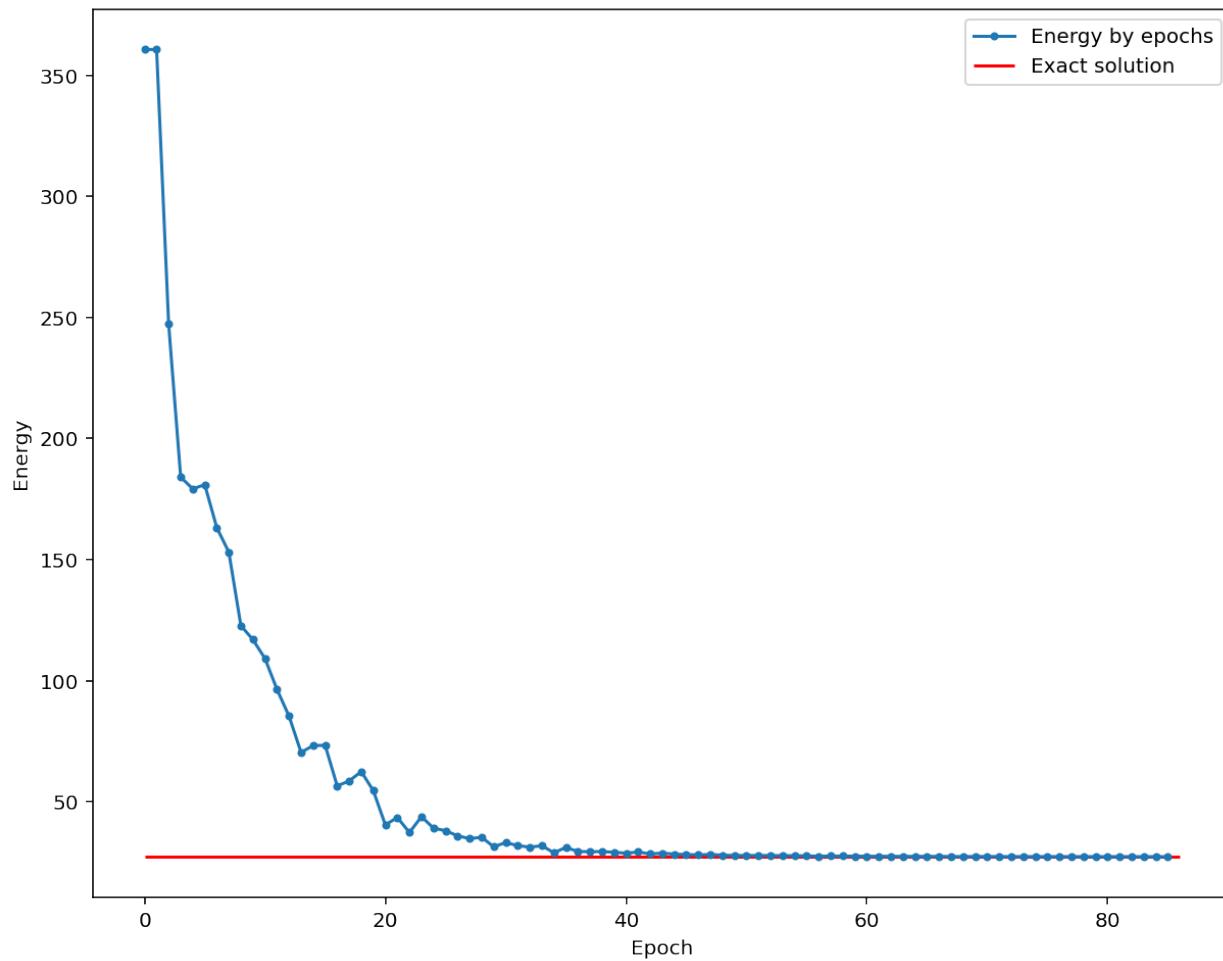
```
epoch: 65          energy: 27.411348
```

```
epoch: 70          energy: 27.348231
```

```
epoch: 75          energy: 27.329692
```

```
epoch: 80          energy: 27.324815
```

```
plt.figure(figsize=(10, 8))
plt.plot(np.arange(len(energy)), energy, ".-", label="Energy by epochs")
plt.hlines(
    np.real(scipy_solution[0][0]) * numer_stability_const,
    0,
    len(energy),
    color="red",
    label="Exact solution",
)
plt.xlabel("Epoch")
plt.ylabel("Energy")
plt.legend()
plt.show()
```



Видим, что VQE отлично сходится к основному состоянию, которое знаем из SciPy и ARPACK. Теперь давайте убедимся, что квантовый путь совпадет с тем, что получили из NetworkX. Для этого нужно получить распределение вероятностей битовых строк. На реальном компьютере это можно сделать просто измеряя много раз, но на симуляторе есть возможность получить его сразу одной командой.

```
@qml.qnode(device, diff_method="parameter-shift")
def answer(params, wires=size):
    for k in range(num_layers):
        layer(params[k])

    for i in range(size):
        qml.RY(params[-1][i][0], wires=i)
        qml.RZ(params[-1][i][1], wires=i)

    return qml.probs(wires=list(range(size)))

probs = answer(params)
bit_s = probs2bit_str(probs)

# Нулевая вершина всегда первая
q_path = [0, ]
```

(continues on next page)

(continued from previous page)

```
for i in range(3):
    k = 0
    for j in range(3):
        if bit_s[j + i * 3] == "0":
            k = j + 1
    q_path.append(k)
q_path.append(0)

q_solution_cost = 0
for i in range(len(q_path) - 1):
    q_solution_cost += g.get_edge_data(q_path[i], q_path[i + 1])["weight"]

print(f"Quantum solution (nodes order): {q_path}")
print(f"Quantum solution (total cost): {q_solution_cost}")
```

```
Quantum solution (nodes order): [0, 2, 1, 3, 0]
Quantum solution (total cost): 213.0
```

Ура, получился тот же ответ, что и в SciPy/NetworkX. Значит самописный **VQE** работает правильно.

67.8 Заключение

В этой лекции попрактиковались в решение реальных задач при помощи алгоритма **VQE**.

CHAPTER
SIXTYEIGHT

О БЛОКЕ “D-WAVE – КВАНТОВЫЙ ОТЖИГ И СЕМПЛИРОВАНИЕ”

Этот блок посвящен квантовому отжигателю компании D-Wave и задачам, которые можно решать с его помощью.

D-WAVE: КВАНТОВЫЙ ОТЖИГ, ОПТИМИЗАЦИЯ И СЕМПЛИНГ

69.1 Описание лекции

В этой лекции рассмотрим квантовый отжигатель компании D-Wave и задачи, которые можно решать с его помощью. Заострим основное внимание на физике и алгоритмах, нежели на hardware-реализации этого устройства. Эта лекция основана на [официальной документации D-Wave](#).

69.2 Введение в квантовый отжиг

На сегодняшний день, квантовый отжиг представляет собой отдельную модель квантовых вычислений, не связанную с так называемыми Gate-Model квантовыми вычислениями. Эта парадигма позволяет эффективно решать задачи комбинаторной оптимизации больших масштабов, представляющие большой интерес для многих областей науки и индустрии. Вкратце, в модели квантового отжига на физическую модель (коей является массив кубитов) отображаются параметры решаемой задачи, после чего кубиты адиабатически (достаточно медленно) переводятся в основное состояние, которое и кодирует в себе искомое решение. Давайте теперь разберем подробнее данный подход.

69.3 Задачи оптимизации и семплирования

Квантовый отжигатель (Quantum Annealing Processor) в силу своей физической реализации приводит к решениям, соответствующим низкоэнергетическим состояниям физической системы. В то время как задачи оптимизации (Optimization Problems) требуют глобального минимума энергии, иногда хватает и семплов из состояний, близких к основным. Такие задачи называются задачами семплирования (Probabilistic Sampling Problems).

Классическим примером задачи оптимизации в контексте квантового отжига является, например, [задача коммивояжера](#). Квантовая физика, в силу своей природы, может помочь решать такого рода задачи, так как они могут быть переформулированы в задачи дискретной минимизации энергии. Одно из основных правил физики состоит в том, что все физические системы стремятся к состояниям с минимальной энергией. Например, объекты скользят с холмов, горячие вещи со временем остывают и так далее. Этот принцип соблюдается и в мире квантовой физики. Квантовый отжиг просто использует квантовую физику для поиска низкоэнергетических состояний системы и, как следствие, оптимального или почти оптимального решения.

В свою очередь, задачи семплирования, часто являющиеся вычислительно сложными (например, задача семплирования из ограниченной машины Больцмана), идеально подходят к квантовым отжигателям в случае когда необходимо построить вероятностную модель, а доступ к полному вероятностному распределению состояний не представляется возможным. Такая постановка задачи подходит например, чтобы обучить модель генерировать изображения рукописных цифр из знаменитого датасета MNIST. По этой теме можно послушать

[доклад](#) Никиты Щуцкого “Quantum-Classical Neural Networks in Computational Drug Design”, сделанный в рамках трека ODS по квантовым вычислениям.

69.4 Квантовая физика под капотом

Как помним, квантовый гамильтониан является оператором, ставящим в соответствие собственным состояниям системы ее энергии. Множество собственных чисел гамильтониана называется его спектром. В квантовом отжигателе D-Wave гамильтониан имеет следующую форму:

$$\mathcal{H}_{Ising} = \underbrace{-A(t) \left(\sum_i \hat{\sigma}_x^{(i)} \right)}_{\text{Initial Hamiltonian}} + \underbrace{B(t) \left(\sum_i h_i \hat{\sigma}_z^{(i)} + \sum_{i,j} J_{i,j} \hat{\sigma}_z^{(i)} \hat{\sigma}_z^{(j)} \right)}_{\text{Final Hamiltonian}},$$

где $A(t), B(t)$ являются функциями от времени $t \in [0, T]$, $\hat{\sigma}_x^{(i)}$ – операторами Паули, действующими на кубиты, а h_i и $J_{i,j}$ – локальными полями и взаимодействиями кубитов соответственно. Физически, вектор h соответствует внешнему магнитному полю, применимому к каждому кубиту. В свою очередь, матрица J отвечает за попарные ZZ взаимодействия кубитов. Функции $A(t), B(t)$ должны быть такими, что $A(0) = B(T) = 1, A(T) = B(0) = 0$, то есть гамильтониан со временем полностью переходит из $\mathcal{H}_{Initial}$ в \mathcal{H}_{Final} . Вид этих функций называется расписанием отжига или annealing schedule.

