

Приветствие

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

Почему именно этот курс?

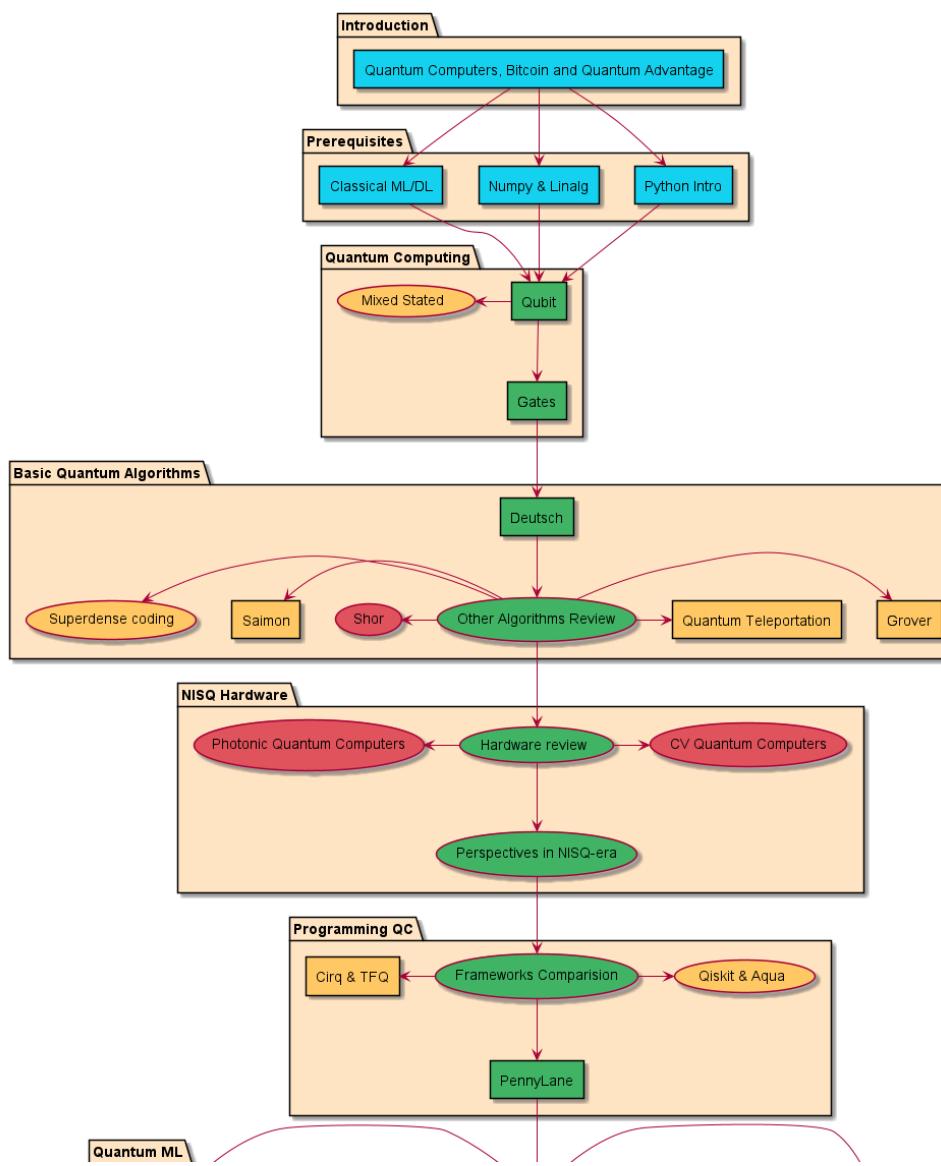
Наш курс отличается от других курсов по квантовым вычислениям:

- 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.

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.



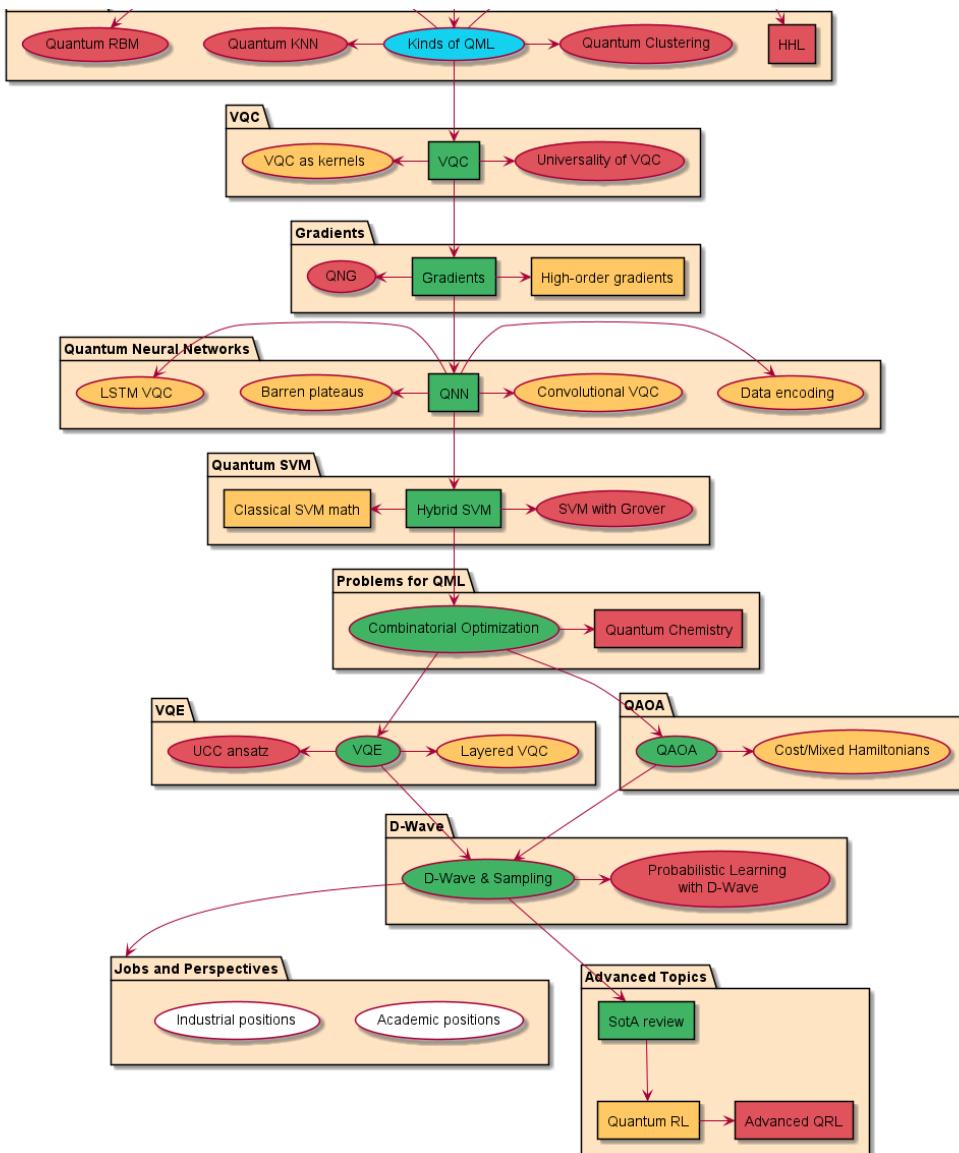


Fig. 1 Программа курса

Как будет проходить этот курс?

Рекомендуем проходить курс в порядке, обозначенном на схеме.

Желаем успехов!

О квантовых компьютерах, биткоине и превосходстве

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?

Что это вообще за компьютеры такие?

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!



Fig. 2 Иллюстрация закона Мура — рост числа транзисторов с 1970-х

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.

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.

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. 3 Ричард Фейнман, 1918-1988

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

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 квантов. Также неправильно было бы говорить о том, что в отличии от классических компьютеров, где есть лишь $|0\rangle$ и $|1\rangle$ 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 это не только использование новых материалов или технической базы, а еще и использование новой, отличной от заложенной Тьюрингом концепции вычислений, представления и обработки информации.

А зачем это вообще нужно?

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. 4 Питер Шор, тот, кто переполошил весь мир своим алгоритмом

Данному алгоритму будет посвящена отдельная лекция нашего курса и 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.

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 $\text{NP} \neq \text{P}$, 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*.



Fig. 5 Визуализация решения задачи коммивояжера – кратчайший путь, чтобы обехать 12 немецких городов – очень важная задача современной логистики

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 тут приходит вторая проблема, а именно, что явно интегрировать уравнение Шрёдингера по времени, или, по простому, решать квантовую механику **численно** тоже вычислительно почти невозможно более чем для двух частиц.



Fig. 6 Эрвин Шрёдингер, 1887-1961, создатель знаменитого уравнения и мема про кота

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 многое другое. 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.

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!

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

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: $\sim 100\text{-}500$ кубит

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

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.

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.



Fig. 8 Квантовый компьютер компании IBM выглядит примерно так

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. Тема 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 $\backslash(\text{lsim } 55)$ кубит
- рекорд в относительно точных, но медленных и плохо масштабируемых компьютерах это $\backslash(\text{lsim } 20)$ кубит
- рекорд в точных и масштабируемых, но очень трудно программируемых компьютерах это $\backslash(\text{lsim } 25)$ 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

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.

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!

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).

Понятие квантового превосходства

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



Fig. 9 Джон Прескил, который и придумал этот термин. Еще он известен своим знаменитым пари с другим физиком Стивеном Хокингом (которое Хокинг проиграл)

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.

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 \sim 20) дней;
- 2021 год, группа китайских ученых оптимизирует классический алгоритм и решает задачу на 60 видеокартах *NVIDIA* за 7 дней;
- 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.

А как это вообще выглядит? И сколько стоит?

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

- сверхнизкие температуры
- сверхвысокий вакум
- 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 USD.

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.

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!

О блоке

Этот блок включает в себя:

- вступительный тест по Python, позволяющий определить свой уровень;
- 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 пройти [тест по ссылке](#) – 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.

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.

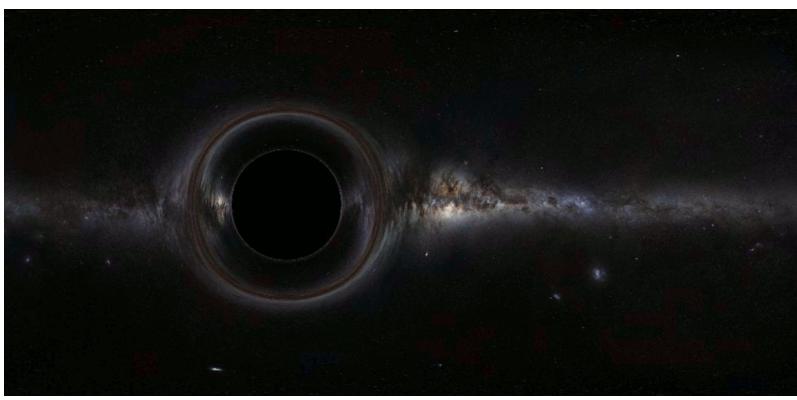


Fig. 10 [Рассчитанная в Python симуляция преломлений света черной дырой](#)

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 *расширенный 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.

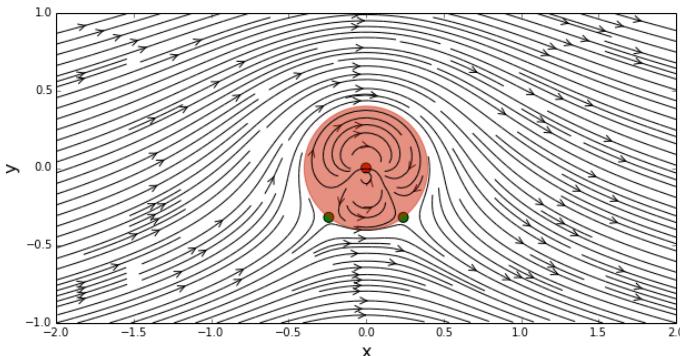


Fig. 11 Пример моделирования аэродинамики в Python с помощью библиотеки AeroPython

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.

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.

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.

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 ([Дзэн Питона](#)) на русском языке – [по ссылке](#).
- [Python](#) – это *открытый проект*, в который каждый может внести изменения (но они должны быть предварительно одобрены), например, [IUT](#).
- 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. 12 Кот для привлечения внимания и в благодарность за то, что вы начали проходить курс и сделали самый сложный шаг – прошли первую лекцию!

Знакомство с инструментарием: Jupyter

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.

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:

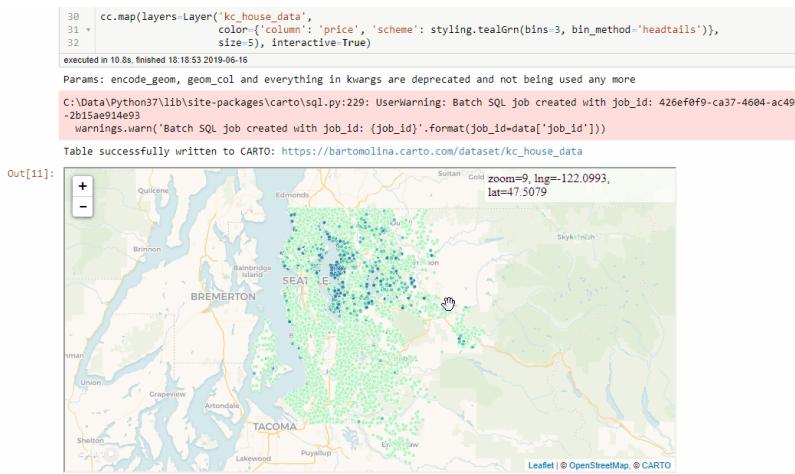


Fig. 13 Пример интерактивной визуализации прямо внутри рабочего файла с кодом

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.

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:

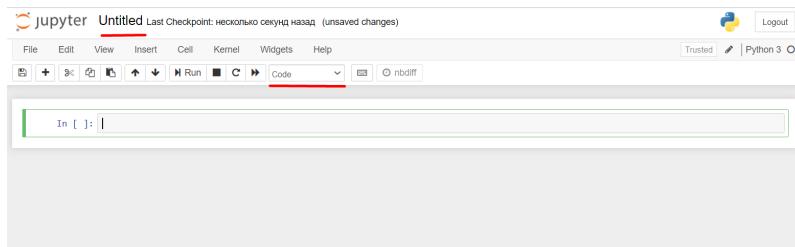


Fig. 14 Пример пустой только что созданной тетрадки.

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 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 немного 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**:

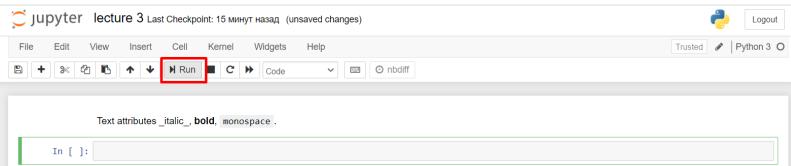


Fig. 15 Пример выполненной ячейки. Красным квадратом выделена кнопка Run.

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.

Multivariate Normal Distribution Equation

Recall the equation for the normal distribution from the **Gaussians** chapter:

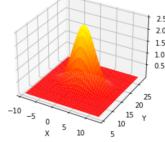
$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2}(x - \mu)^2/\sigma^2\right]$$

Here is the multivariate normal distribution in n dimensions.

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

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 $\mu = [2, 17]$ and a covariance of $\Sigma = [10, 0; 0, 4]$. The three dimensional shape shows the probability density for any value of (X, Y) in the x -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. 16 Пример грамотно оформленной Jupyter-тетрадки. Такую можно скинуть коллегам – и всем все будет понятно!

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 вставить **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**.

```
In [1]: print('Привет!')
Привет!
```

```
In [ ]: print('Привет!')
```

Fig. 17 Верхняя ячейка с `print` написана в Code-режиме и корректно выполняется, печатая строку приветствием. Нижняя ячейка же содержит текст, а не код, поэтому работать не будет (точнее, код отобразится как текст, но не будет выполнен).

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 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.

Режимы работы

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

```
In [5]: print('Еще одна ячейка!')
Еще одна ячейка!
```

```
In [5]: print('Привет!')
```

```
In [5]: print('Еще одна ячейка!')
Еще одна ячейка!
```

Fig. 18 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).