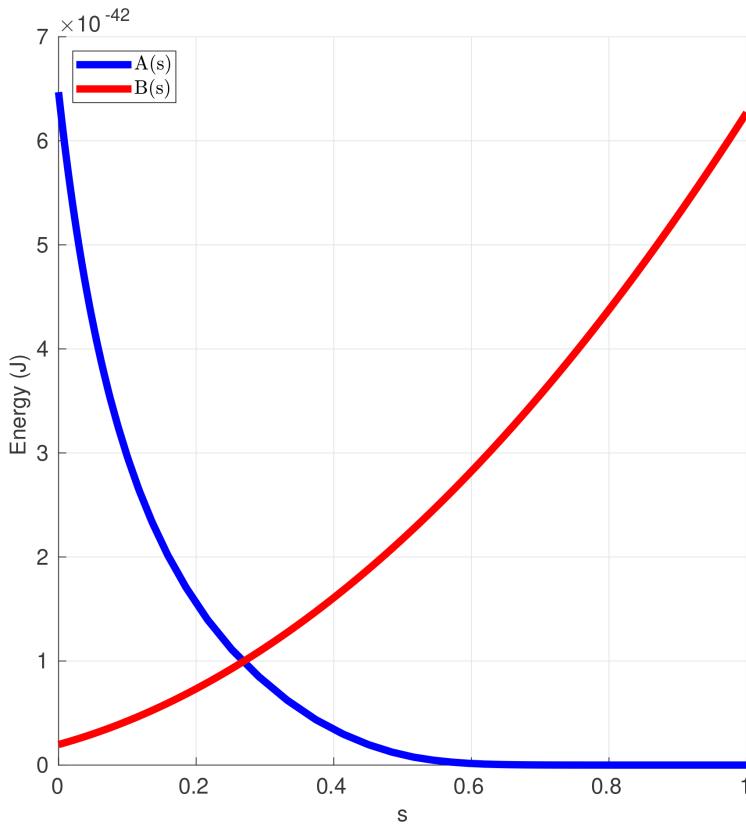


Fig. 69.1: Пример расписания отжига: функций $A(t), B(t)$.

Отметим также, что матрица взаимодействий кубитов $J_{i,j}$ ограничена графом взаимодействий кубитов, который

обычно фиксирован. Для практического применения в коде детальное представление этого графа знать не нужно, но ознакомиться с ним можно на специально посвященной этому странице документации D-Wave.

Разберем подробно, что же представляет собой данный гамильтониан. Начальный гамильтониан $\mathcal{H}_{Initial}$ представляет собой гамильтониан с основным состоянием состоящим из суперпозиции 0 и 1. Этот гамильтониан в литературе также называется tunneling Hamiltonian. Финальный гамильтониан \mathcal{H}_{Final} , в свою очередь, является гамильтонианом задачи (его называют также Problem Hamiltonian), чье основное состояние и представляет собой решение, параметры которого находятся в матрице взаимодействий кубитов $J_{i,j}$ и векторе локальных полей h_i . Обратим внимание на тот факт, что \mathcal{H}_{Final} – полностью классический гамильтониан, он диагонален в вычислительном базисе (σ_z). Каждое собственное (классическое) состояние этого гамильтониана соответствует классической функции потерь для решаемой задачи.

Таким образом, в квантовом отжиге система инициализируется в основном состоянии начального гамильтониана – такое состояние легко приготовить экспериментально, то есть система легко переходит в основное состояние начального гамильтониана в силу его простоты. По мере самого отжига, исходя из адиабатической теоремы, основное состояние начального гамильтониана переходит в основное состояние финального гамильтониана. Для наглядности, на графике ниже (взят из [этой](#) статьи) показано типичное поведение первых энергетических уровней гамильтониана. Основному состоянию соответствует кривая с наименьшей энергией.

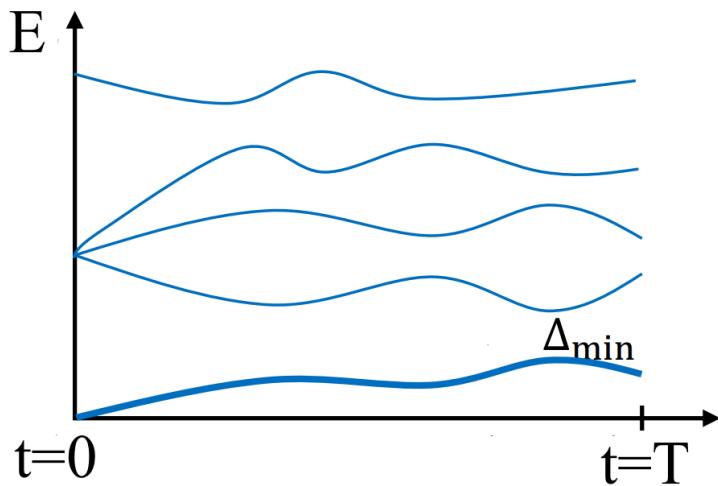


Fig. 69.2: Типичная зависимость от времени энергетических уровней гамильтонианов, используемых в квантовом отжиге

Очень важно отметить, что для успеха данного алгоритма необходимо выполнение условий адиабатической теоремы:

1. минимальная энергетическая щель Δ_{min} (разница энергий основного и первого возбужденного состояний) не должна обращаться в 0 в любой момент времени;
2. функции $A(t)$ и $B(t)$ должны меняться достаточно медленно.

Можно показать, что второе условие может быть переформулировано в виде $T \gg \frac{1}{\Delta_{min}^2}$. Именно это неравенство ответственно за наибольшую проблему, с которой можно столкнуться при практической реализации данного алгоритма. В частности, из него видно, что время алгоритма зависит от минимальной (за все время алгоритма) энергетической щели, которая не должна быть слишком маленькой, иначе время алгоритма станет слишком большим. Если же щель будет слишком большой (что соответствует быстрому отжигу), то система может перейти в возбужденное состояние. Однако, основные состояния начального и финального гамильтонианов являются, качественно, сильно разными. Основное состояние начального гамильтониана является тривиальным, в то время как основное состояние финального гамильтониана кодирует в себе решение сложной задачи оптимизации. Это наталкивает нас на мысль, что в течение процесса квантового отжига, система может пережить

квантовый фазовый переход.

Для квантовых фазовых переходов первого рода известно, что $\Delta_{\min} \propto e^{-N}$, где N – количество кубитов, что приводит к $T \propto e^{2N}$, то есть экспоненциально большому времени алгоритма. В этом случае задача считается “сложной”, так как алгоритм не масштабируется.

В свою очередь, для квантовых фазовых переходов второго рода верно, что $\Delta_{\min} \propto N^{-l}$ ($l > 0$), что приводит к $T \propto N^{2l}$, то есть полиномиальному времени алгоритма и задача считается “простой”.

69.5 Пример: задача раскраски графа

Давайте теперь попробуем решить с помощью квантового отжига следующую задачу комбинаторной оптимизации – [задачу раскраски графа](#), или, как она называется в англоязычной литературе – *graph coloring problem*. В этой задаче необходимо с помощью заданного числа цветов присвоить цвет каждой вершине графа так, чтобы любые две вершины, имеющие общее ребро, имели разные цвета. Назовем граф $G = (E, V)$, где E – множество ребер, а V – множество вершин. Для наглядности, рассмотрим эту задачу в виде раскраски карты Канады четырьмя цветами – необходимо каждой канадской провинции (аналогу штата в США) присвоить цвет так, чтобы все граничащие провинции имели разные цвета. Этот пример находится также в [документации D-Wave](#). Визуально, решение этой задачи будет выглядеть как на следующем графике.



Fig. 69.3: Раскраска карты Канады с помощью четырех цветов.

D-Wave предоставляют доступ к трем видам бэкендов: квантовому, классическому и гибридному. Так как курс посвящен квантовому машинному обучению, воспользуемся непосредственно квантовым отжигателем. Для этого необходимо получить токен, который генерируется в личном кабинете пользователя при регистрации в облачном сервисе D-Wave.

Импортируем нужные библиотеки.

```
import dwavebinarycsp
from dwave.system import DWaveSampler, EmbeddingComposite
import networkx as nx
import matplotlib.pyplot as plt
```

Отобразим задачу на графике, где провинции будут вершинами графа, а границы – его ребрами. Например, (AB, BC) – ребро, соответствующее границе между провинциями Альберта и Британская Колумбия. Обозначения провинций представлены в таблице ниже.

код	provинция	код	provинция	код	provинция
AB	Alberta	BC	British Columbia	MB	Manitoba
NB	New Brunswick	NL	Newfoundland and Labrador	NS	Nova Scotia
NT	Northwest Territories	NU	Nunavut	ON	Ontario
PE	Prince Edward Island	QC	Quebec	SK	Saskatchewan
YT	Yukon				

```
provinces = ["AB", "BC", "MB", "NB", "NL", "NS", "NT", "NU", "ON", "PE", "QC", "SK",
             "YT"]
neighbors = [("AB", "BC"), ("AB", "NT"), ("AB", "SK"), ("BC", "NT"), ("BC", "YT"),
             ("MB", "NU"), ("MB", "ON"), ("MB", "SK"), ("NB", "NS"), ("NB", "QC"),
             ("NL", "QC"), ("NT", "NU"), ("NT", "SK"), ("NT", "YT"), ("ON", "QC")]
```

Чтобы “объяснить” отжигателю эту задачу, необходимо сформулировать ее в виде задачи удовлетворения ограничений (Constraint Satisfaction Problem (CSP)). В свою очередь, библиотеки D-Wave умеют отображать ее на модель Изинга. Математически, это выглядит следующим образом: каждой провинции (вершине графа) сопоставляется 4 бинарные переменные (каждая отвечает за один из четырех цветов) $x_i \in \{0, 1\}$, где одна из переменных “выбирает” цвет и переключается в состояние $x_i = 1$, как и остальные три переменные остаются в состоянии 0.