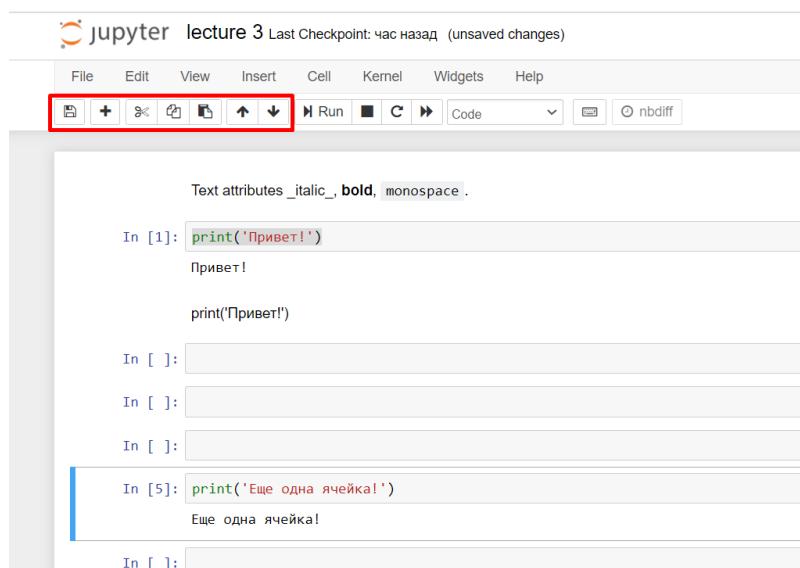


Fig. 19 Кнопки управления ноутбуком.

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 **пощаще**, чтобы не потерять результаты работы!
2. Создание ячейки ниже текущей (**B**) – **B** 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).



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.

Оставшиеся кнопки на панели

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

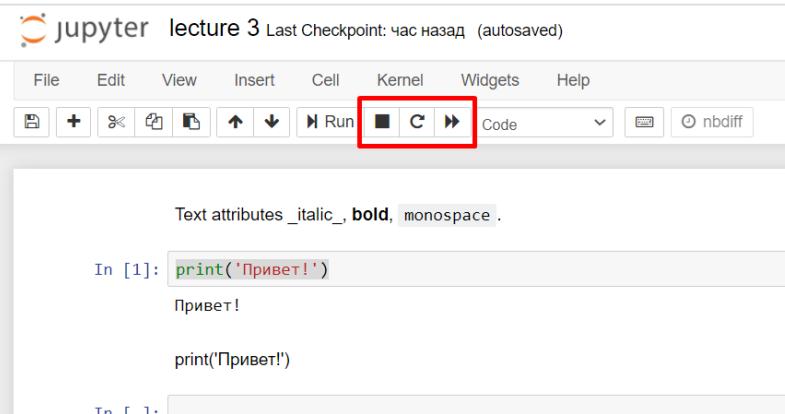


Fig. 20 Кнопки управления ЯДРОМ ноутбуком.

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.

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.

```

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. 21 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**ните) 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]**).

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
theta = 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(theta, 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 строке 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.

```
In [17]: the_answer_to_the_ultimate_question_of_life_the_universe_everything = 42
the_answer_but_divided_by_two = 21

In [ ]: the_an[  
the_answer_but_divided_by_two
In [ ]: the_answer_to_the_ultimate_question_of_life_the_universe_everything
```

Fig. 22 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!

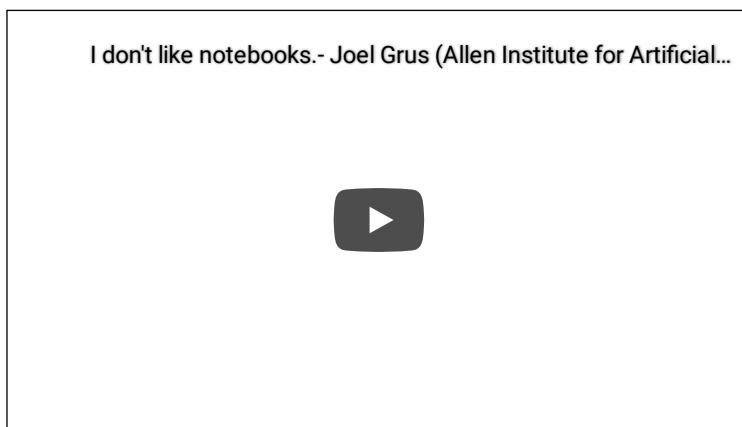
Что мы узнали из лекции

- 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.

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

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.

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/c
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 квадрат 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 (m) 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).

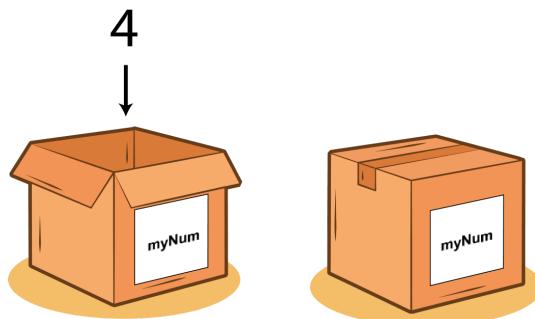


Fig. 23 [Объявить переменную – значит положить объект в коробку с подписью](#).

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).

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 трёх функций, включая приведение типа
print(type(int(c)))

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

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'>
```

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!"

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
```

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 приводят типы).

Условные конструкции, булева логика и сравнения

Описание лекции

В этой лекции мы расскажем про:

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

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 ($-(-5) = 5$). Эту 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 – дополнительно – 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.

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.

Способы получения `bool`

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

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

Fig. 24 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
        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 операнд. Логическое **НЕ** возвращает **True, если операнд 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.

Блоки кода и отступы

В примерах выше вы наверняка заметили упоминание термина “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, указывающим на начало строки кода) создаст отступ в четыре пробела. Пользуйтесь этим, чтобы не перегружать клавишу пробела лишними постукиваниями :).

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

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.

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”[[https://ru.wikipedia.org/wiki/Контейнер_\(программирование\)](https://ru.wikipedia.org/wiki/Контейнер_(программирование))], 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**:

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

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 ошибетесь 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 единственным 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.

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).

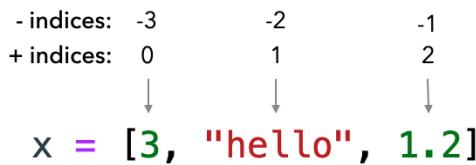


Fig. 25 Пример списка из трех объектов. Сверху показаны их индексы, включая отрицательные

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 -

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)]
print(f"Верно ли, что единица входит в some_slice? {1 in some_slice}")
print(f"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 **вычислимые** (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 её 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.

Циклы

До сих пор в примерах мы хоть и обращались к некоторым 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")
```

```

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 = []

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.

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**”.

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 отдельного 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]`).

О блоке

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. 26 Как “видит” квантовый мир пара нейросетей VQGAN и CLIP, сгенерировано в [Google Colab](#)

Машинное обучение как пазл

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. 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.

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.

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.

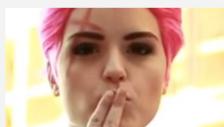
Note

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.

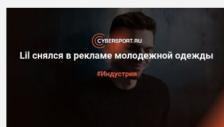
Пример 1. Рекомендация новостного контента

Новостному порталу хочется понять, какой контент нравится пользователям и по каким ссылкам они будут кликать. Здесь *событием* будет то, что пользователю нравится рекомендуемый контент.

Читайте также



Персонажи Overwatch
разделились для аниме-
фестиваля. Видео



Lil снялся в рекламе
молодежной одежды



«Настоящая зависимость, из
которой сложно вытащить».
Как Первый канал смешал

Fig. 27 Блок “Читайте также” на новостном портале

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.

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, чтоб это делали эксперты уровня редактора журнала, а еще и лучше бы 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.

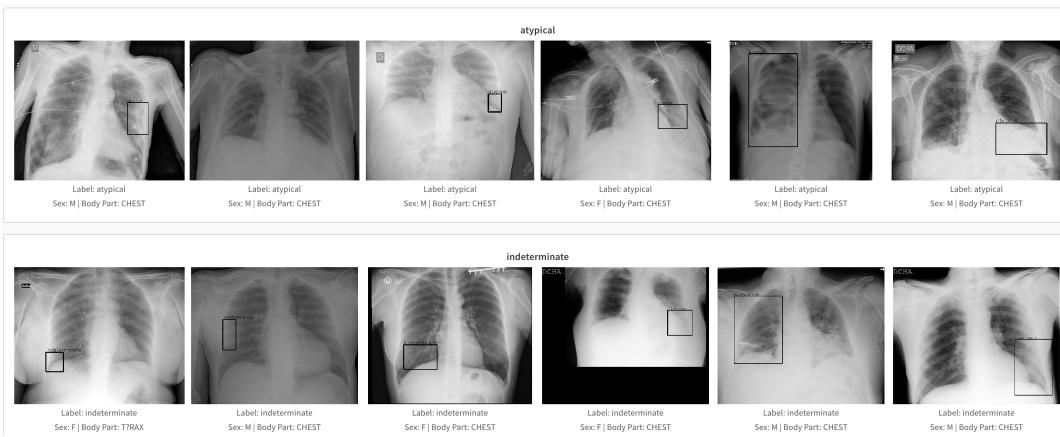


Fig. 28 Пример данных [соревнования](#) по определению аномалий на рентгенограммах грудной клетки.

[Источник](#)

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 признаком в обучающей выборке.

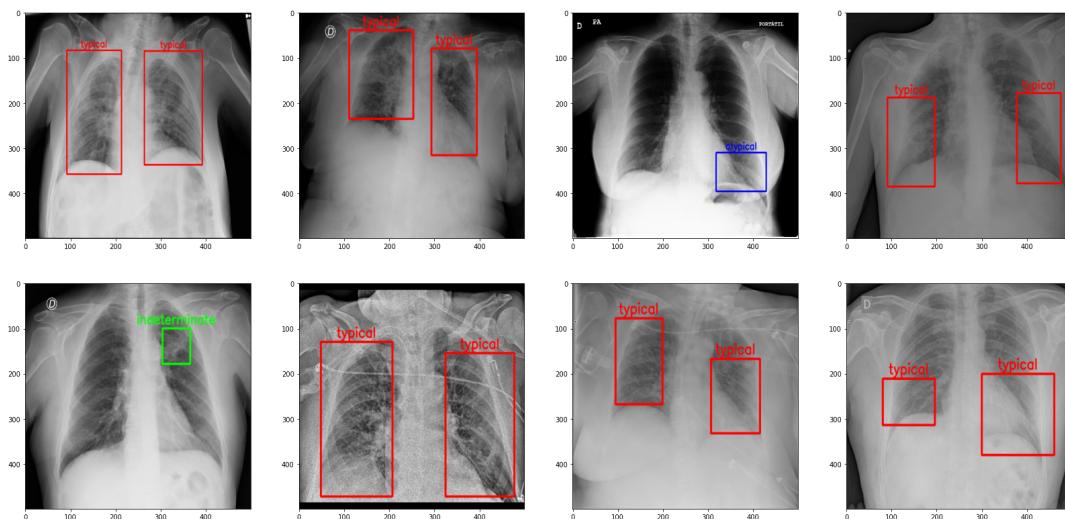


Fig. 29 Прогноз модели детекции, обученной на данных [соревнования](#) по определению аномалий на рентгенограммах грудной клетки. [Источник](#)

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.

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, разпознавание 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. 30 Пример обнаружения объектов на изображении. [Источник](#)

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 “сочлаась”.

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. 31 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 Дэлл 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

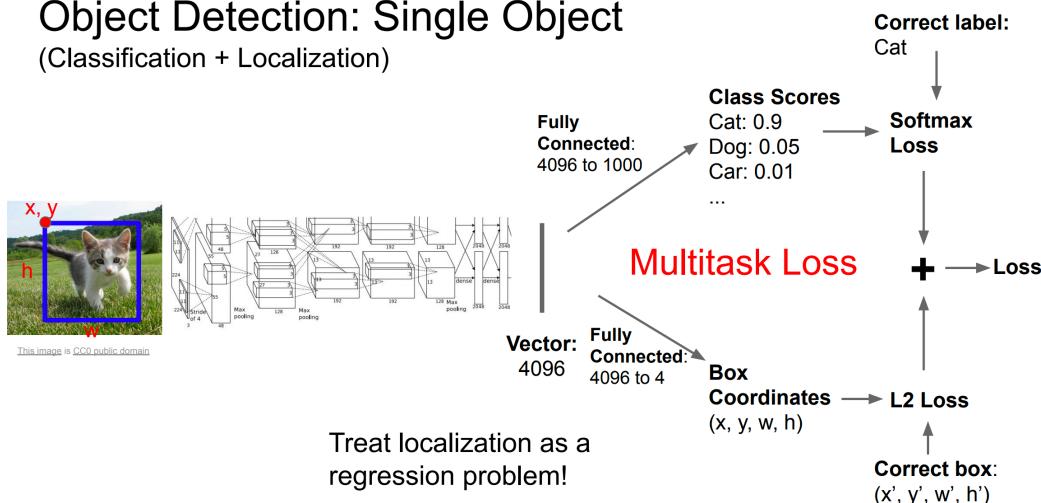
Тут задача свелась к задаче регрессии с текстовыми входными данными, и можно использовать простую функцию потерь – среднеквадратичную ошибку (Mean Squared Error). Подробнее о функциях потерь в задаче регрессии можно прочитать в [посте](#) А.Г. Дьяконова.

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. 32 Слайд стэнфордского курса cs231n про функцию потерь в задаче детекции объектов.

[Источник.](#)

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 ТПУ.

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 такие 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.

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.

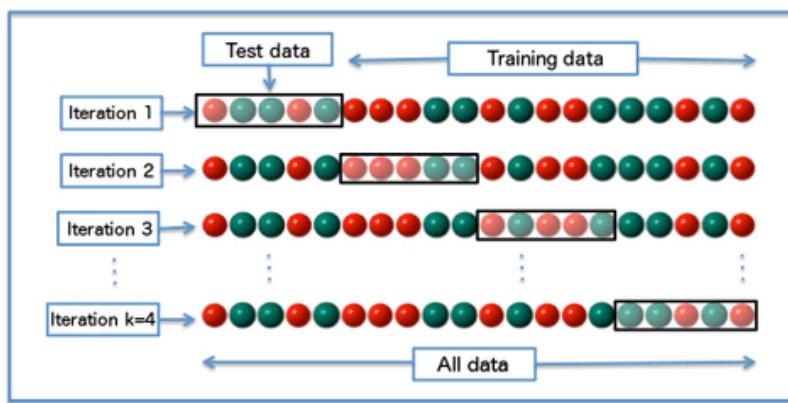


Fig. 33 Схема K-fold кросс-валидации.

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 A/B-тесты, которые позволяют 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. A/B-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, приводят 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 A/B-тест (а точнее, для задач ранжирования намного лучше использовать [интерливинг](#)). 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 (A/B-тест, интерливинг) помимо оффлайн-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 дорогостоящим и 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.

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, чтобы 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.

О блоке

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;
- об алгоритмах Шора и Гровера.

Квантовый бит

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.

Введение

Это первая лекция основного блока нашего курса. Прежде чем мы начнем детально разбирать понятие кубита, давайте взглянем на общий пайплайн квантовых схем.

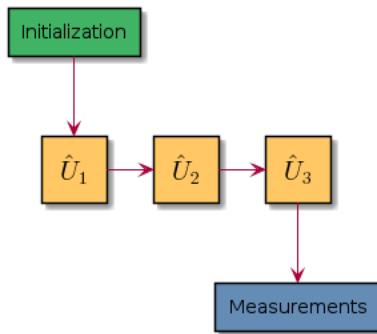


Fig. 34 Схема любого квантового алгоритма

Любая квантовая схема включает в себя:

- кубиты, инициализируемые в начальное состояние, обычно $|\text{ket}{0}\rangle$;
- 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.

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 квантомеханических систем (и кубитов в частности) их состояния и значения – это не одно и то же.

Состояние vs значение

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**.

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 *суперпозиции* состояний “жив” и “мертв” одновременно.

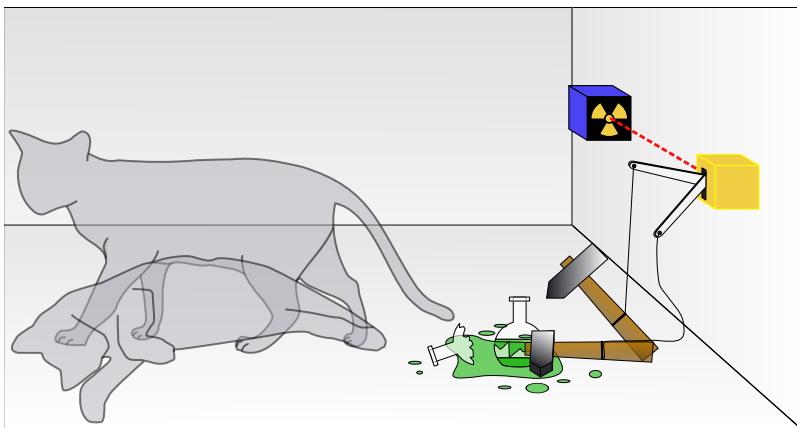


Fig. 35 Код Шредингера

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 $|\Psi\rangle$ (кет – вектор-столбец) – это широко принятая в квантовой механике и квантовых вычислениях нотация Дирака:

$$|\Psi\rangle = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$$

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.

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,光子和其他粒子表现出了作为波的性质(干涉和衍射),而同时它们也表现出了作为粒子的性质(最小波长). 例如, 光的波动性(波函数)描述了光的强度分布, 而光的粒子性(量子)描述了光的能量分布. 因此, 在描述量子力学时, 我们需要同时考虑这两种性质.

Для более детального ответа авторы курса рекомендуют читать книги по истории квантовой физики (и по самой квантовой физике).

Значение чисел $|c_0\rangle$ и $|c_1\rangle$ мы обсудим чуть позже, а пока запишем наш кубит $|\Psi\rangle$ 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) .

```
print(qubit.shape)
```

```
(2, 1)
```

Связь состояния и значения кубита

Разберем подробнее вектор $|\Psi\rangle$ и значение цифр $|c_0, c_1\rangle$. 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:

```
\begin{bmatrix} |0\rangle \\ |1\rangle \end{bmatrix}
```

Что мы можем сказать об этих состояниях? Как минимум следующее:

- они ортогональны ($\langle 0 | 1 \rangle = 0$);
- они имеют единичную норму;
- они образуют базис.

Что это значит для нас? А то, что любое состояние $|\Psi\rangle$ можно записать как линейную комбинацию векторов $|0\rangle$ и $|1\rangle$, причем коэффициентами в этой комбинации будут как раз наши $|c_0, c_1\rangle$:

```
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
```

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 $\langle c_0 | c_1 \rangle = \frac{1}{\sqrt{2}}$ 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
```

Мы будем очень часто пользоваться транспонированием и взятием комплексно сопряженного от векторов. В квантовой механике это имеет специальное обозначение $\langle \Psi | \Psi \rangle = \Psi^\dagger \Psi$ (бра – вектор-строка). Тогда наше правило нормировки из =NumPy= кода может быть записано в нотации Дирака так: $|\langle \Psi | \Psi \rangle| = 1$

Сфера Блоха

Описанный выше базис $|\Psi_0\rangle, |\Psi_1\rangle$ не является единственным возможным. Вектора $|\Psi_0\rangle, |\Psi_1\rangle$ – это лишь самый часто применимый базис, который называют \mathbf{Z} базисом. Но есть и другие варианты.

Возможные базисы

Z-базис

Уже описанные нами $|\Psi_0\rangle$ и $|\Psi_1\rangle$.

X-базис

Базисные состояния $|\Psi_+\rangle = \frac{1}{\sqrt{2}}(|\Psi_0\rangle + |\Psi_1\rangle)$ и $|\Psi_-\rangle = \frac{1}{\sqrt{2}}(|\Psi_0\rangle - |\Psi_1\rangle)$:

```
plus = (basis_0 + basis_1) / np.sqrt(2)
minus = (basis_0 - basis_1) / np.sqrt(2)
```

Y-базис

Базисные состояния $|\Psi_R\rangle = \frac{1}{\sqrt{2}}(|\Psi_0\rangle + i|\Psi_1\rangle)$ и $|\Psi_L\rangle = \frac{1}{\sqrt{2}}(|\Psi_0\rangle - i|\Psi_1\rangle)$:

```
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))
print(np.allclose(np.conj(plus).T @ minus, 0))
print(np.allclose(np.conj(R).T @ L, 0))
```

```
True
True
True
```

Заметьте, что в наших векторных пространствах скалярное произведение – это $\langle \vec{a} | \vec{b} \rangle = \langle a | b \rangle$ (бра-кет). Именно поэтому нужно делать транспонирование и комплексное сопряжение первого вектора в паре.

Сфера Блоха

Обозначения $|\Psi\rangle, |\Phi\rangle, |\Psi+\rangle, |\Psi-\rangle, |\Psi_R\rangle, |\Psi_L\rangle$ выбраны неслучайно: они имеют геометрический смысл.

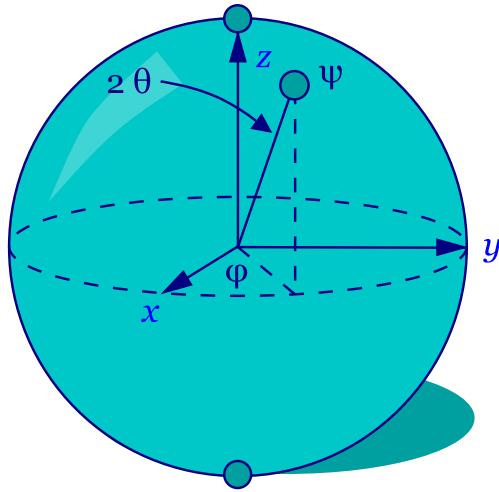


Fig. 36 Сфера Блоха

Принято считать, что ось \mathbf{Z} – это основная ось, так как физически квантовые компьютеры измеряют именно по ней. Ось \mathbf{X} “смотрит на нас” и поэтому обозначается $|\Psi+\rangle$ и $|\Psi-\rangle$. А ось \mathbf{Y} – “право” ($|\Psi_R\rangle$) и “лево” ($|\Psi_L\rangle$).

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:

$$|\Psi\rangle = c_0 |\Psi_0\rangle + c_1 |\Psi_1\rangle = \cos\theta |\Psi_0\rangle + e^{i\phi} \sin\theta |\Psi_1\rangle$$

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 базовую лекцию по линейной алгебре и комплексным числам, там эти моменты освещаются более подробно.

Что можно делать с таким кубитом?

Линейные операторы

Любое действие, которое мы совершаем с кубитом в состоянии $|\Psi\rangle$, должно переводить его в другое состояние $|\Phi\rangle$. Что переводит один вектор в другой вектор в том же пространстве? Правильно, матрица. Другими словами, линейный оператор. Мы будем обозначать операторы как $|\hat{U}\rangle$.

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 $\langle \hat{U} | \Psi \rangle = \langle \Phi | \hat{U} | \Psi \rangle$, 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: $\langle \Psi | \hat{U}^\dagger \hat{U} | \Psi \rangle = \langle \Psi | \Psi \rangle$. Другими словами, унитарный оператор удовлетворяет условию $\langle \hat{U}^\dagger \hat{U} | \Psi \rangle = \langle \Psi |$.

Обратимость

Одно из важных следствий унитарности операций над кубитами – это их обратимость. Если вы сделали какую-то последовательность унитарных операций над кубитами $\langle \hat{U}_1 \hat{U}_2 \dots \hat{U}_n | \Psi \rangle$, то их можно вернуть в начальное состояние, ведь у унитарного оператора всегда есть обратный оператор $\langle \hat{U}_1 \hat{U}_2 \dots \hat{U}_n |^{-1} = \langle \hat{U}_n \dots \hat{U}_1 |$.

Note

Квантовый компьютер должен уметь делать несколько не унитарных операций, например, инициализацию кубита в определенное состояние (например, $|0\rangle$) и считывание состояния кубитов. Такие неунитарные операции приводят к потере информации и являются необратимыми.

Пример оператора

В дальнейших лекциях мы разберем много операторов, так как именно операторы (или **квантовые гейты**) являются основой квантовых вычислений. А пока посмотрим простой пример: оператор Адамара (**Hadamard gate**), который переводит $|0\rangle$ в $|+\rangle$.

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 $|0\rangle$ в $|+\rangle$.

$$\begin{bmatrix} \hat{H} & \\ \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

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) @ h, np.eye(2)))
```

True

Правильное действие

```
print(np.allclose(h @ basis_0, plus))
```

True

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 Эверетту 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, у кубита может быть несколько разных базисов: $|\langle 0 \rangle, |\langle 1 \rangle, |\langle + \rangle, |\langle - \rangle, |\langle R \rangle, |\langle L \rangle$. Значение кубита в каждом из этих базисов может быть измерено. Но что такое измерение с точки зрения математики?

Операторы Паули

На самом деле, любая наблюдаемая величина соответствует какому-то оператору. Например, измерения в разных базисах $|\langle X \rangle, |\langle Y \rangle, |\langle Z \rangle$ соответствуют операторам Паули:

```
\begin{split} \hat{\sigma}_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad \hat{\sigma}_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \\ \hat{\sigma}_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \end{split}
```

```
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.

Собственные значения

Мы поняли, что есть связь между нашими измерениями и операторами. Но какая именно? Что значит, например, что измерения по оси $|\langle Z \rangle$ соответствуют оператору $|\langle \hat{\sigma}_Z \rangle|$?

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.

Собственные вектора $|\langle \hat{\sigma}_Z \rangle|$

Вернемся к нашему оператору $|\langle \hat{\sigma}_Z \rangle|$. Легко убедиться, что его собственные значения равны 1 и -1, а соответствующие им собственные вектора – это $|\langle 1 \rangle, |\langle 0 \rangle$:

```
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 $|\langle Z \rangle|$ всегда будет давать нам одно из этих двух значений и переводить состояние кубита в соответствующий собственный вектор.

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:

$$\hat{U}|\Psi\rangle = u|\Psi\rangle$$

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

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**.

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 $|\Psi\rangle$? Другими словами, какое будет математическое ожидание большого числа измерений? Это можно записать следующим образом:

$$\langle \hat{U} \rangle = \langle \Psi | \hat{U} | \Psi \rangle$$

Например, оператор $\hat{\sigma}_z$ полностью не определен в состоянии $|\Psi\rangle$, 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]]
```

С другой стороны, измеряя состояние $|\Psi\rangle$ в X-базисе мы всегда будем получать 1:

```
print(plus.conj().T @ pauli_x @ plus)

[[1.+0.j]]
```

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, $|0 < 1\rangle$, $|00 < 01 < 10 < 11\rangle$, и т.д.), то вероятность каждой битовой строки получается следующим выражением:

$$P = |\langle \Psi | \vec{v} \rangle|^2$$

где \vec{v} – 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 $|\langle \Psi | \vec{v} \rangle|^2$ – это идентично и есть плотность вероятности.

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 $\langle \Phi | \Psi \rangle$ проекция будет линейным оператором:

$$\hat{P} = |\Phi\rangle\langle\Phi|$$

```
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))
```

Правило Борна

Вероятность наблюдения каждого из собственных значений λ какого-то оператора \hat{U} :

определяется как результат измерения оператора проекции на соответствующий собственный вектор:

$$|\langle \Psi | \lambda | \Psi \rangle|^2$$

Считать ожидаемое значение оператора мы уже умеем. Давайте убедимся, что для состояния $|\Psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ 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
p_1 = super_position.conj().T @ proj_1 @ super_position

print(np.allclose(p_0 + p_1, 1.0))
print(np.allclose(p_0, 0.5))
```

```
True
True
```

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

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](#).

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:

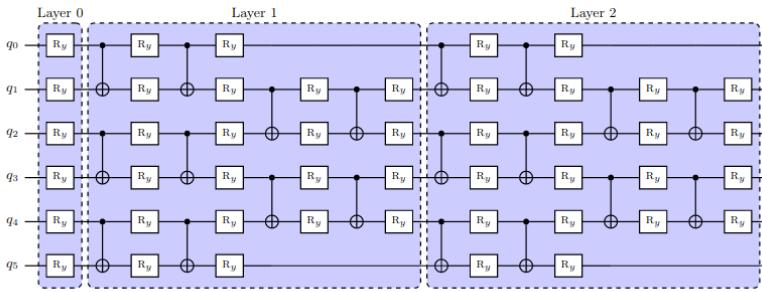


FIG. 2: L-VQE ansatz for a 6-qubit quantum state. R_y denotes rotation around the y -axis defined as $R_y(\theta) \equiv e^{-i\frac{\theta}{2}Y}$. Every R_y contains a parameter that is optimized over in the outer loop.

Fig. 37 Схема Layered-VQE

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.

T-гейт

T-гейт очень популярен в универсальных квантовых вычислениях. Его матрица имеет вид:

$$\begin{bmatrix} \hat{T} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1+i}{\sqrt{2}} \end{bmatrix} \end{bmatrix}$$

Любой однокубитный гейт можно аппроксимировать последовательностью гейтов Адамара и T-гейтов. Чем точнее требуется аппроксимация, тем длиннее будет аппроксимирующая последовательность.

Помимо важной роли в математике квантовых вычислений, гейт Адамара и T-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.

Гейты поворота вокруг оси

Поворотные гейты играют центральную роль в квантовом машинном обучении. Вспомним на секунду, как выглядят наши однокубитные состояния на сфере Блоха:

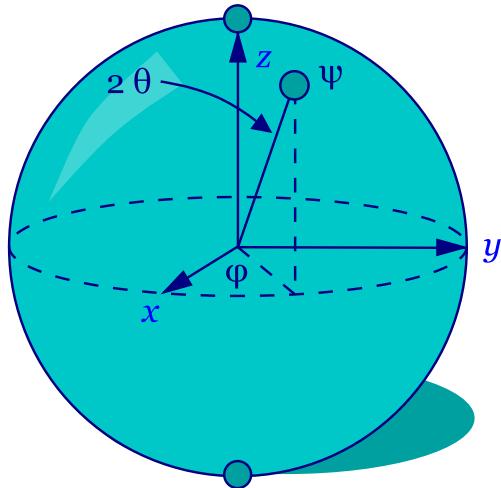


Fig. 38 Сфера Блоха

Любой однокубитный гейт можно представить как вращение вектора состояния $|\Psi\rangle$ 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 $\hat{R}Y(\phi)$.

Гейт $\hat{R}Y(\phi)$

Сам гейт определяется следующим образом:

```
\begin{split} \hat{R}Y(\phi) = \begin{bmatrix} \cos(\frac{\phi}{2}) & -\sin(\frac{\phi}{2}) \\ \sin(\frac{\phi}{2}) & \cos(\frac{\phi}{2}) \end{bmatrix} \end{split}
```

```
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 $|0\rangle$:

```
basis = np.array([1 + 0j, 0 + 0j]).reshape((2, 1))
```

Внимательно посмотрим на сферу Блоха. Можно заметить, что если повернуть состояние из $|0\rangle$ на π и измерить значение $\langle \hat{\sigma}_z \rangle$, то получится 1. А если повернуть на $-\pi$, то получится 0:

```
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:

```
\begin{split} |\Psi\rangle = \begin{bmatrix} 0.42 \\ \sqrt{1 - 0.42^2} \end{bmatrix} \end{split}
```

```
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" + str(expval(random_state, pauli_z)) + "\n")
print("X:\n" + str(expval(random_state, pauli_x)) + "\n")

print("Z after RY:\n" + str(expval(ry(random_state, 2 * np.pi), pauli_z)) + "\n")
print("X after RY:\n" + 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 $\hat{R}X(\phi)$ и $\hat{R}Z(\phi)$:

```
\begin{split} \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}
 \hat{R}Z(\phi) = \begin{bmatrix} e^{-i\frac{\phi}{2}} & 0 \\ 0 & e^{i\frac{\phi}{2}} \end{bmatrix} \end{split}
```

Общая форма записи однокубитных гейтов

В общем случае однокубитные гейты могут быть также записаны следующим образом:

$$\begin{aligned} \left[\begin{array}{c} \hat{R}^{\alpha} \\ \vec{n} \end{array} \right] &= e^{-i\frac{\alpha}{2}\hat{\sigma}_z} \begin{bmatrix} \cos(\frac{\alpha}{2}) & -e^{i\frac{\alpha}{2}} \sin(\frac{\alpha}{2}) \\ e^{i\frac{\alpha}{2}} \sin(\frac{\alpha}{2}) & \cos(\frac{\alpha}{2}) \end{bmatrix} \begin{bmatrix} \cos(\theta) & -e^{i\phi} \sin(\theta) \\ e^{i\phi} \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} \vec{n}_x \\ \vec{n}_y \\ \vec{n}_z \end{bmatrix} \\ &\text{где } \alpha \text{ – это угол поворота, } \vec{n} \text{ – единичный вектор в направлении оси поворота, а } \hat{\sigma}_z \text{ –} \end{aligned}$$

= $\hat{\sigma}_z$ – это вектор, составленный из операторов Паули. Если использовать покоординатную запись и $\vec{n} = [n_x, n_y, n_z]$ задает ось вращения, то

$$\left[\begin{array}{c} \hat{R}^{\alpha} \\ \vec{n} \end{array} \right] = e^{-i\frac{\alpha}{2}} \begin{bmatrix} \cos(\theta) & -e^{i\phi} \sin(\theta) \\ e^{i\phi} \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} \vec{n}_x \\ \vec{n}_y \\ \vec{n}_z \end{bmatrix}$$

Забегая вперед, можно сказать, что именно гейты вращений – это основа квантовых вариационных схем, главного инструмента этого курса.