Теперь займемся ограничениями. Следующая функция выражает ограничение того, что соседние вершины в графике не могут быть одного и того же цвета. Область ее определения и значения – булевые переменные.

```
def not_both_1(v: bool, u: bool) -> bool:
    return not (v and u)
```

Зададим конфигурации цветов.

```
one_color_configurations = {(0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0)}
colors = len(one_color_configurations)
```

Создадим объект класса CSP и зададим следующие ограничения:

- каждая провинция имеет лишь один цвет;
- каждая ее соседняя не может иметь такой же цвет.

```
csp = dwavebinarycsp.ConstraintSatisfactionProblem(dwavebinarycsp.BINARY)

for province in provinces:
    variables = [province+str(i) for i in range(colors)]
    csp.add_constraint(one_color_configurations, variables)

for neighbor in neighbors:
```

(continues on next page)

(continued from previous page)

```
v, u = neighbor
for i in range(colors):
    variables = [v+str(i), u+str(i)]
    csp.add_constraint(not_both_1, variables)
```

Конвертируем модель из CSP в BQM (Binary Quadratic Model) – бинарную квадратичную модель. Это модель Изинга, в которой значения спинов $\sigma_i = 0.5 * (x_i + 1)$. Таким образом, каждая провинция Канады и ее цвет будет кодироваться четырьмя спинами, один из которых направлен вверх (вниз), пока остальные направлены вниз (вверх).

```
bqm = dwavebinarycsp.stitch(csp)
```

Создадим семплер, передав в него API Token, создав прежде переменное окружение Dwave_TOKEN, к примеру, в терминале посредством export Dwave_TOKEN=insert_api_token:

```
import os

base_sampler = DWaveSampler(token=os.environ["Dwave_TOKEN"])
```

Дадим задание квантовому семплеру на 1000.

```
sampler = EmbeddingComposite(base_sampler)
sampleset = sampler.sample(bqm, num_reads=1000, label="Canada Map Coloring")
```

Проверим, что было найдено решение задачи,

```
sample = sampleset.first.sample
if not csp.check(sample):
    print("Failed to color map. Try sampling again.")
else:
    print(sample)
```

```
{'AB0': 0, 'AB1': 0, 'AB2': 1, 'AB3': 0, 'BC0': 1, 'BC1': 0, 'BC2': 0, 'BC3': 0, 'MB0':
↳ 0, 'MB1': 0, 'MB2': 1, 'MB3': 0, 'NB0': 1, 'NB1': 0, 'NB2': 0, 'NB3': 0, 'NL0': 0,
↳ 0, 'NL1': 1, 'NL2': 0, 'NL3': 0, 'NS0': 0, 'NS1': 0, 'NS2': 1, 'NS3': 0, 'NT0': 0,
↳ 0, 'NT1': 0, 'NT2': 0, 'NT3': 1, 'NU0': 1, 'NU1': 0, 'NU2': 0, 'NU3': 0, 'ON0': 1, 'ON1':
↳ 0, 'ON2': 0, 'ON3': 0, 'PE0': 0, 'PE1': 0, 'PE2': 0, 'PE3': 1, 'QC0': 0, 'QC1': 0,
↳ 0, 'QC2': 0, 'QC3': 1, 'SK0': 0, 'SK1': 1, 'SK2': 0, 'SK3': 0, 'YT0': 0, 'YT1': 1,
↳ 0, 'YT2': 0, 'YT3': 0}
```

Мы получили словарь sample, который будет выглядеть следующим образом: его ключи будут кодировать провинцию и цвет, а значения – бинарные переменные, означающие раскрашена ли провинция этим цветом. Этот словарь – семпл из состояния, полученного при решении данной задачи методом квантового отжига. Построим визуализацию решения в виде графа с раскрашенными вершинами.

```
from typing import List
from typing import Tuple
from typing import Dict

def plot_map(sample: Dict[str, bool], provinces: List[str], neighbors: List[Tuple[str]]) -> None:
    G = nx.Graph()
    G.add_nodes_from(provinces)
    G.add_edges_from(neighbors)
```

(continues on next page)

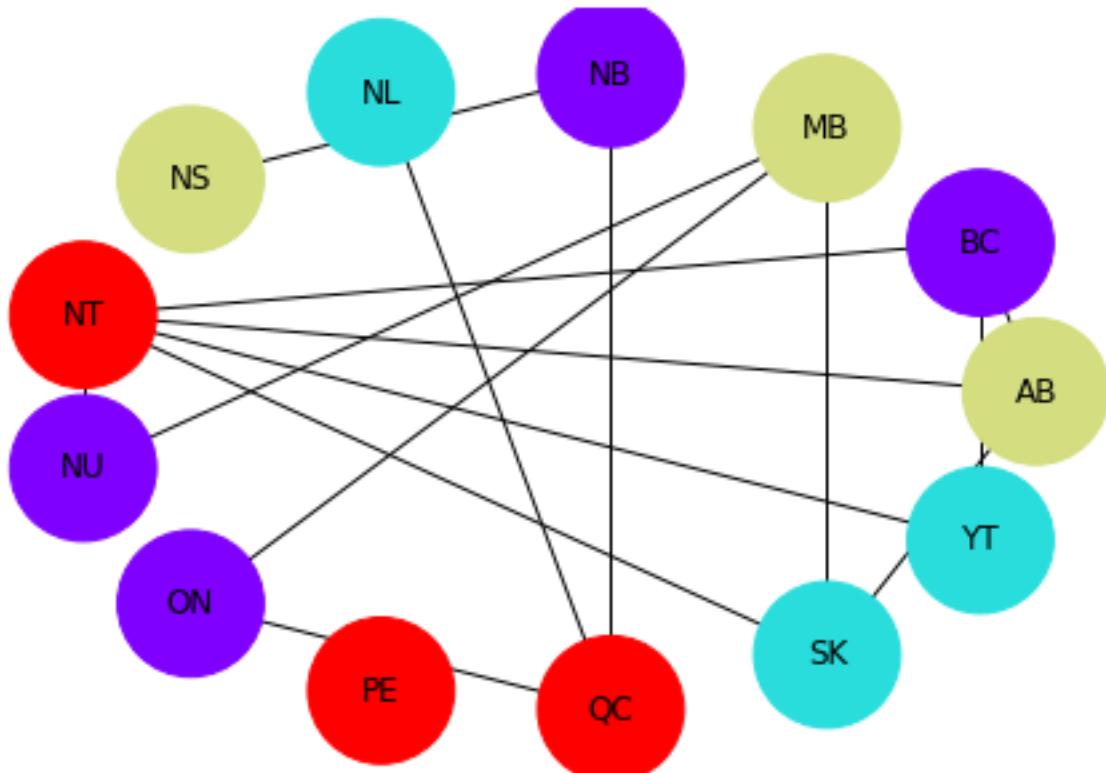
(continued from previous page)

```

color_map = {}
for province in provinces:
    for i in range(colors):
        if sample[province+str(i)]:
            color_map[province] = i
node_colors = [color_map.get(node) for node in G.nodes()]
nx.draw_circular(G, with_labels=True, node_color=node_colors, node_size=3000,
cmap=plt.cm.rainbow)
plt.show()

plot_map(sample, provinces, neighbors)

```



После запуска функции `plot_map`, получим изображение графа, в котором вершины будут соответствовать провинциям и цветам, а ребра – границам между провинциями. Вы можете сами убедиться, что ни одна провинция не будет иметь такой же цвет, как и ее сосед. Также отметим, что такая задача может иметь не одно решение, поэтому не нужно ожидать, что график будет получаться всегда одним и тем же.

Таким образом, сумели решить задачу раскраски графа с помощью квантового отжигателя.

69.6 Пример: задача электронной структуры молекулы

Поскольку в курсе также имеем дело с задачами квантовой химии, давайте разберем соответствующий пример из [TKTD20]. В данной статье авторы осуществляют точную диагонализацию физического гамильтониана с помощью квантового отжигателя D-Wave. Этот подход они называют QAE – Quantum Annealing Eigen-solver. Разберем эту статью гораздо менее подробно, чем предыдущую задачу.

Основная проблема применения квантового отжига в квантовой химии – найти отображение гамильтониана электронной структуры молекулы на гамильтониан Изинга (вспомним \mathcal{H}_{Final} из теории квантового отжига, описанной в начале этой лекции). Обычно такие отображения начинаются с записи электронного гамильтониана в [формализме вторичного квантования](#), а затем используют преобразования [Жордана-Вигнера](#) (Jordan–Wigner transformation) или Браво-Китаева (Bravyi–Kitaev transformation). Минусы такого подхода заключаются в том, что:

- растет число кубитов, необходимое для описания системы на языке (напомним, классического, а значит – диагонального) гамильтониана Изинга
- в гамильтониане появляются элементы вне диагонали (благодаря членам содержащим σ^x, σ^y).

Подход авторов к задаче состоит из двух частей. Первый шаг состоит из составления электронного гамильтониана в базисе слетеровских детерминантов, которые в свою очередь составлены из одноэлектронных самосогласованных [орбиталей Хартри-Фока](#).

В свою очередь, второй этап состоит из нахождения первых нескольких собственных векторов (волновых функций) и собственных значений (соответствующих энергий) этого гамильтониана. Под капотом, задача нахождения собственных значений и собственных векторов отображается на задачу квантового отжига. Это самая хитрая часть алгоритма. Это отображение построено на минимизации отношения Рэлея (Rayleigh–Ritz quotient, RRQ) $R_{\mathcal{H}} = \langle \psi | \mathcal{H} | \psi \rangle / \langle \psi | \psi \rangle$, где \mathcal{H} – электронный гамильтониан, а ψ – волновая функция. Соответствующая минимальному R волновая функция кодируется в бинарный вектор с помощью power-of-two scheme. Подробней про это можно почитать в статьях, на которые ссылается оригинальная работа. Примечательно, что метод минимизации отношения Рэлея соответствует вариационному методу в квантовой химии.

На следующей схеме (взята из оригинальной работы) изображен алгоритм QAE в общем виде. В нашей терминологии, A соответствует \mathcal{H} , v – ψ . $F(v) = \langle v, Av \rangle + \lambda \cdot \langle v, v \rangle$ – функция потерь в формате QUBO – Quantum Unconstrained Binary Optimization (функция потерь на языке QUBO получается, если в функции потерь на языке модели Изинга, каждую спиновую переменную $\sigma_i \in \{-1, 1\}$ преобразовать по закону $x_i = (\sigma_i + 1)/2$, $\sigma_i \in \{0, 1\}$), с классическим множителем лагранжа λ .

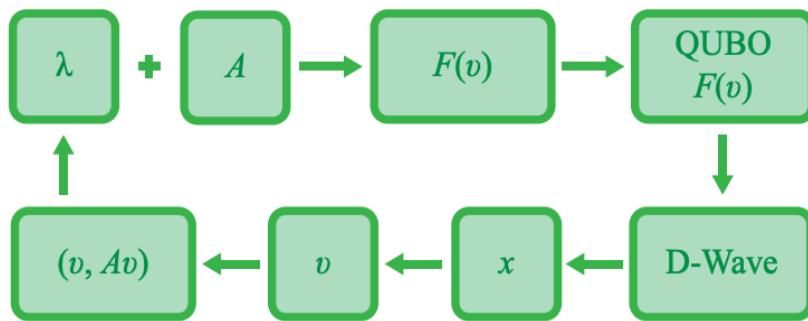


Fig. 69.4: Схема алгоритма QAE.

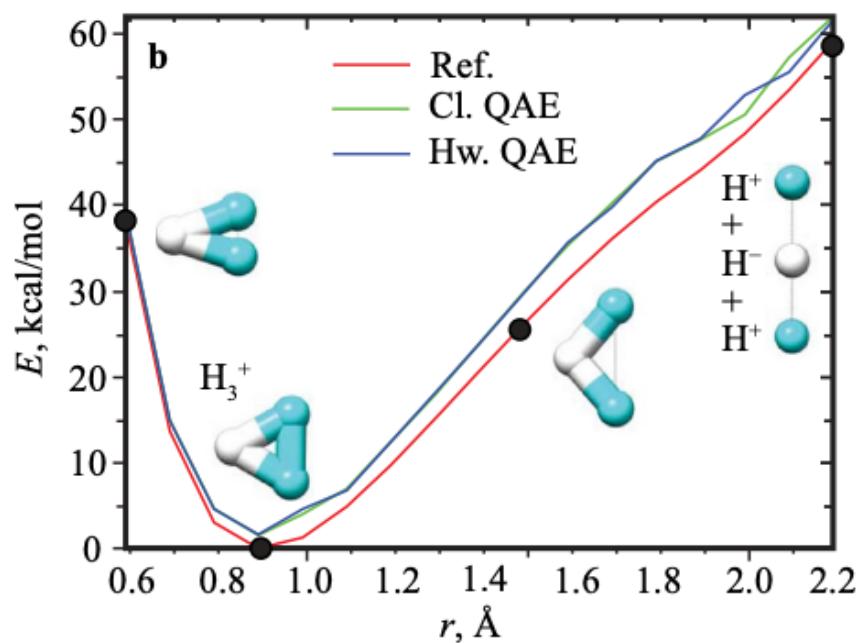


Fig. 69.5: Результаты алгоритма на примере молекулы H_3^+ .

CHAPTER
SEVENTY

О БЛОКЕ “QUANTUM APPROXIMATE OPTIMIZATION ALGORITHM”

В этом блоке узнаете об алгоритме квантовой приближенной оптимизации.

QUANTUM APPROXIMATE OPTIMIZATION ALGORITHM

71.1 Введение

В лекции рассматривается еще один алгоритм для приближенного решения *NP-задач комбинаторной оптимизации*, который называется **Quantum Approximate Optimization Algorithm** (далее **QAOA**) [FGG14].

71.1.1 Квантовый отжиг (повторение)

В прошлых лекциях активно *рассказывалось* о квантовых аннилерах (*отжигателях*, англ. *annealers*) – аналоговых устройствах, реализующих поиск основного состояния системы. Выпишем еще раз *гамильтониан*, который там используется:

$$\mathcal{H}_{Ising} = \underbrace{-A(t) \left(\sum_i \hat{\sigma}_x^{(i)} \right)}_{\text{Initial Hamiltonian}} + \underbrace{B(t) \left(\sum_i h_i \hat{\sigma}_z^{(i)} + \sum_{i,j} J_{i,j} \hat{\sigma}_z^{(i)} \hat{\sigma}_z^{(j)} \right)}_{\text{Final Hamiltonian}},$$

В процессе квантового отжига плавно меняются параметры A и B :

В той же лекции о D-Wave указывалась главная проблема – риск перехода системы из основного состояния в возбужденное при недостаточно медленном изменении параметров A и B :

Таким образом, большой проблемой для нас является выбор правильного “расписания” отжига, то есть зависимостей $A(t)$ и $B(t)$.

71.1.2 От железа к симуляции

У отжига “в железе” есть существенная проблема – трудно правильным образом подбирать расписание отжига. Но есть и другая идея! Изначально квантовые компьютеры создавались для симуляции квантовой динамики. Воспользуемся техникой симуляции, которая называется *trotterization*.

71.1.3 Trotterization и симуляция аннилера

В данном случае наша цель – получить выражение для финального состояния Ψ , которое будет отвечать решению квантового отжигателя. Финальное состояние есть решение уравнения Шредингера:

$$\Psi(t) = e^{-i\mathcal{H}_{Ising}(t)t} \Psi(0) = e^{-i(A(t)\mathcal{H}_{initial} + B(t)\mathcal{H}_{cost})t} \Psi(0),$$

где $\mathcal{H}_{initial}$ – это *tunneling* или начальный гамильтониан, а \mathcal{H}_{cost} – это так называемый *problem* или *cost* гамильтониан, который отвечает задаче. Подробно эти темы обсуждались в *лекции о D-Wave*. Для тех, кто

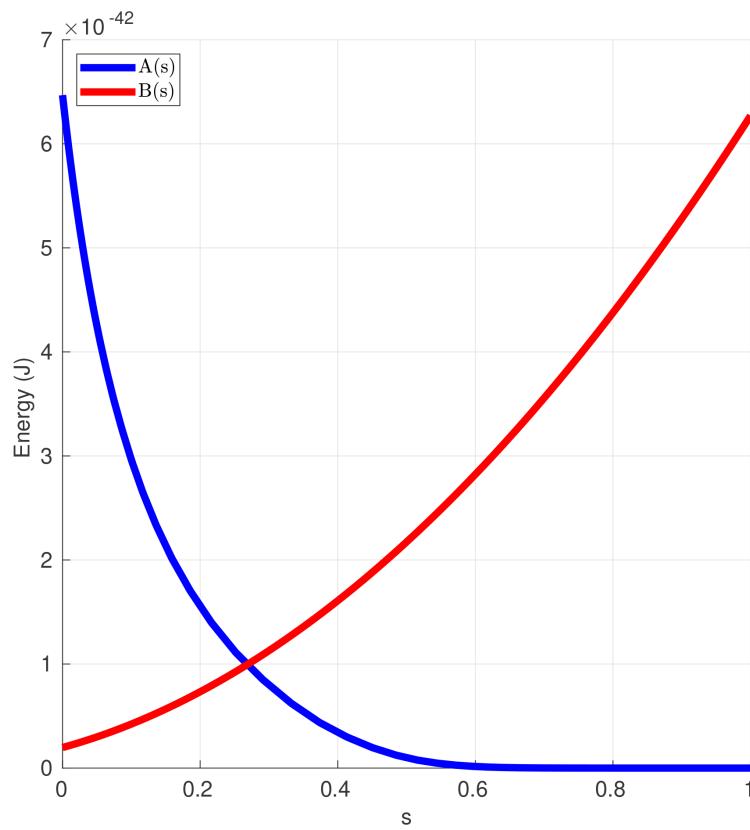


Fig. 71.1: Пример расписания отжига: функций $A(t)$, $B(t)$.

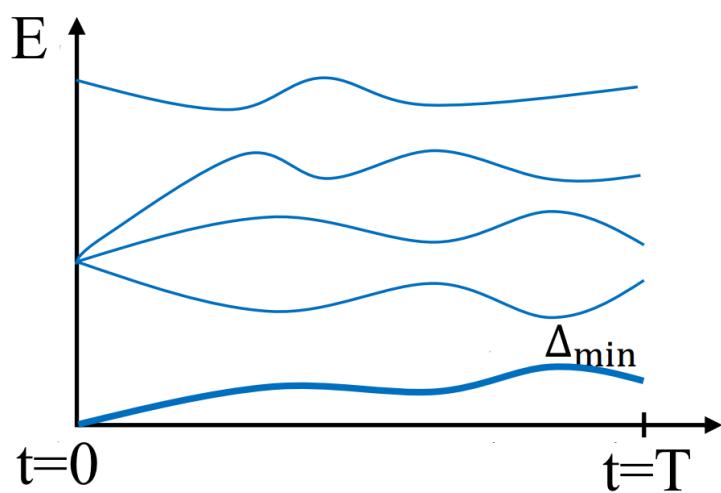


Fig. 71.2: Типичная зависимость от времени энергетических уровней гамильтонианов, используемых в квантовом отжиге

сейчас ничего не понял, рекомендуется также вернуться к лекции *о переходе от комбинаторных задач к гамильтонианам*, чтобы понять, как получается *cost* гамильтониан.