Phase-shift гейт

Другой важный гейт – это так называемый phase-shift гейт или \hat{U}_1 гейт. Его матричная форма имеет следующий вид:

$$\begin{aligned} \left[\begin{array}{c} \hat{U}_1(\phi) \end{array} \right] &= \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix} \end{aligned}$$

```
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.

Гейты \hat{U}_2 и \hat{U}_3

Более редкие в QML гейты, которые однако все равно встречаются в статьях.

$$\begin{aligned} \left[\begin{array}{c} \hat{U}_2(\phi, \lambda) \end{array} \right] &= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi + \lambda)} \end{bmatrix} \end{aligned}$$
$$\left[\begin{array}{c} \hat{U}_3(\theta, \phi, \lambda) \end{array} \right] = \hat{U}_1(\phi + \lambda) \hat{R}_Z(-\lambda) \hat{U}_1(\theta) \hat{R}_Y(\frac{\phi}{2}) \hat{R}_Z(\lambda) \hat{U}_1(\phi + \lambda)$$

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 $\hat{U}_3(\theta, \phi, \lambda)$:

$$\begin{aligned} \left[\begin{array}{c} \hat{U}_3(\theta, \phi, \lambda) \end{array} \right] &= \begin{bmatrix} \cos(\frac{\theta}{2}) & -e^{i\frac{\phi}{2}} \sin(\frac{\theta}{2}) \\ e^{i\frac{\phi}{2}} \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{bmatrix} \begin{bmatrix} 1 & -e^{i\lambda} \\ e^{i\lambda} & e^{i(\phi + \lambda)} \end{bmatrix} \end{aligned}$$

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 .

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 $|0\rangle$. Так как мы всегда предполагаем, что начальное состояние кубитов – это именно $|0\rangle$, 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 - 1j]])
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} :

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

```

i = np.eye(2, dtype=np.complex128)
print(i)

```

```

[[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.

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.

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 \mathbb{C}^4 , состояние трех кубитов – вектор в пространстве \mathbb{C}^8 , то есть состояние N кубитов описывается вектором размерности 2^N в комплексном пространстве. Вероятности каждой из возможных битовых строк ($0000\dots00$), ($0000\dots01$), ($0000\dots10$), и так далее) получаются по методу Шредингера, который мы обсуждали в конце прошлой лекции:

$$|\Psi\rangle = |\psi\rangle$$

Нужно отсортировать наши битовые строки в лексикографическом порядке – и вероятность i -й битовой строки будет равна квадрату i -го элемента вектора $|\Psi\rangle$.

Формально многокубитные состояния описываются с помощью математического концепта так называемого тензорного произведения, иначе – произведения Кронекера, обозначаемого значком \otimes . Так, если $|\Psi\rangle_A \otimes |\Psi\rangle_B$, то $|\Psi\rangle_{AB} = |\Psi\rangle_A \otimes |\Psi\rangle_B$. $|\Psi\rangle_A \otimes |\Psi\rangle_B = |\Psi\rangle_A \otimes |\Psi\rangle_B$. О том, как элементы вектора $|\Psi\rangle_{AB}$ выражаются через элементы векторов $|\Psi\rangle_A$ и $|\Psi\rangle_B$, можно прочитать на Википедии в статье [“Произведение Кронекера”](#).

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 $(2^N \times 2^N)$.

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:

$$\begin{aligned} \hat{H} \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

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, i) @ np.kron(basis, basis)))
```

True

Наблюдаемые для многокубитных гейтов

Аналогичным образом можно сконструировать и наблюдаемые. Например, если мы хотим измерять одновременно два спина по оси \hat{Z} , то наблюдаемая будет выглядеть так:

$$\begin{aligned} \hat{\mathbf{Z}}\hat{\mathbf{Z}} = \hat{\sigma}_z \otimes \hat{\sigma}_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \end{aligned}$$

```
print(np.kron(basis, basis).conj().T @ np.kron(pauli_z, pauli_z) @ np.kron(basis, basis))
```

$[[1.+0.j]]$

Основные двухкубитные гейты

Основные многокубитные гейты, которые предоставляют современные квантовые компьютеры, – это двухкубитные гейты.

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.

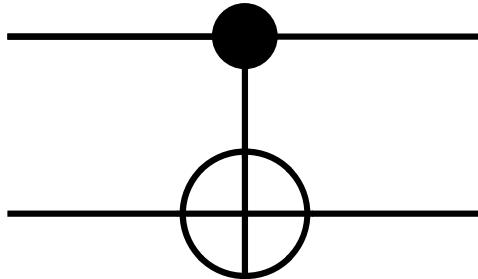


Fig. 39 Гейт CNOT

Иногда этот гейт также называют гейтом CX. В матричном виде этот оператор можно записать так:

$$\hat{\text{CNOT}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

```
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 $(|0\rangle)$ в $(|1\rangle)$ и наоборот.

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:

$$\begin{aligned} \hat{CY} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{bmatrix} \\ \hat{CZ} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \end{aligned}$$

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:

$$\hat{CZ} = \left(\hat{I} \otimes \hat{H} \right) \hat{CX} \left(\hat{I} \otimes \hat{H} \right)$$

Этим свойством обладают отнюдь не все двухкубитные гейты. Например, таковым является гейт iSWAP:

$$\begin{aligned} \text{iSWAP} &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & -i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

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:

$$\begin{aligned} \text{fSim}(\theta, \phi) &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -i\sin\theta & 0 \\ 0 & i\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & e^{-i\phi} \end{bmatrix} \end{aligned}$$

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”.

Первое знакомство с PennyLane

На сегодняшний день существует достаточно много фреймворков для программирования квантовых компьютеров. Для целей этого курса мы будем использовать [PennyLane](#). Он предоставляет высокоуровневый [Python API](#) и создан специально для решения задач квантового машинного обучения.

```
import pennylane as qml
```

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:

```
device = qml.device("default.qubit", 2)
```

Первый аргумент тут – указание устройства, а второй – число кубитов.

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
```

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.

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 $\langle R^{\dagger} \vec{v} | \alpha \rangle$.
- Вокруг какой оси и на какой угол вращает состояние гейт Адамара?
- Гейт SWAP меняет кубиты местами. Его унитарная матрица имеет вид:

```
\begin{split} \mathrm{SWAP} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{split}
```

Попробуйте составить последовательность гейтов, реализующую $\langle \mathrm{SWAP} \rangle$, из гейтов $\langle \mathrm{iSWAP} \rangle$, $\langle \hat{\mathrm{CZ}} \rangle$ и $\langle \hat{\mathrm{RZ}}(\phi) \rangle$.

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 $|0\rangle$, либо $|1\rangle$. 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:

1). $|f_1(x) = 0\rangle$

Это функция, всегда возвращающая $|0\rangle$, независимо от входного значения.

Для нее справедливы выражения:

$$|f_1(0) = 0\rangle$$

$$|f_1(1) = 0\rangle$$

2). $|f_2(x) = 1\rangle$

Такая функция всегда возвращает $|1\rangle$, то есть верно следующее:

$$|f_2(0) = 1\rangle$$

$$|f_2(1) = 1\rangle$$

Ну а теперь посмотрим на сбалансированные функции.

Для них характерно то, что они могут возвращать как $|0\rangle$, так и $|1\rangle$. В этом и заключается "баланс".

3). $|f_3(x) = x\rangle$

Это тождественная функция, которая ничего не делает с входным значением.

Для нее справедливо следующее:

$$|f_3(0) = 0\rangle$$

$$|f_3(1) = 1\rangle$$

4). $|f_4(x) = \overline{x}\rangle$

А вот эта функция инвертирует входное значение, то есть возвращает не то число, которое было подано на вход, а другое:

$$|f_4(0) = 1\rangle$$

$$|f_4(1) = 0\rangle$$

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 $|0\rangle$. Допустим, мы получили на выходе также $|0\rangle$. Мы можем сказать, что данная функция - либо $|f_1\rangle$ (константная функция, всегда возвращающая $|0\rangle$), либо $|f_3\rangle$ (сбалансированная функция, не меняющая входное значение). Для окончательного решения мы должны сделать еще один шаг - отправить в функцию значение $|1\rangle$. Если при этом мы получим опять $|0\rangle$, то это функция $|f_1\rangle$, а если получили на выходе $|1\rangle$, то искомая функция - $|f_3\rangle$.

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:

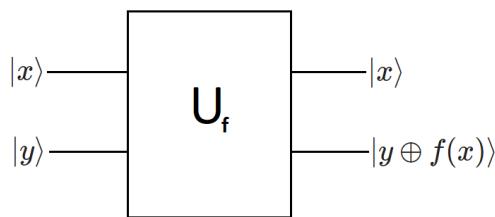


Fig. 40 Схема 1.

$|U_f\rangle$ 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 $|y\rangle$ 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 $|y\rangle$ и $|f(x)\rangle$.

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:

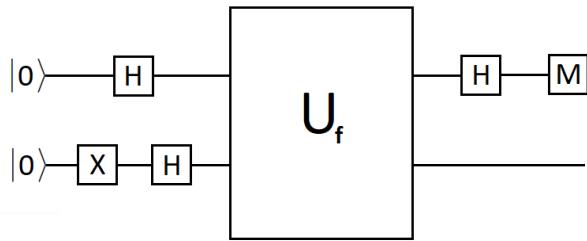


Fig. 41 Схема 2.

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:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

а нижний - в такой:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

Если рассмотреть это как систему $|\psi\rangle$, состоящую из двух кубитов, то она будет выглядеть так:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle)$$

Сразу после U_f система будет находиться в состоянии:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle)$$

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 действует оператор Адамара и будет произведено измерение.

В том случае, если функция сбалансирована, результат измерения верхнего кубита будет равен $|1\rangle$, а если несбалансированна - $|0\rangle$.

Разберемся подробнее, почему это происходит.

Рассмотрим все возможные $|f(x)\rangle$, которые могут находиться в черном ящике:

1). $|f(x) = f_1\rangle$

В этом случае $|f(x)\rangle$ всегда принимает значение $|0\rangle$, и система кубитов будет выглядеть так:

$$|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle - |1\rangle) = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)$$

2). $|f(x) = f_2\rangle$

$|f(x)\rangle$ будет равно $|1\rangle$ 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 $|f_1\rangle$ и $|f_2\rangle$, являющихся несбалансированными, мы получаем фактически одно и тоже состояние. Если после этого применить к первому кубиту оператор Адамара, то после измерения мы получим значение $|0\rangle$.

Рассмотрим теперь сбалансированные функции $|f_3\rangle$ и $|f_4\rangle$.

3). $|f(x) = f_3\rangle$

Здесь ситуация сложнее, так как $|f(x)\rangle$ будет зависеть от состояния первого кубита. Поэтому мы раскроем скобки, а значения функции подставим позже:

$$|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle + f(x)|1\rangle - |1\rangle + f(x)|0\rangle) = \frac{1}{2}(|0\rangle |0\rangle + f(x)|1\rangle - |0\rangle |1\rangle + f(x)|0\rangle - |1\rangle |1\rangle + f(x)|0\rangle) =$$

$$= \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) = \frac{1}{2}(|0\rangle |0\rangle + |1\rangle |0\rangle - |0\rangle |1\rangle - |1\rangle |1\rangle) =$$

$$= \frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle) = \frac{1}{2}(|0\rangle - |1\rangle)(|0\rangle - |1\rangle)$$

Видно, что первый кубит поменял свое состояние - теперь он в суперпозиции $(\sqrt{2}/2)(|0\rangle - |1\rangle)$, так что далее к нему можно применить оператор Адамара, после которого он перейдет в состояние $|1\rangle$.

4). $|f(x) = f_4\rangle$

Здесь будет похожая ситуация:

$$|\psi\rangle = \frac{1}{2}(|0\rangle + |1\rangle)(|0\rangle + f(x)|1\rangle - |1\rangle + f(x)|0\rangle) = \frac{1}{2}(|0\rangle |0\rangle + f(x)|1\rangle - |0\rangle |1\rangle + f(x)|0\rangle - |1\rangle |1\rangle + f(x)|0\rangle) =$$

$$= \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) = \frac{1}{2}(|0\rangle |0\rangle + |1\rangle |0\rangle - |0\rangle |1\rangle - |1\rangle |1\rangle) =$$

$$= -\frac{1}{2}(|00\rangle - |01\rangle + |11\rangle - |10\rangle) = -\frac{1}{2}(|0\rangle - |1\rangle)(|0\rangle - |1\rangle)$$

Получили то же состояние $|\psi\rangle$, что и для $|f_3\rangle$, с точностью до фазового множителя. Соответственно, здесь первый кубит после применения оператора Адамара также будет измерен с результатом $|1\rangle$.

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 $|f(x)\rangle$. (Задание рекомендуется сделать до прочтения программистской части по алгоритму Дойча, так как там содержится ответ).

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 её, точнее, все её четыре варианта. Для того, чтобы нам 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 1 (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 = ['f1', 'f2', 'f3', 'f4']
    n = np.random.randint(0, 4)
    return black_boxes[n]
```

А теперь самое важное - сам алгоритм Дойча:

```
@qml.qnode(dev)
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. Для начала выберем случайным образом функцию:

```
black_box_name = random_black_box()
```

А затем запустим алгоритм Дойча и выведем результат его работы. Собственное значение |1⟩ оператора Z будет соответствовать состоянию |0⟩ (функция несбалансированна), а собственное значение |-1⟩ - состоянию |1⟩ (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.

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 \sqrt{N} операций, где \sqrt{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 $\sqrt{\sqrt{N}}$ действий. Конечно же, такое ускорение не будет экспоненциальным, как при использовании некоторых других квантовых алгоритмов, но оно будет квадратичным, что также довольно неплохо.



Fig. 42 Лов Гровер

❶ 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 ($|Id\rangle$) записи, которая удовлетворяет определенным условиям. Функция-оракул находит такую запись (для простоты будем сначала считать, что такая запись одна) и помечает соответствующий ей $|Id\rangle$. Отметка делается достаточно оригинальным способом: $|Id\rangle$ умножается на $|-1\rangle$.

Для полной ясности соотнесем количество $|Id\rangle$ 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 $|Id\rangle$ 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 $\sqrt{-1}$). Все эти четыре числа могут существовать в квантовой схеме одновременно, если кубиты приведены в состояние суперпозиции.

Пусть искомый $|Id\rangle$ равен $|11\rangle$ (будем пользоваться двоичной системой и вести счет с нуля), тогда после работы функции-оракула мы будем иметь $|4\rangle$ 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 $|Id\rangle$ 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:

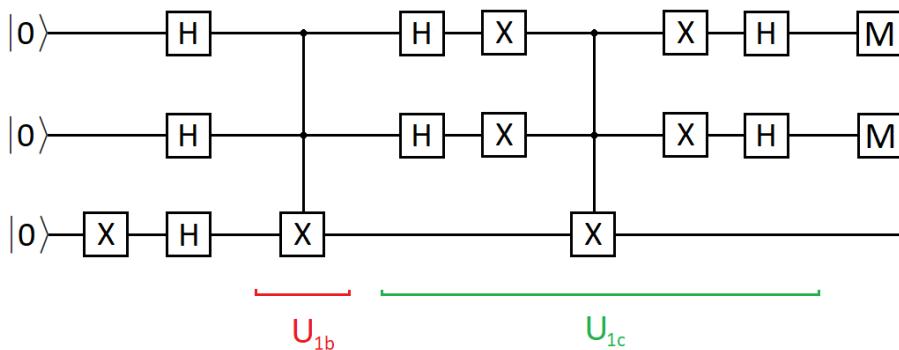


Fig. 43 Алгоритм Гровера для $n=2$ (искомый индекс 11).

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 $|0\rangle$, кроме вспомогательного кубита – до действия оператора Адамара он должен быть приведен в состояние $|1\rangle$.

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 $|11\rangle$, 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 $|1\rangle$, он будет применять к управляемому кубиту (это как раз будет вспомогательный кубит) гейт X .

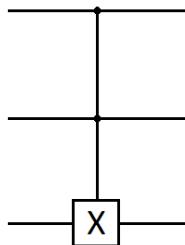


Fig. 44 Гейт Тoffоли.

На состояния верхних кубитов, кодирующих индексы $|00\rangle$, $|01\rangle$ и $|10\rangle$ гейт Тoffоли не будет реагировать, и вспомогательный кубит будет находиться в состоянии $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

Но при срабатывании гейта на индексе $|11\rangle$ 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 $|11\rangle$, так что можно считать, что вспомогательный кубит остался неизменным, и отнести минус к состоянию $|11\rangle$ верхних кубитов. Таким образом, индекс $|11\rangle$ помечен минусом как искомый. Другими словами, функция-оракул перевела состояние $|11\rangle |q_{\text{helper}}\rangle$ в состояние $|-11\rangle |q_{\text{helper}}\rangle$, где $|-11\rangle$ – 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 проявит себя при измерении. Да и сам индекс $|11\rangle$ 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 $|c\rangle$ – состояние системы перед первой итерацией – является линейной комбинацией векторов, соответствующим горизонтальной и вертикальной осям.

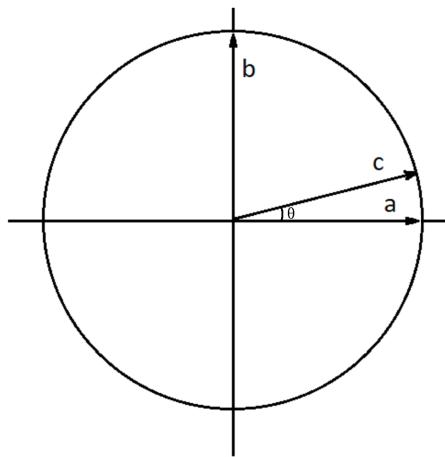


Fig. 45 Состояние системы перед первой итерацией.

Можно выразить вектор $|c\rangle$ для нашего случая (системы из двух кубитов с искомым индексом $|11\rangle$), обозначив его координаты за $|x\rangle$ и $|y\rangle$:

$$|\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = x \frac{1}{\sqrt{3}}(|00\rangle + |01\rangle + |10\rangle) + y |11\rangle$$

Из данного уравнения находим $|x = \frac{1}{\sqrt{3}}|2\rangle$ и $|y = \frac{1}{2}|2\rangle$.

По этим координатам можно понять, что угол между вектором $|c\rangle$ 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 :

$$|x = \cos\theta\rangle$$

$$|y = \sin\theta\rangle$$

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 метку.

После применения функции-оракула текущий вектор отразится относительно горизонтальной оси. Объясняется это очень легко – его вертикальная компонента (вектор $|11\rangle$) становится отрицательной.

Вектор $|c_{1b}\rangle$ – это отражение вектора $|c\rangle$ на угол θ вниз относительно горизонтальной оси:

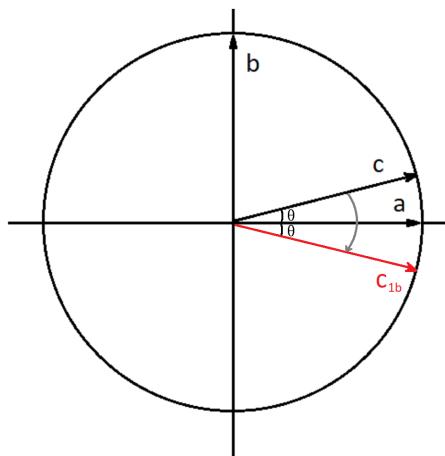


Fig. 46 Состояние системы после первой части первой итерации.

Такое отражение в нашем примере производится с помощью операции CNOT , но в общем случае операция выглядит так:

$$U_{1b} = I - 2|b\rangle\langle b|$$

Функцию-оракул мы здесь обозначили как $|U_{1b}\rangle$. Она меняет знак только для вертикальной составляющей вектора состояния, поэтому и происходит отражение.

Проверим формулу в действии, применив ее для нашего примера:

$$\begin{aligned} |U_{1b}|c\rangle = (I - 2|11\rangle\langle 11|) \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \frac{1}{2} \\ (|00\rangle + |01\rangle + |10\rangle + |11\rangle - 2|11\rangle) = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle) \end{aligned}$$

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 $|c\rangle$. Нетрудно заметить, что при этом текущий вектор состояния станет равен $(\cos\theta|a\rangle + \sin\theta|b\rangle)$.

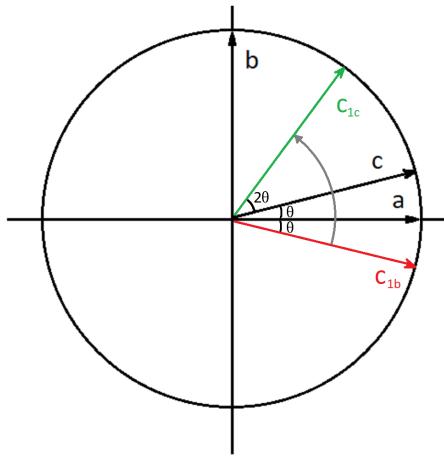


Fig. 47 Состояние системы после второй части первой итерации.

Операция для получения вектора $|c_{1c}\rangle$ будет выглядеть так:

$$U_{1c} = 2|c\rangle\langle c| - I$$

Посчитаем, чему равен вектор $|c_{1c}\rangle$ для нашего примера:

$$\begin{aligned} |U_{1c}|c_{1b}\rangle = (2|c\rangle\langle c| - I) \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle) = \\ |11\rangle \end{aligned}$$

Произошло отражение текущего вектора состояния $|c_{1b}\rangle$ относительно вектора $|c\rangle$. Если представить $|c_{1b}\rangle$ как $(k_1|c\rangle + k_2|b\rangle)$, где $|c\rangle$ – вектор, перпендикулярный $|c\rangle$, а k_1 и k_2 – действительные коэффициенты, то тогда отраженный вектор будет равен $(k_1|c\rangle - k_2|b\rangle)$.

В нашей квантовой схеме эта часть итерации реализована таким образом:

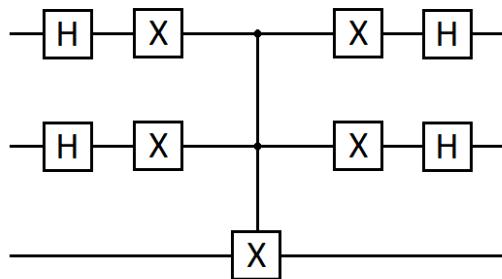


Fig. 48 Вторая часть итерации.

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 $\frac{\pi}{3}$, то есть это уже и есть желаемый угол $\frac{\pi}{2}$.

В общем случае этот угол равен $\sqrt{(2t+1)}\theta \approx \frac{\pi}{2}$, где t – номер произведенной итерации. Отсюда можно вывести число итераций, необходимое для работы алгоритма. При большом количестве итераций t угол θ будет стремиться к $\frac{\pi}{2}$, так что его можно заменить на $\sin(\theta)$, который, в свою очередь, равен $\frac{1}{\sqrt{N}}$:

$$\sqrt{(2t+1)}\theta \approx \frac{\pi}{2} \Rightarrow t \approx \frac{\pi^2}{4\sin^2(\theta)}$$

Если пренебречь единицей в скобках на основании того, что t – большое число, можно найти, что t приблизительно равно $\frac{\pi^2}{4\sin^2(\theta)}$.

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 $|c\rangle$. 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 $\frac{\pi}{4}\sqrt{N}$ итераций. Таким образом, для успешной работы алгоритма Гровера необходимо знать число K , чтобы можно было найти через него угол θ , а затем число итераций.

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 $|11\rangle$:

```
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 $\backslash(N)$ 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 $\backslash(11)$, 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.

Задание

1. Распишите операторы $\backslash(U_{\{1b\}})$ и $\backslash(U_{\{1c\}})$ из примера в виде матриц $\backslash(4)x\backslash(4)$ и проведите расчеты для получения $\backslash(c_{\{1b\}})$ и $\backslash(c_{\{1c\}})$ в виде векторов-столбцов.
2. Модифицируйте приведенный выше код алгоритма Гровера для двухкубитной базы данных так, чтобы искомый индекс соответствовал состоянию $\backslash|00\rangle$.

О блоке

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.

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](#).

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, analytic=False)

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([0.506, 0.    , 0.    , 0.494], 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])

result = circuit()
print(result)

[0.501 0.    0.    0.499]
```

В данном примере мы взяли двухкубитную систему и создали запутанное состояние, а затем с помощью метода `probs` вычислили вероятности получения состояний $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$.

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.Z`. 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 $|0\rangle$ ($|0\rangle$) 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., 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])
```

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, analytic=False)
```

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──────────| Sample[Z]
1: ───────────U0─| Sample[Z]
U0 =
[[1 0 0 0]
 [0 0 1 0]
 [0 1 0 0]
 [0 0 0 1]]
```

Cirq & TFQ

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](#).

Работа с кубитами

Импорты и схема

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](#):

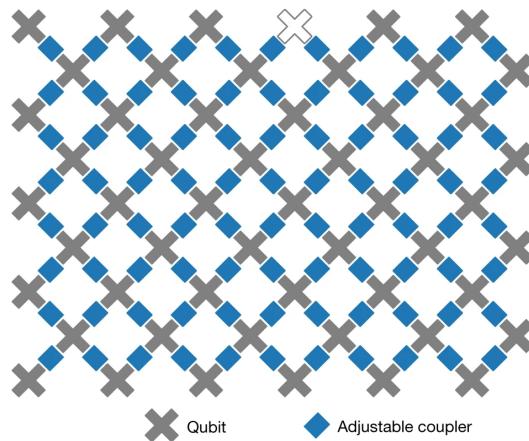


Fig. 49 Изображение из [AAB+19]

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 $(|0\rangle)$ и $(|1\rangle)$ и измеряет его:

```
circuit = cirq.Circuit()
circuit.append(cirq.H(qubit))
circuit.append(cirq.measure(qubit))
print(circuit)
```

```
(0, 0): —H—M—
```

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 $(|0\rangle)$, то есть если мы будем сэмплировать, то должны получать $(\text{sim} 0.5)$. Проверим это с разным числом сэмплов:

```
print("5 сэмплов:")
print(sim.sample(circuit, repetitions=5).mean())
print("\n100 сэмплов:")
print(sim.sample(circuit, repetitions=100).mean())
print("\n1000 сэмплов:")
print(sim.sample(circuit, repetitions=1000).mean())
```

```
5 сэмплов:
(0, 0)    0.4
dtype: float64

100 сэмплов:
(0, 0)    0.45
dtype: float64

1000 сэмплов:
(0, 0)    0.494
dtype: float64
```

Note

Метод `sim.sample` озвращает хорошо знакомый всем специалистам в области Data Science объект `pandas.DataFrame`. Для тех, кто слышит про такой впервые рекомендуем обратиться к вводным лекциям про `Python` и классическое машинное обучение.

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 `Goolge` или их новых квантовых симуляторах на TPU:

```
print(sim.run(circuit, repetitions=25))
```

```
(0, 0)=0001010110011110011101100
```

Тут мы просто видим последовательность наших измерений.

Квантовое машинное обучение

Импорты

Мы будем использовать `Tensorflow` и `Tensorflow Quantum`.

```
import tensorflow as tf
import tensorflow_quantum as tft
```

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)

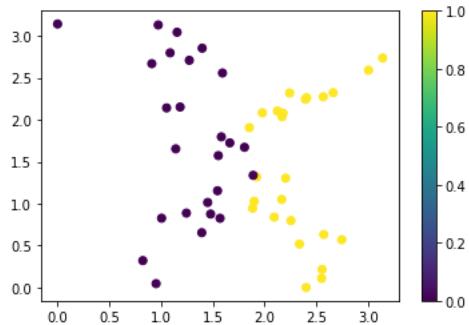
def normalize(x):
    x_min = x.min()
    x_max = x.max()

    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()
```



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)]
```

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])
```

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]))

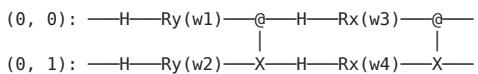
trainable_circuit.append(cirq.CNOT.on(qubits[0], qubits[1]))

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)

```



Наблюдаемые

В качестве операторов, которые мы будем измерять, воспользуемся парой $\langle \hat{X}Y \rangle$ и $\langle \hat{Y}X \rangle$ для наших кубитов:

```

ops = [cirq.X.on(qubits[0]) * cirq.Y.on(qubits[1]), cirq.Y.on(qubits[0]) *
       cirq.X.on(qubits[1])]

```

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)

```

- 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 #
=====                ======              =====
input_1 (InputLayer)   [(None, )]          0
pqc (PQC)             (None, 2)           4
dense (Dense)         (None, 2)           6
dense_1 (Dense)       (None, 1)           3
=====
Total params: 13
Trainable params: 13
Non-trainable params: 0

```

Предсказания со случайной инициализацией

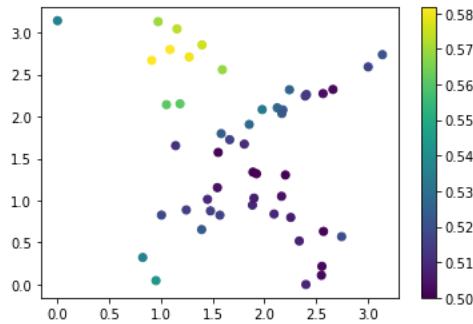
Наша нейросеть имеет случайные начальные параметры. Давайте посмотрим, что она предсказывает до обучения:

```

preds = model(x_input).numpy()

plt.figure(figsize=(6, 4))
cb = plt.scatter(x[:, 0], x[:, 1], c=preds)
plt.colorbar(cb)
plt.show()

```



Обучение сети

- запустим обучение

```

model.fit(x=x_input, y=y, epochs=200, verbose=0)

```

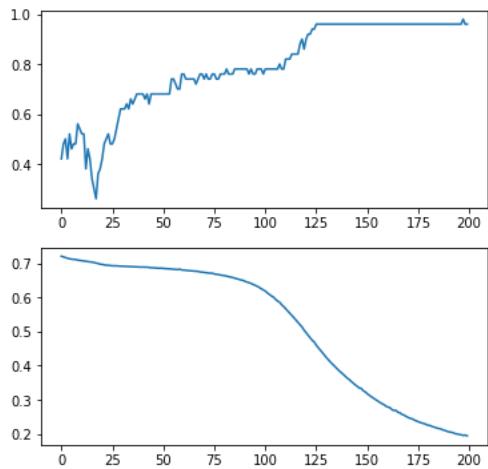
```
<tensorflow.python.keras.callbacks.History at 0x7f19d50e17c0>
```

- визуализируем логи обучения

```

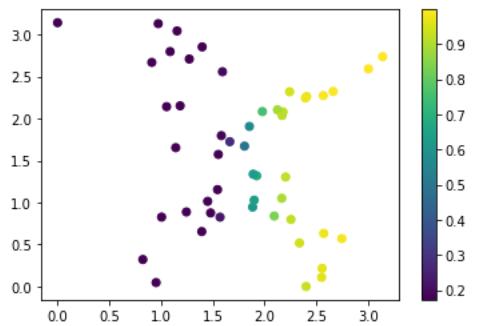
f, ax = plt.subplots(2, figsize=(6, 6))
ax[0].plot(model.history.history["binary_accuracy"])
ax[1].plot(model.history.history["binary_crossentropy"])
plt.show()

```



- визуализируем предсказания

```
preds_after_training = model(x_input).numpy()
plt.figure(figsize=(6, 4))
cb = plt.scatter(x[:, 0], x[:, 1], c=preds_after_training)
plt.colorbar(cb)
plt.show()
```



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](#).

О блоке

Этот блок включает в себя:

- вариационные квантовые схемы (VQC);
- квантовые градиенты.

Продвинутые темы блока дополнительно рассказывают:

- об универсальности VQC как аппроксиматоров;
- об интерпретации VQC как ядер.

Вариационные квантовые схемы

Описание лекции

На этой лекции мы впервые познакомимся непосредственно с квантовым машинным обучением. Теперь вместо [NumPy](#) мы будем использовать [PennyLane](#). 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.

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.

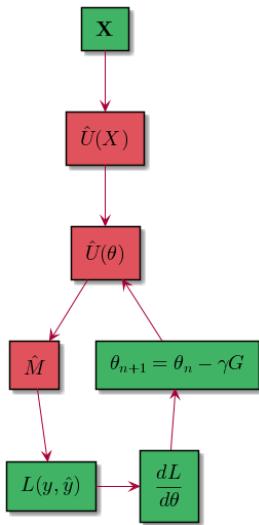


Fig. 50 Пример цикла смешанного обучения

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 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.

Квантово-классическое обучение

Основная идея квантово-классического обучения заключается в том, что в эпоху Noise 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.

Давайте более детально посмотрим на квантовую схему, которую можно использовать в таком подходе.

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 X и параметры θ в квантовое состояние $|\Psi(\theta)\rangle$.

\mathbf{X}). Затем будем варьировать наше состояние $|\Psi\rangle$, 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.

Аппроксимация матрицы Паули X оператором поворота

Из предыдущих лекций мы помним, что оператор $\hat{\mathbf{X}}$ работает как квантовый аналог оператора $=\text{NOT}$ – 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(simple_x_gate())
print(simple_x_gate.draw())
```

```
-1.0
0: —X— (Z)
```

Note

Тут мы измеряем наш кубит в \mathbf{Z} -базисе и чаще всего будем измерять именно в нем. \mathbf{Z} -базис является неким “стандартом” для квантовых вычислений, так как это измерение “ближе к железу”. Также \mathbf{Z} -базис удобен для нас из-за диагональности [оператора Паули](#) (σ^z) .

С другой стороны, у нас есть оператор $\hat{RX}(\phi)$, который “вращает” состояние нашего кубита вокруг оси \mathbf{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} :

```
print(f"Try 1.0: {vqc(0)}")
print(f"Try 2.0: {vqc(2)}")
print(f"Try 3.0: {vqc(3)}")
print(f"Try 3.14159265359: {vqc(3.14159265359)}")
```

```
Try 1.0: 1.0
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, но следующую лекцию мы полностью посвятим тому, 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.