Note: Уравнение Шредингера практически не решается численно для сколько-нибудь больших задач, поэтому мы примем как факт то, что точно посчитать квантовую динамику аннилера у нас не выйдет, тем более не выйдет как-то пытаться оптимизировать

Для решения проблемы “нерешаемости” уравнения Шредингера перейдем от непрерывного времени t и зависимостей $A(t)$ и $B(t)$ к N дискретных моментов времени t_1, t_2, \dots, t_n . Тогда можно заменить расписание отжига в виде непрерывных коэффициентов на набор дискретных коэффициентов, каждый из которых отвечает своему моменту времени:

$$\begin{aligned} A(t) &\rightarrow \gamma_1, \gamma_2, \dots, \gamma_N \\ B(t) &\rightarrow \beta_1, \beta_2, \dots, \beta_N \end{aligned}$$

Финальное состояние записывается так:

$$\Psi(t) = e^{-i\gamma_1 \mathcal{H}_{initial}} e^{-i\beta_1 \mathcal{H}_{cost}} \dots e^{-i\gamma_N \mathcal{H}_{initial}} e^{-i\beta_N \mathcal{H}_{cost}} \Psi(0)$$

71.1.4 Оптимизация расписания

Теперь задача оптимизации расписания по своей сути сведена к следующей:

$$\begin{aligned} \arg \min_{\gamma_1, \dots, \gamma_N, \beta_1, \dots, \beta_N} & \Psi_{final} | \mathcal{H}_{cost} | \Psi_{final} \\ \Psi_{final} &= e^{-i\gamma_1 \mathcal{H}_{initial}} e^{-i\beta_1 \mathcal{H}_{cost}} \dots e^{-i\gamma_N \mathcal{H}_{initial}} e^{-i\beta_N \mathcal{H}_{cost}} \Psi(0) \end{aligned}$$

А это уже хорошо знакомая задача оптимизации результата измерения состояния, заданного при помощи некоторой *VQC*, которая параметризована набором действительных чисел $\gamma_1, \dots, \gamma_N, \beta_1, \dots, \beta_N$. И эта задача решается хорошо уже знакомыми *градиентными методами*.

71.2 Пример задачи оптимизации

Рассмотрим задачу оптимизации n -разрядного набора данных, для которого нужно найти некоторый минимум (или максимум). Алгоритм задается двумя гамильтонианами H_p и H_M , а также $2p$ параметрами: $\gamma_1, \dots, \gamma_p$ и β_1, \dots, β_p .

QAOA использует унитарный оператор $U(\beta, \gamma)$, принимающий на вход вещественные параметры β , γ , и описывается уже знакомым квантовым состоянием Ψ . Цель поиска – найти те самые оптимальные β_{opt} и γ_{opt} .

Оператор U состоит из двух частей:

- оператор, меняющий фазу U_{phase}

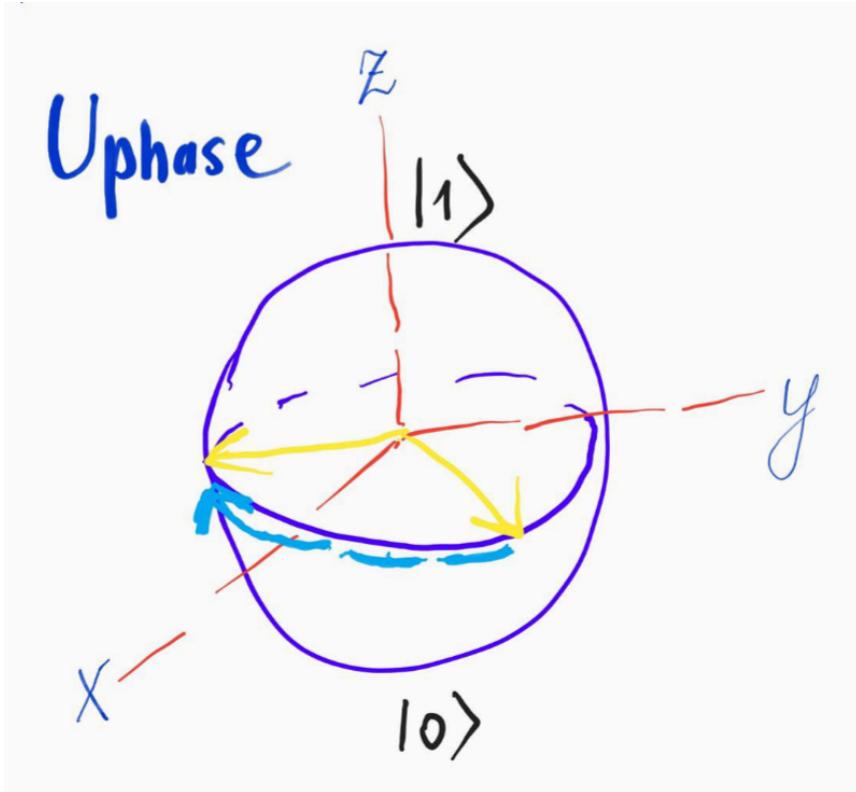
$$U_{\text{phase}}(\gamma) = e^{-i\gamma H_{\text{phase}}}$$

- оператор, смешивающий кубиты U_{mixer}

$$U_{\text{mixer}}(\beta) = e^{-i\beta H_{\text{mixer}}}$$

Оператор U_{phase} совершает вращение относительно осей Z или Y с помощью соответствующих матриц Паули

$$H_{\text{phase}} = Z \text{ or } Y \text{ axis rotation}$$

Fig. 71.3: Оператор U_{phase}

U_{mixed} в классическом случае использует матрицу $X \text{NOT}$.

Операторы применяются к начальному состоянию Ψ_0 (путем поочередного применения гамильтонианов H_p и H_M) последовательно раз (или, иначе говоря, используются p слоев), где продолжительность j -й итерации определяется параметрами γ_j и β_j соответственно

$$\phi(\beta, \gamma) = \underbrace{U_{\text{mixer}}(\beta)U_{\text{phase}}(\gamma) \dots U_{\text{mixer}}(\beta)U_{\text{phase}}(\gamma)}_{p \text{ times}} \Psi_0$$

Общая схема для n кубитов выглядит следующим образом

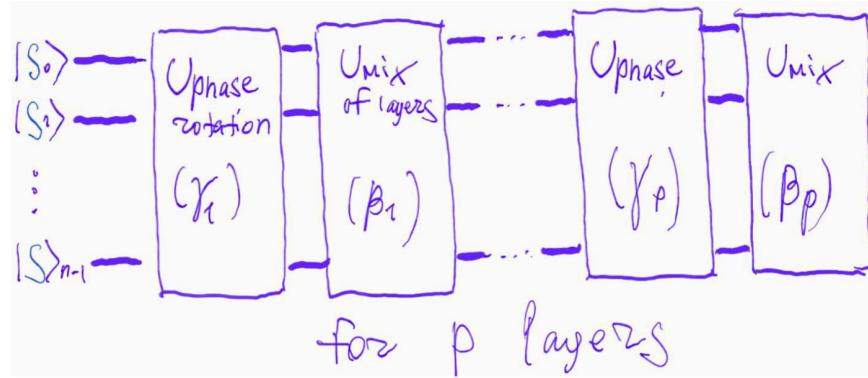
Итак, алгоритм состоит из следующих основных этапов:

1. приготовление начального состояния Ψ_0 из n кубитов. Начальное состояние выбирается как равное состояние суперпозиции всех возможных решений

$$\Psi_0 = \frac{1}{\sqrt{2^n}} \sum_x x$$

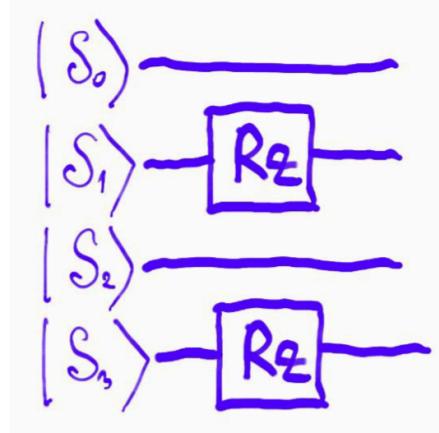
2. последующее применение к каждому кубиту матриц Адамара для осуществления суперпозиции всевозможных состояний
3. применяем оператор вращения фазы

$$U_{\text{phase}} = \sum_{i \neq j}^{n-1} e^{-i\gamma Z_i Z_j}$$

Fig. 71.4: Общая схема для n кубитов

например, вот так

$$H_p = (I_0 \otimes Z_1 \otimes I_2 \otimes Z_3)$$



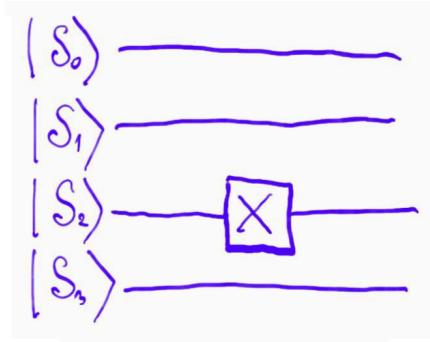
Напоминаем, как выглядит данный оператор в матричном виде: $Z = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.

4. применяем смешивающий оператор

$$U_{\text{mixer}} = \sum_{i=0}^{n-1} e^{-i\beta X_i}$$

к примеру, так

$$U_{\text{mixer}} = (I \otimes I \otimes X \otimes Z)$$



$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

В данном алгоритме используется адиабатический метод эволюции состояния [FGGS00] Ψ_0 с переменным гамильтонианом: на каждой итерации параметры β и γ понемногу изменяются.

Далее производится измерение финального состояния в Z -базисе и вычисление $\Psi(\beta, \gamma)H_{phase}\Psi(\beta, \gamma)$. Найденный минимум будет соответствовать оптимальным β и γ .

Описанные выше шаги могут быть полностью повторены с обновлёнными наборами временных параметров в рамках классического цикла оптимизации (такого как градиентный спуск или другие подходы), используемого для оптимизации параметров алгоритма.

Возвращается лучшее решение, найденное за всё время поиска.

71.3 Quantum Alternating Operator Ansatz

Применение “анзаца” в алгоритме квантовой приближенной оптимизации заключается в модернизации оператора смешивания U_{mixer} и предполагает использование $CNOT$, а не X .

Анзац рассматривает более общие параметризованные унитарные трансформации, а не только соответствующие эволюции фиксированного локального гамильтониана во времени. Он позволяет более эффективно реализовывать операции смешивания, особенно в задачах оптимизации с жесткими ограничениями.

На рисунках ниже представлена абстрактная визуализация “смешивания” и обозначение оператора:

Классически семейство анзацев можно поделить на три основных типа:

- Hardware Efficient Ansatz (HEA) – запутывающий все кубиты;
- Alternating Layered Ansatz (ALT) [FGG14];
- Tensor Product Ansatz (TEN) [HPM+19].

На рисунке ниже изображены упрощённые (без учета анзаца и фазовых гейтов) для понимания схемы описанных компоновок – по одному слою каждого типа:

Конечно, к компоновке смешанных гейтов можно подходить сколь угодно творчески, и пример общей схемы, реализующей QAOAz, представлен ниже:

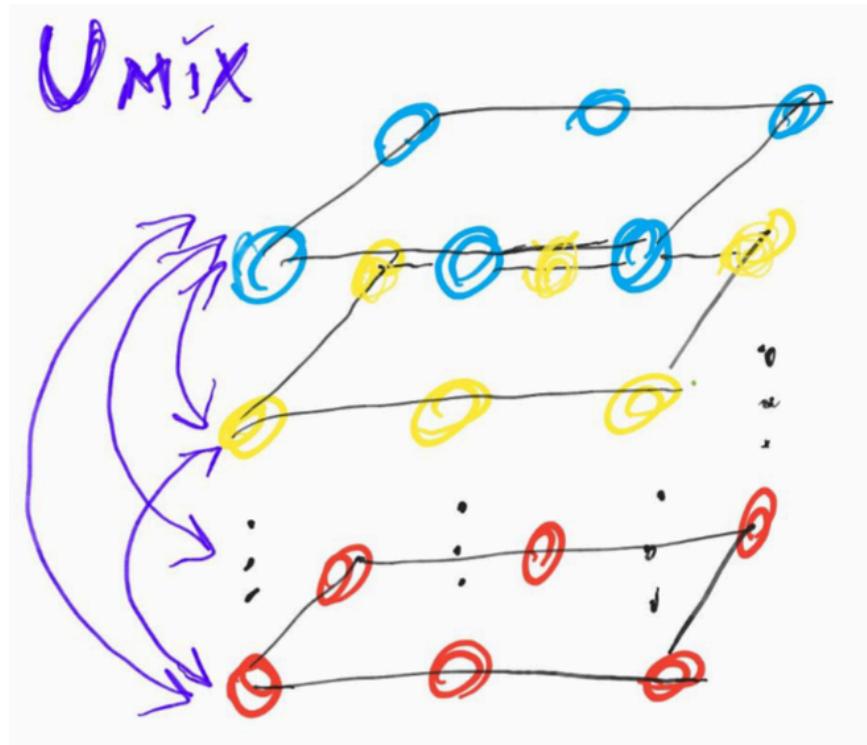
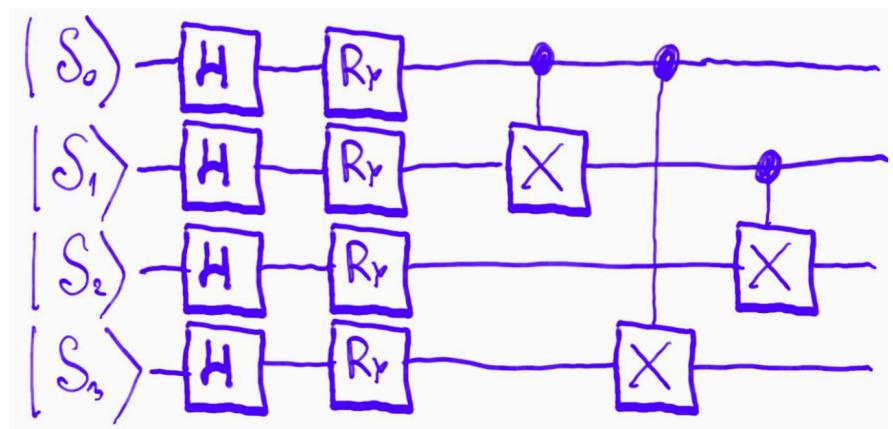
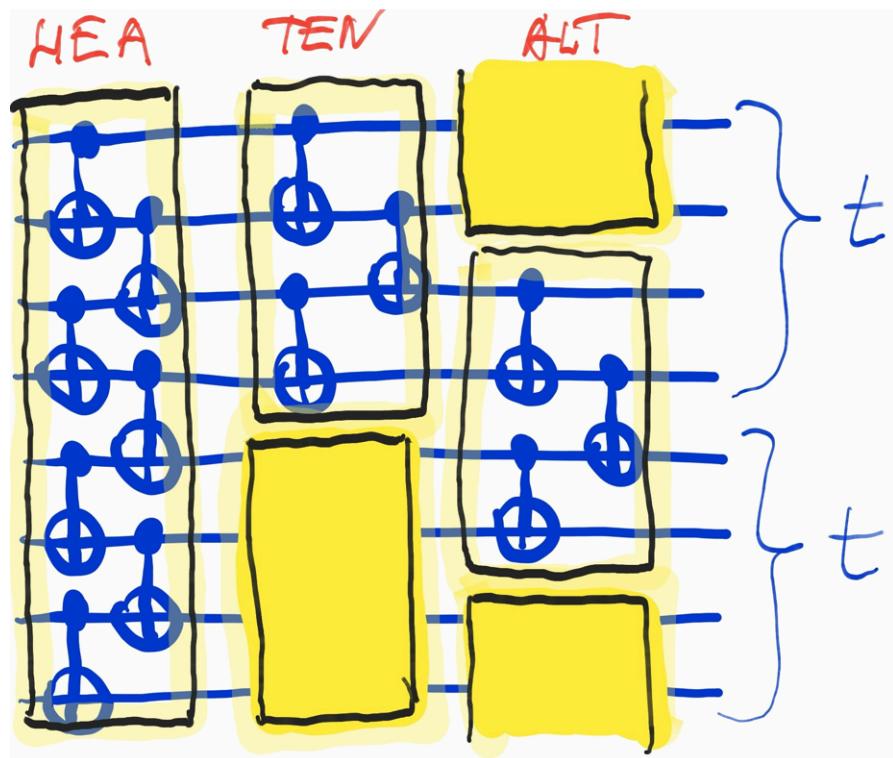


Fig. 71.5: “Смешивание”

$$U_{\text{Mix}}(\beta_i) := \sum_{j,j+k}^{n-1} U_{\text{Mix } j, j+k}(\beta_i)$$

The diagram shows a quantum circuit element labeled $U_{\text{Mix } j, j+k}(\beta_i)$. It consists of two input ports labeled j and $j+k$, and two output ports labeled j and $j+k$. Ellipses indicate that there are other similar elements in the sequence.

Fig. 71.6: Обозначение оператора



CHAPTER
SEVENTYTWO

АКАДЕМИЯ И ИНДУСТРИЯ

CHAPTER
SEVENTYTHREE

СПИСОК ЛИТЕРАТУРЫ

СПИСОК АВТОРОВ КУРСА

74.1 Создатели курса

- Семен Синченко
- Юрий Кашницкий
- Виктор Трохименко

74.2 Основные авторы

- Александр Березутский
- Александра Овсянникова
- Алексей Озерин
- Алексей Пронькин
- Андрей Лукьяненко
- Борис Зимка
- Вадим Петров
- Дани Эль-Айясс
- Дмитрий Бажанов
- Дмитрий Бурдений
- Евгений Желтоножский
- Игорь Котенков
- Игорь Токарев
- Илья Беседин
- Карина Решетова
- Леонид Сендерович
- Николай Карелин
- Никита Щуцкий
- Семен Синченко
- Сергей Ширкин

- Юрий Кашницкий

74.3 Основные ревьюеры

- Борис Зимка
- Виктор Трохименко
- Евгений Желтоножский
- Николай Карелин
- Роман Тезиков
- Юрий Кашницкий
- Ivan Dikhtyar

74.4 Техническая поддержка

- Алексей Пронькин
- Дмитрий Коржов

74.5 Редакторы

- Екатерина Пацкова
- Наталья Маркова

БЛАГОДАРНОСТИ

Команда курса выражает благодарность:

- сообществу [Open Data Science](#) за предоставление платформы для размещения курса и техническую поддержку;

ГЛОССАРИЙ

76.1 А

Алгоритм Гровера Англ. Grover's algorithm

Алгоритм квантового приближения для задачи оптимизации (QAOA, quantum approximate optimization algorithm), <https://www.osp.ru/os/2019/03/13055118>, <https://habr.com/ru/post/513474/>

Аннилер Англ. annealer. Устройство для приближенного решения задачи оптимизации https://en.wikipedia.org/wiki/Quantum_annealing

76.2 Б

Бра-вектор Англ. bra-vector

76.3 В

Вакуумное состояние Англ. Vacuum state, также используется название “основное состояние” (basic state).

Так часто называется квантовое состояние с вектором 0. Термин заимствован из квантовой оптики, чаще всего не имеет ничего общего с “обыденным” понятием “вакуума”.

Вариационная квантовая схема Англ. Variational Quantum Circuits (VQC)

Вариационное машинное обучение Англ. variational machine learning

76.4 Г

Гейты Англ. Quantum gates

Гейт Адамара Англ. Hadamard gate

76.5 И

Измерение Англ. Quantum measurement

76.6 К

Квантовая информация Também используется термин “квантовая информатика”, англ. Quantum Information

Квантовый оператор Англ. Quantum operator

Квантово-классическое обучение Англ. Quantum-classical machine learning

Квантовое состояние Англ. Quantum state

Кет-вектор Англ. ket-vector

Кот Шредингера Англ. Schrödinger's cat

Кубит Também кубит или q-bit, англ. Qubit, от quantum bit. Минимальная единица информации на квантовом компьютере. Как и (классический) бит, кубит допускает два состояния, которые обычно обозначаются кет-векторами 0 и 1, а на уровне реализации кубит – это двухуровневая квантовая система, такая как две поляризации фотона или спина электрона.