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.

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:

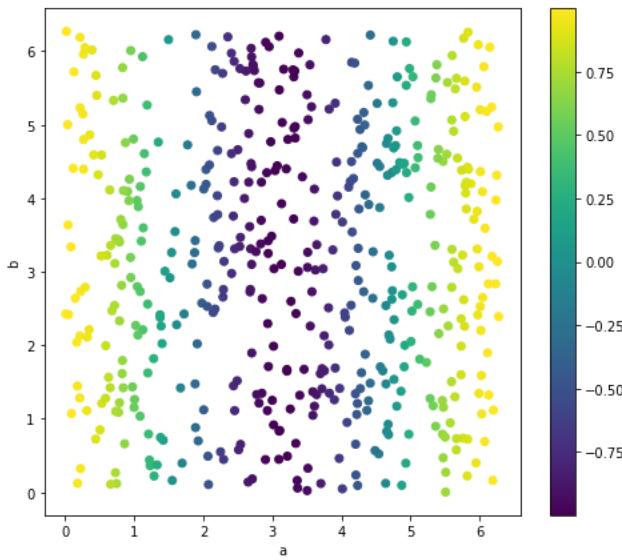
```
import numpy as np
import matplotlib.pyplot as plt

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 “поворнуть” 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
```

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)

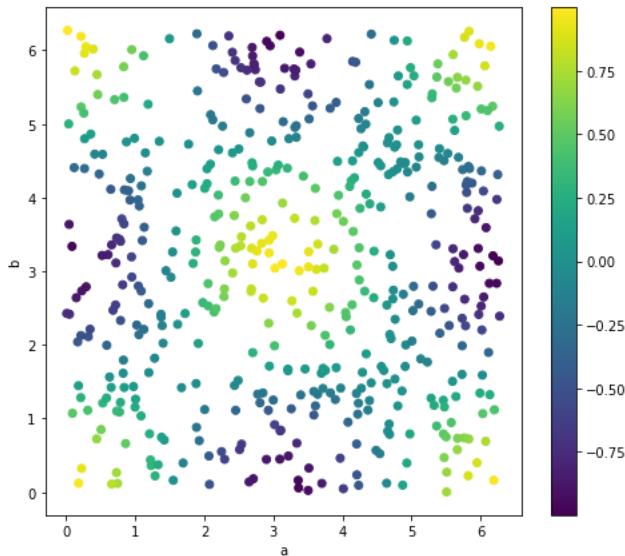
@qml.qnode(dev2)
def cnot_vqc(a, b):
    qml.RY(a, wires=0)
    qml.RY(b, wires=1)
    qml.RZ(a, wires=0)
    qml.RZ(b, wires=1)
    qml.CZ(wires=[0, 1])
    return qml.expval(qml.PauliZ(0) @ qml.PauliZ(1))
```

В этой схеме мы видим оператор \hat{CZ} , 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.

Что мы узнали из лекции

- В эпоху NISQ эпохи 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

О блоке

Этот блок включает в себя обзор способов оценки градиента VQC.

Продвинутые темы блока раскрывают следующие темы:

- квантовые градиенты старших порядков;
- квантовый натуральный градиент.

Градиенты квантовых схем

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

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?

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

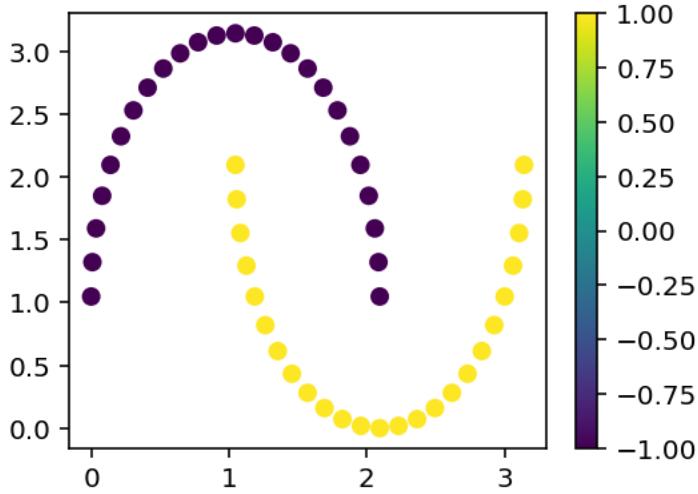
x, y = make_moons(n_samples=50)
```

Для удобства мы сразу переведем метки классов из $\{0, 1\}$ в $\{-1, 1\}$, а признаки \mathbf{X} переведем в $[0, 1]$:

```
def normalize(x):
    """
    Переводит значения в интервал от 0 до 1
    """
    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()
```



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 $\langle \mathbf{X} \rangle$ вращениями, а сверху применим несколько параметризованных слоев вращений.

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])
```

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 $\langle \hat{\text{Rot}}(\theta_1, \theta_2, \theta_3) \rangle$
- Вращение 2-го кубита $\langle \hat{\text{Rot}}(\theta_4, \theta_5, \theta_6) \rangle$
- “Запутывающий” оператор, который действует на оба кубита сразу – в нашем случае это $\langle \hat{\text{CZ}} \rangle$

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 $\langle \hat{\text{CZ}} \rangle$.

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))
```

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

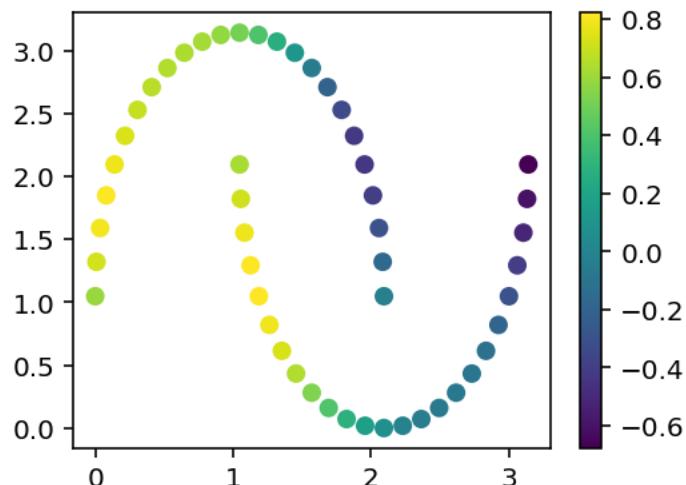
Именно это и реализовано в коде.

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.

Функция потерь

Прежде чем варьировать 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.

Квадратичное отклонение

В качестве функции потерь, которая является дифференцируемой, мы будем использовать наиболее простой вариант – среднеквадратичное отклонение. 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))
```

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]
```

Решение средствами PennyLane

Библиотека [PennyLane](#) может использовать один из нескольких движков для автоматического дифференцирования:

- [NumPy Autograd](#)
- [PyTorch](#)
- [Tensorflow](#)
- [Jax](#)

По больше 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))

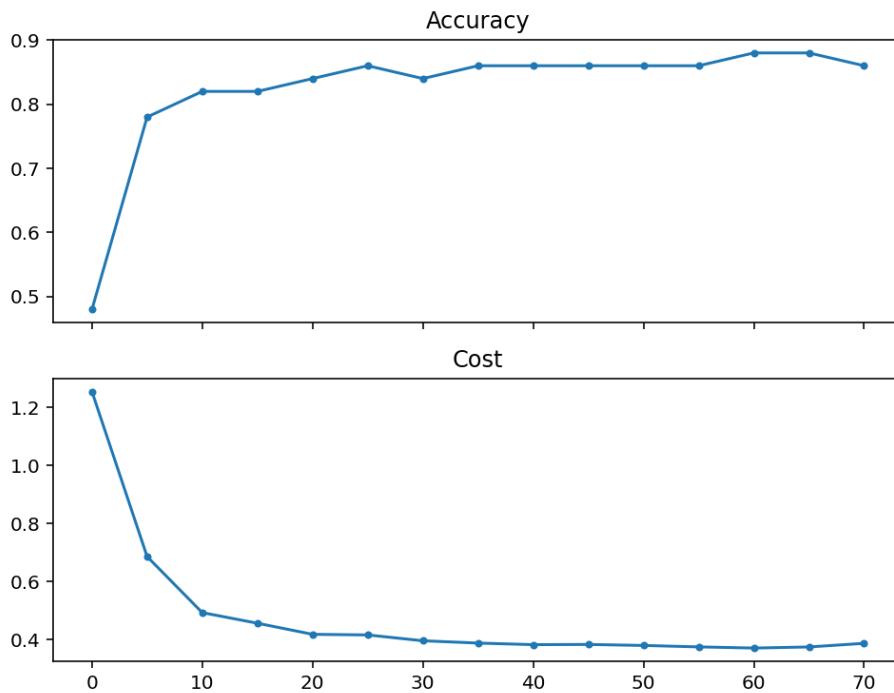
```

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()

```

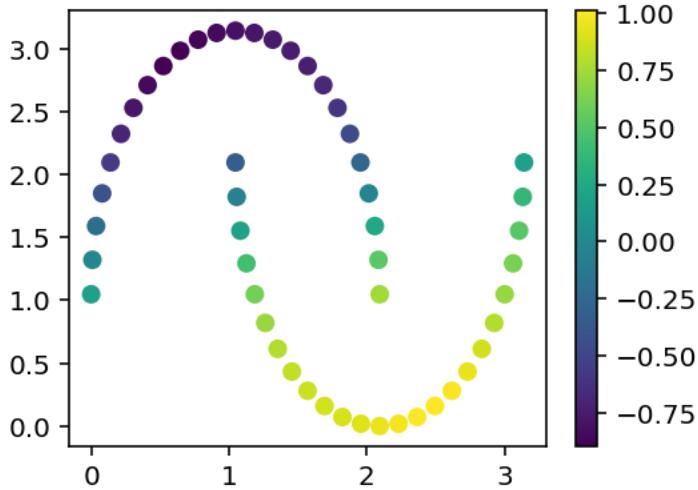


А также на результаты классификации:

```

plt.figure(figsize=(4, 3))
cb = plt.scatter(x[:, 0], x[:, 1], c=apply_node(x, q))
plt.colorbar(cb)
plt.show()

```



А как оно работает?

Теперь, когда мы увидели процесс оптимизации квантовой схемы, давайте попробуем подумать, а как оно на самом деле работает?

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:

$$\frac{df}{dx} \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

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 $\|\Delta\|$ 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.

Parameter-shift rule

Более точная оценка может быть получена методом, который называется **Parameter shift**. Он основан на том, что для квантового “черного ящика” $\langle \hat{U}(\theta) \rangle$, которым является наша схема, частная производная по параметру θ_i выражается так:

$$\langle \nabla_{\theta_i} \hat{U} \rangle = \frac{1}{2} (\langle \hat{U}(\theta + \pi/2) \rangle - \langle \hat{U}(\theta - \pi/2) \rangle)$$

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 θ_i -му параметру может быть получена вычислением сначала ожидаемого значения схемы с параметром θ_i , смещенным на $\pi/2$ 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):

$$\frac{\partial L}{\partial \theta_i} = \frac{1}{2} (\langle \hat{U}(\theta + \pi/2) \rangle - \langle \hat{U}(\theta - \pi/2) \rangle)$$

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.

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

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](#)].

Важность гейтов вращений

Если задуматься, то одним из основных (если не единственных) способов сделать параметризованную квантовую схему является использование гейтов вращений, таких как $\{\hat{RX}, \hat{RY}, \hat{RZ}\}$. 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 $(\sigma_x, \sigma_y, \sigma_z)$.

Если представить схему, содержащую множество параметризованных операторов, то итоговая запись имеет вид:

$$U_{j...k} = U_j, \dots, U_k | \Psi \rangle$$

Производная от измерения

Давайте вспомним, как выглядит квантово-классическая схема обучения с **VQC**.

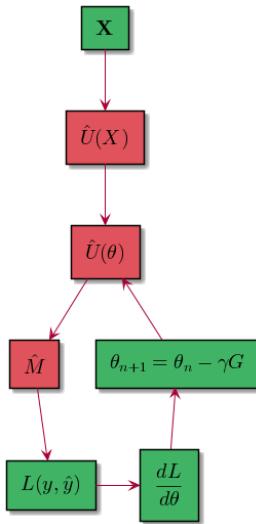


Fig. 51 Квантово-классическая схема

Видно, что мы хотим считать производную не от самой параметризованной схемы $(U_{j\dots k})$, а от наблюдаемой. Для тех, кто забыл, что такое наблюдаемая, 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:

$$\langle G_i = \frac{\partial}{\partial \theta_i} \langle \Psi | \hat{M} | \Psi \rangle \rangle$$

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 $(\theta_j, \dots, \theta_i, \dots, \theta_k)$.

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 $(|\Psi_i\rangle)$, которое пришло на вход (i) -го гейта в нашей последовательности:

$$\langle M(\theta_i) \rangle = \text{Tr}(M U_{k\dots 1} |\Psi_i\rangle \langle \Psi_i|)$$

где (ρ_i) это матрица плотности $(|\Psi_i\rangle \langle \Psi_i|)$. 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_i) , которое “пришло” на вход гейта (U_i) и того оператора Паули (P_i) , который мы используем в (U_i) :

$$\frac{\partial}{\partial \theta_i} \langle M \rangle = -\frac{1}{2} \text{Tr}(M U_{k\dots i} P_i U_{i-1\dots 1} \rho_i U_{i-1\dots 1}^\dagger U_{k\dots i}^\dagger)$$

Этот коммутатор может быть переписан следующим образом:

$$\langle [P_i, \rho_i] \rangle = i[U_i \text{Bigl}(\frac{\partial}{\partial \theta_i} \text{Bigr}) \rho_i U_i^\dagger - U_i \text{Bigl}(\frac{\partial}{\partial \theta_i} \text{Bigr}) \rho_i U_i^\dagger]$$

Тогда соответствующий градиент $(\frac{\partial}{\partial \theta_i} \langle M \rangle)$ можно записать через смещения на $(\pm \frac{\partial}{\partial \theta_i})$:

$$\begin{aligned} \langle \frac{\partial}{\partial \theta_i} \langle M \rangle \rangle &= \frac{1}{2} \text{Tr}[M U_{k\dots i+1} U_i (\pm \frac{\partial}{\partial \theta_i}) \rho_i U_i^\dagger U_{i-1\dots 1}^\dagger] \\ &\quad + \frac{1}{2} \text{Tr}[M U_{k\dots i+1} U_i^\dagger (\pm \frac{\partial}{\partial \theta_i}) \rho_i U_i U_{i-1\dots 1}^\dagger] \end{aligned}$$

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 $(\pm \frac{\partial}{\partial \theta_i})$ и *backward* со смещением на $(-\frac{\partial}{\partial \theta_i})$.

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 $\langle \hat{M} \rangle$ может всегда быть представлено [MBK21] как сумма вида:

$$\langle \hat{U}_i(\theta_i) \hat{M} \hat{U}_i^\dagger(\theta_i) \rangle = \hat{A} + \hat{B} \cos(\theta_i) + \hat{C} \sin(\theta_i)$$
 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 $U(\theta) = e^{-\frac{i}{2}\theta \hat{H}}$, получаем:

$$\begin{aligned} & \langle \cos(\frac{\theta}{2}) \hat{M} \cos(\frac{\theta}{2}) \rangle = \cos^2(\frac{\theta}{2}) \hat{M} + i \sin(\frac{\theta}{2}) \cos(\frac{\theta}{2}) \hat{M} - i \sin(\frac{\theta}{2}) \cos(\frac{\theta}{2}) \hat{M} + i \sin^2(\frac{\theta}{2}) \hat{M} = \\ & \frac{1}{2} \left(\hat{M} + i \sin(\theta) \hat{M} + i \sin(\theta) \hat{M} - i \sin(\theta) \hat{M} \right) = \frac{1}{2} \left(\hat{M} + i \sin(\theta) \hat{M} - i \sin(\theta) \hat{M} \right) = \frac{1}{2} \hat{M} \end{aligned}$$

Тогда можно воспользоваться правилами тригонометрии, а именно, тем что для любого $s \neq k\pi$, $\text{atan}(s) \in \{1, 2, \dots, N\}$ справедливо:

$$\frac{1}{2} \left(\sin(\theta + s) - \sin(\theta - s) \right) = \frac{1}{2} \sin(s) \cos(\theta) + \frac{1}{2} \cos(s) \sin(\theta)$$

И подставим это в выражение для $\langle \partial_{\theta_i} \hat{M} \rangle$:

$$\frac{1}{2} \left(\sin(\theta_i + s) - \sin(\theta_i - s) \right) = \frac{1}{2} \sin(s) \cos(\theta_i) + \frac{1}{2} \cos(s) \sin(\theta_i)$$

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:

$$\boxed{\frac{\partial f(\hat{M}(\theta))}{\partial \theta_i} = \frac{f(\hat{M}(\theta + s) \mathbf{e}_i) - f(\hat{M}(\theta - s) \mathbf{e}_i)}{2s}}$$

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 $O(N^2)$, 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]:

$$\begin{aligned} \frac{\partial^2 f}{\partial \theta_i^2} = & \frac{f(\hat{M}(\theta + s_1) \mathbf{e}_i + s_2 \mathbf{e}_j) - f(\hat{M}(\theta - s_1) \mathbf{e}_i - s_2 \mathbf{e}_j) + \\ & f(\hat{M}(\theta + s_1) \mathbf{e}_i - s_2 \mathbf{e}_j) - f(\hat{M}(\theta - s_1) \mathbf{e}_i + s_2 \mathbf{e}_j)}{4s_1 s_2} \end{aligned}$$

Взяв $s_1 = s_2$, можно упростить это выражение к следующему виду:

```
\begin{split}
\begin{gathered}
\frac{f(\theta) + s\mathbf{a}) + f(\theta) + s\mathbf{b}) - f(\theta) + s\mathbf{c}) - f(\theta) + s\mathbf{d})}{(2\sin(s))^2} \\
\mathbf{a} = \mathbf{e}_i + \mathbf{e}_j \quad \mathbf{b} = -\mathbf{e}_i - \mathbf{e}_j \quad \mathbf{c} = \mathbf{e}_i - \mathbf{e}_j \quad \mathbf{d} = -\mathbf{e}_i + \mathbf{e}_j
\end{gathered}
\end{split}
```

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 $\langle s_g \rangle$ при вычислении вектора градиента, а также такое значение $\langle s_h \rangle$ 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:

$$\frac{f(\theta) + 2s\mathbf{e}_i) + f(\theta) - 2s\mathbf{e}_i)}{(2\sin(s))^2}$$

Можно заметить, что, например, использование $\langle s = \frac{\pi}{4} \rangle$ для гессиана, а также "стандартного" $\langle s = \frac{\pi}{2} \rangle$ для градиента позволит полностью переиспользовать в диагональных элементах гессиана значения, которые мы получили при расчете градиента. А значение $\langle f(\theta) \rangle$ 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 [\[And19\]](#).

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.

О блоке

Этот блок включает в себя:

- квантово-классический SVM;
- полностью квантовый SVM.

Продвинутые темы блока дополнительно рассказывают:

- продвинутую математику SVM;
- интеграцию с алгоритмом Гровера;
- варианты применения HHL-алгоритма.

Квантово-классический SVM

Описание лекции

Лекция будет построена следующим образом:

- Вспомним, что такое классический SVM
- Поговорим о классическом *kernel trick*
- Посмотрим, как можно использовать **VQC** как ядра SVM
- Напишем и применим код обучения смешанного SVM

Классический SVM

В данной лекции мы будем много говорить об **SVM** (*Support Vector Machine*) – алгоритме классического машинного обучения, в основе которого лежит построение оптимальной разделяющей гиперплоскости. Для детального понимания работы этого алгоритма настоятельно рекомендуется вернуться к вводной лекции про SVM.

Давайте кратко вспомним, как устроен этот алгоритм.

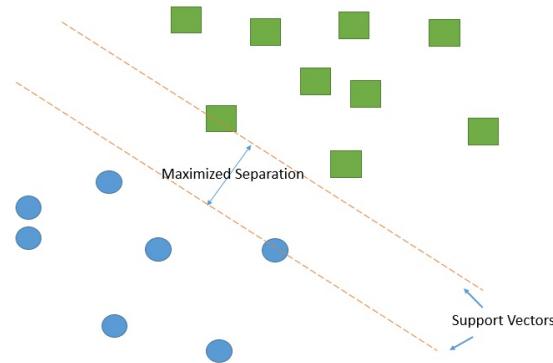


Fig. 52 Классический SVM

В данном случае алгоритм пытается найти такую разделяющую прямую (в многомерном случае это будет гиперплоскость), чтобы расстояния от этой прямой до точек разных классов были максимальными.

Поскольку обычно нам интересны многомерные пространства, то точки там превращаются в вектора. Также оказывается, что построение такой разделяющей плоскости зависит не от всех точек (векторов), а только от какого-то их подмножества – опорных векторов. Именно поэтому метод и носит название *Support Vector Machine*.

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 один из ключевых теоретических концептов машинного обучения – размерность Вапнича-Червоненкиса.

Сильные стороны SVM

У этого алгоритма есть несколько очень сильных сторон, если сравнивать его, например, с алгоритмом логистической регрессии:

- В реальности нам интересны не все точки, а лишь те, которые лежат вблизи разделяющей гиперплоскости
- Задача поиска такой прямой может быть сформулирована как задача квадратичного программирования
- Решение задачи квадратичного программирования может быть получено аналитически
- Решение может быть сформулировано с использованием лишь скалярных произведений векторов

Kernel-trick

Наиболее интересным для нас будет последнее из списка. Ведь в данном случае мы можем искать оптимальные разделяющие гиперплоскости даже в пространстве бесконечной размерности – главное, чтобы в этом пространстве было определено скалярное произведение.

Это используется в расширении SVM, которое называется **ядерный** SVM. В данном случае мы используем **ядро** для вычисления скалярного произведения и строим разделяющую гиперплоскость не в исходном пространстве, где данные, вообще говоря, могут быть неразделимыми, а в новом пространстве. Для этого нам необходимо лишь иметь выражение для скалярного произведения, которое и называется **ядром**. Хорошие примеры ядер для SVM:

- Полиномиальное ядро
- Радиально-базисная функция (*Radial basis function*)

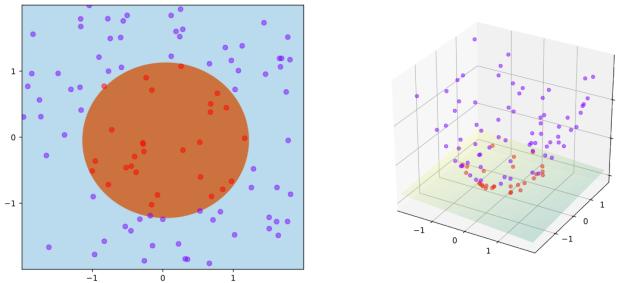


Fig. 53 Пример разделимости в новом пространстве

Давайте посмотрим на то, как выглядят популярные ядра.

Полиномиальное ядро

Для степени ядра $\langle d \rangle$ и параметра нормализации $\langle c \rangle$ скалярное произведение двух векторов $\langle x, y \rangle$ определяется так:

$$\langle K(x, y) = (x^T y + c)^d \rangle$$

RBF

Для параметра ядра $\langle \sigma \rangle$ формула для скалярного произведения такая:
 $\langle K(x, y) = e^{-\frac{\|x - y\|^2}{\sigma^2}} \rangle$

Проблемы с ядрами

Несмотря на огромный потенциал, у ядерного SVM есть большая проблема – масштабируемость. Вычислять ядра иногда может быть проблематично, из-за этого на действительно больших данных на первое место в последние годы вышли нейронные сети.

А теперь давайте посмотрим, что квантовые компьютеры могут дать классическому SVM!

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. Этой теме у нас даже посвящена одна из более ранних лекций продвинутого уровня, где разбираются доводы из статьи [Sch21].

Очевидная идея – попробовать как-то воспользоваться квантовой схемой, чтобы реализовать скалярное произведение двух классических векторов. Именно это и сделали авторы работы [HaylivcekCorcolesT+19].

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 $\backslash(CNOT)$
- 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 $\backslash(CNOT) \rightarrow$ параметризованная парой операция $\rightarrow \backslash(CNOT)$.

Выбор операции

Следуя идеи упомянутой статьи, в качестве что одно-элементной, что двух-элементной операции мы будем использовать гейт $\backslash(U_1)$. Разница будет лишь в том, что мы передаем на вход в качестве параметра.

Feature function

В качестве параметров на входе гейта $\backslash(U_1)$, как мы уже говорили, выступают один или два элемента вектора $\backslash(x)$. Строго это можно записать как функцию такого вида:

$$\begin{aligned} \backslash\begin{cases} \text{split} \\ \phi(x_1, x_2) = \begin{cases} \phi(x), & x_1 = x_2 \\ \phi(x_1, x_2), & x_1 \neq x_2 \end{cases} \end{cases} \end{aligned}$$

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:

$$\begin{aligned} \backslash\begin{cases} \text{split} \\ \text{gathered} \\ \phi(x) = x \cdot \phi(x_1, x_2) = \pi \cos(x_1) \cos(x_2) \end{cases} \end{aligned}$$

Скалярное произведение

Все что мы описали выше, обозначим как квантовую схему $\backslash(U(x))$. Она преобразует нам вектор классических данных $\backslash(x)$ в квантовое состояние $\backslash(\text{ket}\{\Psi\})$. Но нам то нужно получить скалярное произведение $\backslash(\text{braket}\{U(x_1)|U(x_2)\})$! Выглядит сложно, но на самом деле существует эффективный способ получить эту величину без необходимости восстанавливать весь вектор состояния. Можно показать, что величина $\backslash(\text{braket}\{U(x_1)|U(x_2)\})$ равна вероятности нулевой битовой строки ($\backslash(\text{ket}\{0, 0, \dots, 0\})$) при измерении другой схемы: $\backslash(U(x_1)U(x_2)^{\dagger})$.

Все это может казаться сложным и запутанным, но должно стать гораздо понятнее, когда мы посмотрим на пример реализации от начала и до конца.

Пример реализации

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
```

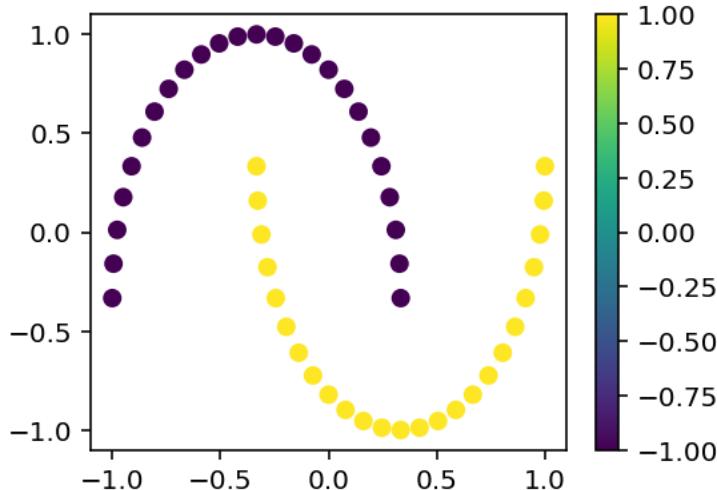
Мы будем работать с уже привычным нам набором “Two Moons”. Только в этом случае мы будем использовать чуть-чуть другую нормализацию – для нашего ядра элементы вектора x должны быть в интервале $\{[-1, 1]\}$. Сразу переведем наши данные в этот диапазон:

```
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 ($|U(x)\rangle$). 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 $(\langle U(x_1)|U(x_2)\rangle) = \langle U(x_1)U(x_2)^\dagger|\Psi\rangle M_0 |\langle U(x_1)U(x_2)^\dagger|\Psi\rangle)$. Тут M_0 – это проектор на один из собственных векторов системы кубитов, а именно на “нулевой”: $M_0 = |\langle 0, \dots, 0|\rangle \langle 0, \dots, 0|$). Проще говоря, мы реализуем схему, которая нам дает вероятности каждой из битовых

строк (а дальше мы просто возьмем первую, она и отвечает строке $\langle 0, \dots, 0 \rangle$):

```
@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:351: 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(0.506)—H—rC—————rC—rC—————rC—H—Rφ(0.862)—H—r|
Probs
1: —H—Rφ(-0.955)—H—lX—Rφ(1.59)—lX—lX—Rφ(-1.81)—lX—H—Rφ(-0.478)—H—l|
Probs
```

Гибридный SVM

Мы не будем сами с нуля писать решение задачи квадратичного программирования. Мы 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 $\langle U(x) | U(x) \rangle = 1$ и $\langle U(x_1) | U(x_2) \rangle = \langle U(x_2) | U(x_1) \rangle$:

```
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')
```

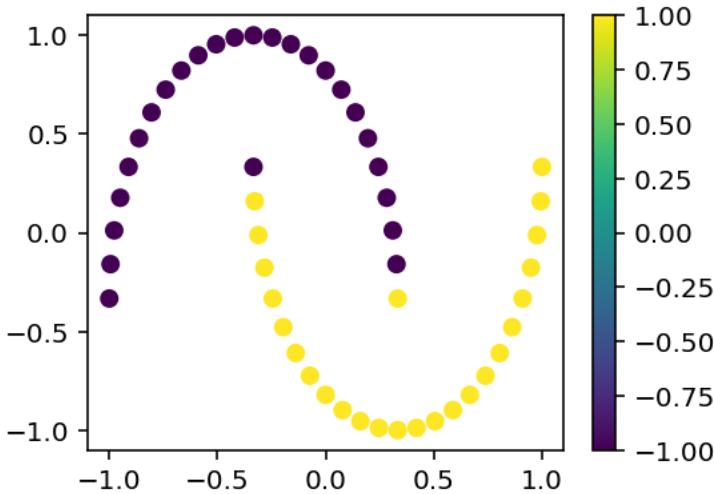
Посчитаем предсказания и посмотрим на результат:

```

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!

Заключение

Расчет полной матрицы скалярных произведений дает нам сложность $\mathcal{O}(N^2)$ 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 факультативных лекциях), причем за сложность всего $\mathcal{O}(N)$! и без расчета полной матрицы Грамма!

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!

Квантовая химия. Введение.

Описание лекции

Из этой лекции мы 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

Введение

В предыдущих лекциях мы 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, такие как 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. Однако 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. Для желающих в конце приведены ссылки для углубленного изучения.

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.

АТОМ ВОДОРОДА

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:

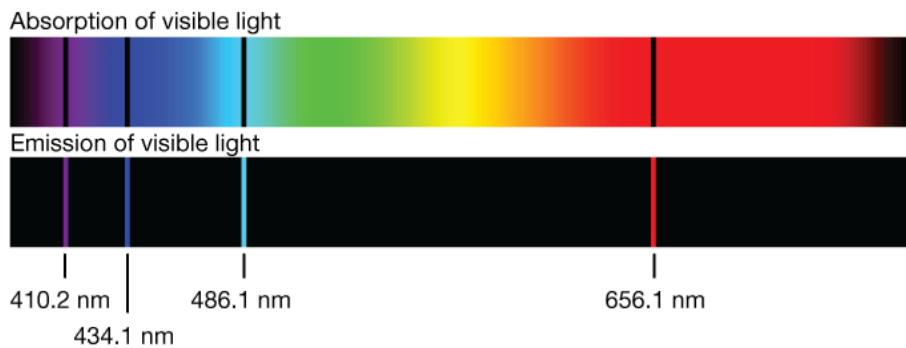


Fig. 54 Спектр поглощения и излучения водорода в видимом диапазоне

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.

Note

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.

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 импульс может принимать только определенные дискретные значения (кванты), а не произвольные:

$$(p \cdot 2\pi r = n \cdot h)$$

$$(p = \frac{n \cdot h}{r})$$

где p – импульс, r – радиус орбиты электрона, \hbar – константа 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”):

$$\langle E_{\text{kin}} \rangle = -E_{\text{pot}} / 2$$

$$\langle E \rangle = E_{\text{kin}} + E_{\text{pot}} = E_{\text{pot}} / 2$$

Записываем кинетическую энергию и потенциал Кулона для электрона в поле протона (в СИ):

$$\langle \frac{p^2}{2m} \rangle = k \frac{e^2}{2R}, \text{ где } k \text{ – постоянная из закона Кулона}$$

Используем постулат Бора и выразим “радиус” орбиты:

$$\langle \frac{n^2 \hbar^2}{2m R^2} \rangle = k \frac{e^2}{2R}$$

$$R = \frac{n^2 \hbar^2}{k m e^2}$$

Находим полную энергию:

$$\langle E \rangle = E_{\text{kin}} + E_{\text{pot}} = -\frac{k e^2}{2R} = -\frac{k^2 n^2 \hbar^2}{2m e^2}$$

Уровни энергии пропорциональны $\langle \frac{1}{n^2} \rangle$ и именно [такая зависимость](#) наблюдается в эксперименте.

Посчитаем минимальную энергию ($n = 1$):

```
from scipy import constants as consts
k = (1 / (4 * consts.pi * consts.epsilon_0)) # постоянная Кулона
E1 = -k**2 * consts.m_e * consts.e**4 / (2 * consts.hbar ** 2) # энергия в Джоулях
E1_ev = E1 / consts.e # энергия в электрон-вольтах
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 импульс квантуется, то есть принимает только дискретные значения), удалось разрешить фундаментальную нестыковку между классической теорией и экспериментом: энергия стала дискретной, линии спектра стали обратно пропорциональны $\langle n \rangle$, 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 их формулы в классическую физику, мы 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.