Кудит Англ. qudit. Квантовая система с d состояниями.

76.7 М

Матрица плотности Também используется термин “оператор плотности”, англ. Density matrix или Density operator

Матричное прямое произведение Синоним произведение Кронекера см. Кронекер. Англ. Tensor product

Метод квантового локального подбора (QLS, quantum local search), <https://www.osp.ru/os/2019/03/13055118>

Многокубитные гейты Англ. Multiqubit gates

76.8 Н

Нотация Дирака Англ. Dirac notation, формализм (система обозначения) для описания квантовых состояний на основе бра- и кет-векторов (англ. bra-ket от bracket, скобка). В этой системе обозначений Ψ обозначает вектор состояния, Ψ^* – сопряженный вектор состояния, а $\Psi \hat{H} \Psi$ – среднее (математическое ожидаемое) значения наблюдаемой для оператора \hat{H} в состоянии Ψ (также говорят о свертке оператора \hat{H} с бра-вектором Ψ и кет-вектором Ψ). Система обозначений Дирака позволяет отделить вектора состояния Ψ от самого состояния Ψ или конкретного математического описания (в виде волновой функции или вектора).

76.9 О

Операторы Паули Также используется термин “матрицы Паули”, англ. Pauli operators или Pauli matrices

76.10 П

Правило сдвига параметров (*) Англ. Parameter-shift rule

Произведение Кронекера То же, что и матричное прямое произведение (тензорное произведение в случае линейных операторов). Англ. Kronecker product

76.11 С

Смешанные состояния Англ. Mixed states

Собственный вектор Англ. Eigenvector

Собственное значение Англ. Eigenvalue

Соотношение неопределенности Também используется термин “принцип неопределенности”, англ. Uncertainty relation или Uncertainty principle

Суперпозиция Англ. Superposition

Сфера Блоха Англ. Bloch sphere

76.12 У

Унитарный оператор Англ. Unitary operator

76.13 Ш

«Шумные» квантовые компьютеры промежуточного масштаба (Noisy intermediate-scale quantum, NISQ, NISQ Hardw

Термин введен Прескиллом в работе <https://arxiv.org/abs/1801.00862> для описания сегодняшнего состояния квантовых вычислений, когда “настоящие” квантовые вычисления (на миллионах кубит) еще невозможны, но современные квантовые компьютеры уже могут выполнять вычисления, которые невозможны на обычных, классических компьютерах.

См. также краткое обсуждение в <https://quantumcomputing.stackexchange.com/questions/1885/what-is-meant-by-noisy-intermediate-scale-quantum-nisq-technology>, и перевод на термина русский <https://www.osp.ru/os/2019/03/13055130>

76.14 Э

Энтропия Англ. Entropy

BIBLIOGRAPHY

- [ion] Single trapped atom captures science photography competition's top prize. Accessed: 2021-10-05. URL: <https://webarchive.nationalarchives.gov.uk/ukgwa/20200930161403/https://epsrc.ukri.org/newsevents/news/single-trapped-atom-captures-science-photography-competitions-top-prize/>.
- [Mon] Varenna summer school on ion traps: quantum simulation, lecture i. Accessed: 2021-10-05. URL: https://iontrap.umd.edu/wp-content/uploads/2014/01/Lecture1_QSIM.pptx.
- [AMunozNK21] Shahnawaz Ahmed, Carlos Sánchez Muñoz, Franco Nori, and Anton Frisk Kockum. Quantum state tomography with conditional generative adversarial networks. *Physical Review Letters*, 127(14):140502, 2021. doi:<https://doi.org/10.1103/PhysRevLett.127.140502>.
- [And19] Neculai Andrei. A diagonal quasi-newton updating method for unconstrained optimization. *Numerical Algorithms*, 81(2):575–590, 2019. doi:<https://doi.org/10.1007/s11075-018-0562-7>.
- [Arn51] Walter Edwin Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of applied mathematics*, 9(1):17–29, 1951. URL: <https://www.ams.org/journals/qam/1951-09-01/S0033-569X-1951-42792-9/S0033-569X-1951-42792-9.pdf>.
- [ABBradler+21] J. M. Arrazola, V. Bergholm, K. Brádler, T. R. Bromley, M. J. Collins, I. Dhand, A. Fumagalli, T. Gerrits, A. Goussev, L. G. Helt, J. Hundal, T. Isacsson, R. B. Israel, J. Izaac, S. Jahangiri, R. Janik, N. Killoran, S. P. Kumar, J. Lavoie, A. E. Lita, D. H. Mahler, M. Menotti, B. Morrison, S. W. Nam, L. Neuhaus, H. Y. Qi, N. Quesada, A. Repington, K. K. Sabapathy, M. Schuld, D. Su, J. Swinerton, A. Száva, K. Tan, P. Tan, V. D. Vaidya, Z. Vernon, Z. Zabaneh, and Y. Zhang. Quantum circuits with many photons on a programmable nanophotonic chip. *Nature*, 591(7848):54–60, mar 2021. arXiv:2103.02109, doi:10.1038/s41586-021-03202-1.
- [AAB+19] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, and others. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019. doi:<https://doi.org/10.1038/s41586-019-1666-5>.
- [BGrotschelJungerR88] Francisco Barahona, Martin Grötschel, Michael Jünger, and Gerhard Reinelt. An application of combinatorial optimization to statistical physics and circuit layout design. *Operations Research*, 36(3):493–513, 1988.
- [BAG21] Afrad Basheer, A. Afham, and Sandeep K. Goyal. Quantum $\$k\$$ -nearest neighbors algorithm. 2021. arXiv:2003.09187.
- [BV04] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, March 2004. ISBN 0521833787. URL: <http://www.amazon.com/exec/obidos/redirect?tag=citeulike-20&TU\textbackslashtextbackslash{}&path=ASIN/0521833787>.
- [BK02] Sergey Bravyi and Alexei Kitaev. Fermionic quantum computation. *Annals of Physics*, 298(1):210–226, 2002.

- [BVM+21] Michael Broughton, Guillaume Verdon, Trevor McCourt, Antonio J. Martinez, Jae Hyeon Yoo, Sergei V. Isakov, Philip Massey, Ramin Halavati, Murphy Yuezhen Niu, Alexander Zlokapa, Evan Peters, Owen Lockwood, Andrea Skolik, Sofiene Jerbi, Vedran Dunjko, Martin Leib, Michael Streif, David Von Dollen, Hongxiang Chen, Shuxiang Cao, Roeland Wiersema, Hsin-Yuan Huang, Jarrod R. McClean, Ryan Babbush, Sergio Boixo, Dave Bacon, Alan K. Ho, Hartmut Neven, and Masoud Mohseni. Tensorflow quantum: a software framework for quantum machine learning. 2021. URL: <https://arxiv.org/abs/2003.02989>, arXiv:2003.02989.
- [CCH+19] Giuseppe Carleo, Kenny Choo, Damian Hofmann, James ET Smith, Tom Westerhout, Fabien Alet, Emily J Davis, Stavros Efthymiou, Ivan Glasser, Sheng-Hsuan Lin, and others. Netket: a machine learning toolkit for many-body quantum systems. *SoftwareX*, 10:100311, 2019.
- [CT17] Giuseppe Carleo and Matthias Troyer. Solving the quantum many-body problem with artificial neural networks. *Science*, 355(6325):602–606, 2017. URL: <https://arxiv.org/abs/1606.02318>.
- [CBSG17] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language. 2017. URL: <https://arxiv.org/abs/1707.03429>, arXiv:1707.03429.
- [DFO20] Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for machine learning*. Cambridge University Press, 2020. <https://mml-book.github.io/book/mml-book.pdf>.
- [DHM+18] Danial Dervovic, Mark Herbster, Peter Mountney, Simone Severini, Naïri Usher, and Leonard Wossnig. Quantum linear systems algorithms: a primer. 2018. arXiv:1802.08227.
- [FGG14] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm. 2014. arXiv:1411.4028.
- [FGGS00] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. Quantum computation by adiabatic evolution. 2000. arXiv:0001106.
- [GKD19] Fred Glover, Gary Kochenberger, and Yu Du. A tutorial on formulating and using qubo models. 2019. arXiv:1811.11538.
- [GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. <http://www.deeplearningbook.org>.
- [GK21] Peter Groszkowski and Jens Koch. Scqubits: a python package for superconducting qubits. *arXiv e-prints*, jul 2021. arXiv:2107.08552.
- [HHL09] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical Review Letters*, Oct 2009. URL: <https://arxiv.org/abs/0811.3171v3>, doi:10.1103/physrevlett.103.150502.
- [Has70] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 04 1970. URL: <https://doi.org/10.1093/biomet/57.1.97>, arXiv:<https://academic.oup.com/biomet/article-pdf/57/1/97/23940249/57-1-97.pdf>, doi:10.1093/biomet/57.1.97.
- [HavlícekCorcolesT+19] Vojtěch Havlíček, Antonio D Córdoles, Kristan Temme, Aram W Harrow, Abhinav Kandala, Jerry M Chow, and Jay M Gambetta. Supervised learning with quantum-enhanced feature spaces. *Nature*, 567(7747):209–212, Mar 2019. URL: <https://arxiv.org/abs/1804.11326>, doi:10.1038/s41586-019-0980-2.
- [HSchatzleNoe20] Jan Hermann, Zeno Schätzle, and Frank Noé. Deep-neural-network solution of the electronic schrödinger equation. *Nature Chemistry*, 12(10):891–897, 2020. URL: <https://www.nature.com/articles/s41557-020-0544-y>.
- [HAGH+20] Mohamed Hibat-Allah, Martin Ganahl, Lauren E Hayward, Roger G Melko, and Juan Carrasquilla. Recurrent neural network wave functions. *Physical Review Research*, 2(2):023358, 2020. URL: <https://arxiv.org/abs/2002.02973>.
- [HZL+17] He-Liang Huang, You-Wei Zhao, Tan Li, Feng-Guang Li, Yu-Tao Du, Xiang-Qun Fu, Shuo Zhang, Xiang Wang, and Wan-Su Bao. Homomorphic encryption experiments on ibm's cloud quantum computing platform. 2017. arXiv:1612.02886.

- [HPM+19] William Huggins, Piyush Patil, Bradley Mitchell, K Birgitta Whaley, and E Miles Stoudenmire. Towards quantum machine learning with tensor networks. *Quantum Science and Technology*, 4(2):024001, Jan 2019. URL: <https://arxiv.org/abs/1803.11537>, doi:10.1088/2058-9565/aaea94.
- [Isi25] Ernst Ising. Beitrag zur theorie des ferromagnetismus. *Zeitschrift für Physik*, 31(1):253–258, 1925.
- [Jas55] Robert Jastrow. Many-body problem with strong forces. *Physical Review*, 98(5):1479, 1955.
- [KMT+17] Abhinav Kandala, Antonio Mezzacapo, Kristan Temme, Maika Takita, Markus Brink, Jerry M Chow, and Jay M Gambetta. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature*, 549(7671):242–246, 2017. URL: <https://www.nature.com/articles/nature23879>, doi:10.1038/nature23879.
- [KYG+07] Jens Koch, Terri M Yu, Jay Gambetta, A A Houck, D I Schuster, J Majer, Alexandre Blais, M H Devoret, S M Girvin, and R J Schoelkopf. Charge-insensitive qubit design derived from the Cooper pair box. *Physical Review A*, 10 2007. doi:10.1103/physreva.76.042319.
- [KKY+19] P Krantz, M Kjaergaard, F Yan, T P Orlando, S Gustavsson, and W D Oliver. A Guide to Superconducting Qubits for New Quantum Information Engineers. *Applied Physics Review*, 2019. arXiv:1904.06560v1.
- [Lan50] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 1950.
- [LD10] Ailsa H Land and Alison G Doig. An automatic method for solving discrete programming problems. In *50 Years of Integer Programming 1958-2008*, pages 105–132. Springer, 2010.
- [LTG21] Junde Li, Rasit Topaloglu, and Swaroop Ghosh. Quantum generative models for small molecule drug discovery. 2021. arXiv:2101.03438.
- [LJGS+90] EY Loh Jr, JE Gubernatis, RT Scalettar, SR White, DJ Scalapino, and RL Sugar. Sign problem in the numerical simulation of many-electron systems. *Physical Review B*, 41(13):9301, 1990.
- [Lom04] Chris Lomont. The hidden subgroup problem-review and open problems. *arXiv preprint quant-ph/0411037*, 2004.
- [Luc14] Andrew Lucas. Ising formulations of many np problems. *Frontiers in physics*, 2:5, 2014. URL: <https://arxiv.org/abs/1302.5843>, doi:<https://doi.org/10.3389/fphy.2014.00005>.
- [MBK21] Andrea Mari, Thomas R. Bromley, and Nathan Killoran. Estimating the gradient and higher-order derivatives on quantum hardware. *Physical Review A*, 103(1):012405, Jan 2021. URL: <https://arxiv.org/abs/2008.06517>, doi:10.1103/physreva.103.012405.
- [MST17] Kyle Mills, Michael Spanner, and Isaac Tamblyn. Deep learning and the schrödinger equation. *Physical Review A*, 96(4):042113, 2017. URL: <https://arxiv.org/abs/1702.01361>.
- [MKF19] Kosuke Mitarai, Masahiro Kitagawa, and Keisuke Fujii. Quantum analog-digital conversion. *Phys. Rev. A*, 99:012301, Jan 2019. URL: <https://link.aps.org/doi/10.1103/PhysRevA.99.012301>, doi:10.1103/PhysRevA.99.012301.
- [MNKF18] Kosuke Mitarai, Makoto Negoro, Masahiro Kitagawa, and Keisuke Fujii. Quantum circuit learning. *Physical Review A*, 98(3):032309, Sep 2018. URL: <https://arxiv.org/abs/1803.00745>, doi:10.1103/PhysRevA.98.032309.
- [MRT18] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2 edition, 2018. ISBN 978-0-262-03940-6.
- [NSS+08] Chetan Nayak, Steven H. Simon, Ady Stern, Michael Freedman, and Sankar Das Sarma. Non-abelian anyons and topological quantum computation. *Reviews of Modern Physics*, 80(3):1083–1159, Sep 2008. URL: <http://dx.doi.org/10.1103/RevModPhys.80.1083>, doi:10.1103/revmodphys.80.1083.

- [New06] M. E. J. Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, May 2006. URL: <https://arxiv.org/abs/physics/0602124>, doi:10.1073/pnas.0601602103.
- [RGM+17] Marco Roth, Marc Ganzhorn, Nikolaj Moll, Stefan Philipp, Gian Salis, and Sebastian Schmidt. Analysis of a parametrically driven exchange-type gate and a two-photon excitation gate between superconducting qubits. *Physical Review A*, 2017. doi:10.1103/PhysRevA.96.062323.
- [SZK+18] N. Samkharadze, G. Zheng, N. Kalhor, D. Brousse, A. Sammak, U. C. Mendes, A. Blais, G. Scappucci, and L. M. K. Vandersypen. Strong spin-photon coupling in silicon. *Science*, 359(6380):1123–1127, 2018. URL: <https://www.science.org/doi/abs/10.1126/science.aar4054>, doi:10.1126/science.aar4054.
- [SRL12] Jacob T. Seeley, Martin J. Richard, and Peter J. Love. The bravyi-kitaev transformation for quantum computation of electronic structure. *The Journal of chemical physics*, 137(22):224109, 2012. URL: <https://arxiv.org/abs/1208.5986>.
- [SB19] Semyon Sinchenko and Dmitry Bazhanov. The deep learning and statistical physics applications to the problems of combinatorial optimization. *arXiv preprint arXiv:1911.10680*, 2019. URL: <https://arxiv.org/abs/1911.10680>.
- [SCZ17] Robert S. Smith, Michael J. Curtis, and William J. Zeng. A practical quantum instruction set architecture. 2017. arXiv:1608.03355.
- [SCBR14] Stanislav Sobolevsky, Riccardo Campari, Alexander Belyi, and Carlo Ratti. General optimization technique for high-quality community detection in complex networks. *Phys. Rev. E*, 90:012811, Jul 2014. URL: <https://link.aps.org/doi/10.1103/PhysRevE.90.012811>, doi:10.1103/PhysRevE.90.012811.
- [SBC+21] Samuel A. Stein, Betis Baheri, Daniel Chen, Ying Mao, Qiang Guan, Ang Li, Bo Fang, and Shuai Xu. Qugan: a generative adversarial network through quantum states. 2021. arXiv:2010.09036.
- [SF14] Leonard Susskind and Art Friedman. *Quantum mechanics: the theoretical minimum*. Basic Books, 2014.
- [SYG+20] Yudai Suzuki, Hiroshi Yano, Qi Gao, Shumpei Uno, Tomoki Tanaka, Manato Akiyama, and Naoki Yamamoto. Analysis and synthesis of feature map for kernel-based quantum classifier. *Quantum Machine Intelligence*, 2(1):1–9, Jul 2020. URL: <https://arxiv.org/abs/1906.10467>, doi:10.1007/s42484-020-00020-y.
- [TKTD20] Alexander Teplukhin, Brian K Kendrick, Sergei Tretiak, and Pavel A Dub. Electronic structure with direct diagonalization on a d-wave quantum annealer. *Scientific reports*, 10(1):1–11, 2020.
- [VDS98] Jan Von Delft and Herbert Schoeller. Bosonization for beginners—refermionization for experts. *Annalen der Physik*, 7(4):225–305, 1998.
- [VD17] Uri Vool and Michel Devoret. Introduction to quantum electromagnetic circuits. *International Journal of Circuit Theory and Applications*, 45(7):897–934, 2017. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cta.2359>, doi:10.1002/cta.2359.
- [WSB+05] A. Wallraff, D. I. Schuster, A. Blais, L. Frunzio, J. Majer, M. H. Devoret, S. M. Girvin, and R. J. Schoelkopf. Approaching unit visibility for control of a superconducting qubit with dispersive readout. *Physical Review Letters*, 95(6):060501, 8 2005. URL: <https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.95.060501>, arXiv:0502645, doi:10.1103/PhysRevLett.95.060501.
- [WBD+19] K. Wright, K. M. Beck, S. Debnath, J. M. Amini, Y. Nam, N. Grzesiak, J. S. Chen, N. C. Pisenti, M. Chmielewski, C. Collins, K. M. Hudek, J. Mizrahi, J. D. Wong-Campos, S. Allen, J. Apisdorf, P. Solomon, M. Williams, A. M. Ducore, A. Blinov, S. M. Kreikemeier, V. Chaplin, M. Keesan, C. Monroe, and J. Kim. Benchmarking an 11-qubit quantum computer. *Nature Communications*, 10(1):1–6, dec 2019. arXiv:1903.08181, doi:10.1038/s41467-019-13534-2.
- [Zac77] Wayne W Zachary. An information flow model for conflict and fission in small groups. *Journal of anthropological research*, 33(4):452–473, 1977.
- [01] Кокин А. А. Валиев К. А. *Квантовые компьютеры: надежды и реальность*. НИЦ "Регулярная и хаотическая динамика", 2001.

- [89] Лифшиц Е. М. Ландау Л. Д. *Квантовая механика: Нерелятивистская теория*. Наука, 1989. URL: <https://www.math.purdue.edu/~eremenko/dvi/LL.pdf>.
- [15a] Иванов М.Г. *Как понимать квантовую механику*. Регулярная и хаотическая динамика, 2015. URL: <https://mipt.ru/upload/medialibrary/533/quant-2.pdf>.
- [07] Килин С. Я. Могилевцев Д. С. *Методы квантовой оптики структурированных резервуаров*. Белорусская наука, 2007. URL: <https://www.litres.ru/d-s-mogilevcev/metody-kvantovoy-optiki-strukturirovannyh-rezervuarov-7073607/>.
- [15b] Леонард Сасскинд and Арт Фридман. *Квантовая механика. Теоретический минимум*. СПб.: Питер, 2015.