Описание лекции

Из этой лекции мы 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.

Введение

Боровская теория оказалась недостаточной для описания таких сложных систем, как молекулы или нейтронодоподобные атомы. В этой лекции мы 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.

Что мы ищем?

С точки зрения квантовой химии, чаще всего мы 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 мы можем подействовать на волновую функцию и получить измеренное значение этой величины.

Например, если мы знаем волновые функции электронов и ядер в кристалле – 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.

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:

$$\left(\dot{\psi} - \frac{1}{2m} \nabla^2 \psi + V(r) \psi \right) = E \psi$$

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 , где E – это энергия частицы.

С точки зрения математики искомая волновая функция является собственным вектором оператора эволюции, а энергия – собственным значением.

С точки зрения объяснения “на пальцах” оператор эволюции при применении к волновой функции должен вернуть нам новую (эволюционированную/изменяющуюся во времени) волновую функцию. Если мы ищем стационарную – не меняющуюся во времени – волновую функцию, то при применении к ней оператора эволюции она не должна изменяться, иначе будет уже не стационарной.

$$\left(\dot{\psi} - \frac{1}{2m} \nabla^2 \psi + V(r) \psi \right) \equiv E \psi$$

Оператор импульса раскрывается через градиент так: $\hat{p} = \dot{\psi}$

Потенциал взаимодействия двух частиц с противоположными единичными зарядами раскрывается так: $V(r) = -\frac{e^2}{r}$

Итого имеем:

$$(-\frac{\hbar^2}{2m}\nabla^2 - \frac{e^2}{r})|\Psi(r)\rangle = E |\Psi(r)\rangle$$

Прежде чем погружаться глубже в решение этого уравнения, нам надо разобраться с квантовыми числами.

Про квантовые числа

Теория Бора базируется на идее квантования импульса, и ее следствием является возникшее в формуле энергии число $\langle n \rangle$. Каждое число $\langle n \rangle$ соответствует определенному состоянию, в котором может находиться электрон и эти состояния отличаются энергией.

Идея о том, 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 $\langle n \rangle$ называется главным [квантовым числом](#), оно определяет энергетический уровень электрона.

Из анализа свойств операторов гамильтониана и момента импульса в квантовой механике выводятся еще три квантовых числа:

- $\langle l \rangle$ – орбитальное квантовое число, $0 \leq l \leq n-1$;
- $\langle m \rangle$ – магнитное квантовое число, $-l \leq m \leq l$;
- $\langle s \rangle$ – 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 $|\Psi\rangle$, для которой уравнение выше верно, описывает электрон, который стабильно находится где-то около ядра и не покидает его, то есть образует с ним атом. При этом функций-решений уравнения на самом деле много, что физически соответствует тому, что электрон может находиться на разных орбиталах. Каждая орбиталь характеризуется квантовыми числами – $\langle n \rangle$, $\langle l \rangle$, $\langle m \rangle$ и $\langle s \rangle$, и обозначается как $|\Psi_{n,l,m,s}\rangle$. $|\Psi_{1,0,0,0}\rangle$ (орбиталь) с минимальной энергией E соответствует основному состоянию (ground state) – она описывает невозбужденный электрон. У водорода только один электрон, поэтому единственная $|\Psi_{1,0,0,0}\rangle$ с минимальной энергией соответствует невозбужденному атому водорода.

В целом основная задача квантовой химии – найти ground state произвольной системы частиц, так как это описывает “обычное” состояние системы, которое встречается чаще всего в реальности. Но если электрону в основном состоянии придать энергию, 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:

$|\Psi_{n,l,m,s}\rangle$ – 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 и тем же n энергия одинакова.

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.

Физически электрон водорода обычно в $|\Psi_{1,0,0,0}\rangle$, так как такое распределение соответствует минимальной энергии.

Долгое и правильное решение, часть 2

Note

А зачем нам вообще сдался спектр и энергии? Вслед за физиками двадцатого века нам приходится продираться 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 крутою 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 вычислительные устройства) работают на основе туннельного эффекта – квантового явления. Так что можно сказать, что квантовая механика используется сплошь и рядом!

Если мы перейдем в уравнении Х в сферические координаты со следующей параметризацией $\langle \Psi | \vec{r} \rangle = \chi(r) Y(\theta, \phi)$ и воспользуемся несколькими волшебными выводами квантмеха [89], то получим:

$$-\frac{\hbar^2}{2m} \frac{d^2 \chi}{dr^2} + \left(-\frac{e^2}{r} + \frac{\hbar^2 \ell(\ell+1)}{2mr^2} - E \right) \chi(r) = 0$$

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).

$$-\frac{\chi''(r)}{2} + \left(\ell(\ell+1) \frac{2r^2}{r} - \frac{1}{\chi(r)} \right) \chi(r) = 0$$

Опустим несколько страниц выкладок [15], учтем граничные условия и получим следующее решение:

$$R_{n\ell}(r) = r^{\ell} e^{-r/n} \sum_{k=0}^{n-\ell-1} \frac{(-2r/n)^k}{(2\ell+2+k)! (n-\ell-k-1)! k!} C$$

$$\langle \Psi | \vec{r} \rangle = R_{n\ell}(r) Y_{\ell m}(\theta, \phi)$$

где $Y_{\ell m}(\theta, \phi)$ – [сферические функции](#).

Если подставить это решение в уравнение Шредингера и найти энергию, то получим:

$$E_n = -\frac{1}{2m} \frac{me^4}{2\hbar^2} \frac{1}{C}$$

То есть получим ту же формулу, что и в теории Бора, и тот же численный результат – 13.6 eV.

Здесь начинает пропасть 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.

От теории к практике

Вернемся из начала двадцатого века обратно в день сегодняшний. Сто лет назад неберущиеся 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 комбайну-вычислителю и забрать ответ, но без понимания о происходящем под капотом есть шансы получить что-то неправильное.

В этой лекции мы 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
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]

e_in_ht = psi4.energy('scf') # энергия в единицах Hartree
print(f"Hydrogen ground state energy: {e_in_ev(e_in_ht)} eV")
```

Здесь мы 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 мы помним, равен 13.6 eV) и мы его улучшим после того как разберемся с тем, что и как мы только что посчитали. Разобраться будет удобнее на примере атома гелия, потому что в атоме водорода есть только один электрон, а в любой реальной системе – больше одного.

Теория самосогласованного поля

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:

$$\dot{\bar{\Psi}}(t, \vec{r}_A, \vec{r}_B) = \left(-\frac{1}{2m} \nabla_{\vec{r}_A}^2 - \frac{e^2}{4\pi\epsilon_0 r_A} + \frac{1}{2m} \nabla_{\vec{r}_B}^2 - \frac{e^2}{4\pi\epsilon_0 r_B} + \frac{e^2}{8\pi\epsilon_0 r_{AB}} \right) \bar{\Psi}(t, \vec{r}_A, \vec{r}_B)$$

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 A 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).

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 снippetе выше строка `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 A 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 A:

$$(\langle h_A + \hat{V}_{\text{eff}} \rangle | \psi_A \rangle = (h_A + \langle \psi_B | \frac{e^2}{r_{AB}} | \psi_B \rangle) | \psi_A \rangle = \epsilon_A | \psi_A \rangle$$

Здесь $\langle h_A \rangle$ – это кинетическая энергия и потенциал ядра для электрона A, а $\langle V_{\text{eff}} \rangle$ – влияние электрона B на электрон A. $\langle V_{\text{eff}} \rangle$ вычисляется как влияние усредненной электронной плотности, распределенной в соответствии с $|\psi_B|$.

На этом шаге мы считаем, что $|\psi_B\rangle$ – фиксированная волновая функция и мы находим “переменную” $|\psi_A\rangle$. Однако $|\psi_A\rangle$ тоже будет влиять на $|\psi_B\rangle$ и, записав аналогичное уравнение для частицы B, мы следующим шагом найдем новую $|\psi_B\rangle$.

После изменения волновых функций каждого электрона мы 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 $|\Psi(r_A, r_B)\rangle$, самый простой вариант – это $|\Psi(r_A, r_B)\rangle = |\psi_A(r_A)\psi_B(r_B)\rangle$, он и был предложен первоначально. Однако есть проблема: такая $|\Psi\rangle$ 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:

$$\langle |\Psi(X_A, X_B)|^2 = |\Psi(X_B, X_A)|^2 \rangle$$

Кроме того, представим что мы обменяли частицы дважды: А с В 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 мы могли получить $\langle|\Psi'|^2 = e^{i\theta} |\Psi|^2 \rangle$.

Но, если мы работаем больше чем в двух пространственных измерениях, то такой двойной обмен эквивалентен отсутствию обмена. Для начала представим что мы обмениваем частицы медленно описывая полукруг частицей А вокруг частицы В и потом сдвигая обе частицы. Тогда двойной обмен значит, что частица А описывает полный круг вокруг частицы В. Также, если мы непрерывно изменим ее маршрут не приближая ее к частице В, то мы ожидаем получить тот же результат.

Итак, какие замкнутые маршруты мы можем получить непрерывно деформируя маршрут в евклидовом пространстве без точки (В)? Ответ для любого измерения больше 2 – [любые](#). Следовательно двойной обмен должен давать такой же результат как и если бы мы просто оставили частицу А на месте [\[NSS+08\]](#).

У этого ограничения есть два решения: либо $\langle|\Psi(X_A, X_B)|^2 = |\Psi(X_B, X_A)|^2 \rangle$ (симметричность), либо $\langle|\Psi(X_A, X_B)|^2 = -|\Psi(X_B, X_A)|^2 \rangle$ (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 приём:

$$|\Psi(X_A, X_B)|^2 = |\psi_A(X_A)|^2 |\psi_B(X_B)|^2 - |\psi_A(X_B)|^2 |\psi_B(X_A)|^2$$

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: $\langle|\Psi(X_A, X_B)|^2 = \begin{vmatrix} |\psi_A(X_A)|^2 & |\psi_B(X_A)|^2 \\ |\psi_A(X_B)|^2 & |\psi_B(X_B)|^2 \end{vmatrix}$

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.

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 немного 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 Слетеера.

SCF в psi4

Теперь можно вернуться к коду и взглянуть на него чуть более осмысленно. При вычислении энергии мы 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 Слетеера. В описании метода SCF мы собирались начинать итерации с водородоподобных волновых функций, но так как весь метод является аппроксимирующим, нам никто не мешает выбрать другие волновые функции, если результаты лучше согласуются с экспериментом. Выбор базиса может существенно влиять на результат вычислений и современные базисы сложнее, чем Слетеоровский детерминант – он просто один из первых и наиболее популярных.

Параметр `reference` означает, какие предположения о волновой функции мы 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({
    '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")
```

Та-дам! Используя более прокачанные базисы, мы получили правильный ответ.

Давайте посмотрим, что еще можно сделать с помощью self-consistent field.

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")
```

Экспериментальное значение энергии атома гелия [равно](#) -79.0 eV.

Молекула водорода

Пока мы рассматривали только атомы, но SCF можно использовать и для молекул – потенциалы становятся сложнее, электронов больше, но общая логика не меняется.

```
psi4.core.clean()

h_mol = psi4.geometry("""
H 0 0 0
H 0 0 0.74
""") # задали 2 атома водорода с явными координатами

psi4.set_options({
    'basis': 'STO-3G',
    'reference': 'rohf',
})

e_in_ht_h = psi4.energy('scf', molecule=h_mol)
print(f"Hydrogen ground state energy: {e_in_ev(e_in_ht_h)} eV")
```

Здесь мы 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 мы зададим неправильные координаты, то рассчитанная энергия окажется неверной. Точнее, она соответствовала бы нефизической 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
""") # неверное расстояние в ангстремах

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")
```

Для некорректной геометрии получилась завышенная энергия, а после оптимизации – почти что такая же энергию, 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.

Молекула этилового спирта

Молекула водорода – это все еще почти игрушечный пример. Давайте попробуем обсчитать молекулу этанола.

Задавать руками геометрию молекулы C2H5OH 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")

```

Итоги

Мы 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 пользоваться SCF в python пакете psi4.

В примерах мы везде вычисляли ground state энергию, но, конечно, зная волновые функции, можно посчитать многое 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 рассеяния рентгена на молекуле... Квантовая механика постулирует, что любая измеримая величина является усреднением определенного оператора по волновой функции, поэтому возможности ограничены в основном вычислительной сложностью, а не теорией.

Для более глубокого погружения в практику квантовой химии можно пройти лабораторные работы psi4: [раз](#), [два](#).

Список литературы

[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>](https://doi.org/10.1007/s11075-018-0562-7).

[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>](https://doi.org/10.1038/s41586-019-1666-5).

[HavlíčekCorcolesT+19]

Vojtěch Havlíček, Antonio D Cárcoles, 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.

[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.

[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.

[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.

[Sch21]

Maria Schuld. Quantum machine learning models are kernel methods. *arXiv e-prints*, pages 26 pages, 2021. URL: <https://arxiv.org/abs/2101.11020>, arXiv:2101.11020.

[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.

[89]

Лифшиц Е. М. Ландау Л. Д. *Квантовая механика: Нерелятивистская теория*. Наука, 1989. URL: <https://www.math.purdue.edu/~eremenko/dvi/LL.pdf>.

[15]

Иванов М.Г. *Как понимать квантовую механику. Регулярная и хаотическая динамика*, 2015. URL: <https://mipt.ru/upload/medialibrary/533/quant-2.pdf>.

Список авторов курса

Создатели курса

1. [Семен Синченко](#)
2. [Юрий Кашницкий](#)
3. [Виктор Трохименко](#)

Основные авторы

1. [Семен Синченко](#)
2. [Илья Беседин](#)
3. [Сергей Ширкин](#)
4. [Александр Березутский](#)

Основные рецензюеры

1. [Юрий Кашницкий](#)
2. [Виктор Трохименко](#)
3. [Борис Зимка](#)
4. [Николай Карелин](#)
5. [Евгений Желтоножский](#)

Редакторы

1. [Наталья Маркова](#)

Техническая поддержка

1. [Дмитрий Коржов](#)

Полный список авторов

A	▼
Б	▼
В	▼
Г	▼
Д	▼

Е

Ж

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Ш

Э

Ю

Я

Благодарности

Команда курса выражает благодарность:

- сообществу [Open Data Science](#) за предоставление платформы для размещения курса и техническую поддержку;

Глоссарий

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/>

Б

Бра-вектор

Англ. bra-vector

В

Вакуумное состояние

Англ. Vacuum state, также используется название "основное состояние" (basic state). Так часто называется квантовое состояние с вектором $|0\rangle$. 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

Г

Гейты

Англ. Quantum gates

Гейт Адамара

Англ. Hadamard gate

Гейт C-NOT

Гейт NOT

T-гейт

Гейты поворота

Phase-shift гейт

Единичный гейт

Гейт iSWAP

fSim

И

Измерение

Англ. Quantum measurement

К

Квантовая информация

Также 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

Также кубит или q-бит, англ. Qubit, от quantum bit. Минимальная единица информации на квантовом компьютере. Как и (классический) бит, кубит допускает два состояния, которые обычно обозначаются кет-векторами $|\langle\text{ket}\{0\}\rangle\rangle$ и $|\langle\text{ket}\{1\}\rangle\rangle$, а на уровне реализации кубит – это двухуровневая квантовая система, такая как две поляризации фотона или спина электрона.

M

Матрица плотности

Также используется термин "оператор плотности", англ. Density matrix или Density operator

Метод квантового локального подбора

(QLS, quantum local search), <https://www.osp.ru/os/2019/03/13055118>

Многокубитные гейты

Англ. Multiqubit gates

H

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 $|\langle\text{ket}\{|\Psi\rangle\rangle\rangle$ обозначает вектор состояния, $|\langle\text{bra}\{|\Psi\rangle\rangle\rangle$ – сопряженный вектор состояния, а $|\langle\text{bra}\{|\Psi\rangle\rangle\rangle|\hat{H}|\langle\text{ket}\{|\Psi\rangle\rangle\rangle\rangle$ – среднее (математическое ожидаемое) значения наблюдаемой для оператора $|\langle\hat{H}|\rangle\rangle$ в состоянии $|\langle\text{ket}\{|\Psi\rangle\rangle\rangle$ (также говорят о свертке оператора $|\langle\hat{H}|\rangle\rangle$ с бра-вектором $|\langle\text{bra}\{|\Psi\rangle\rangle\rangle$ и кет-вектором $|\langle\text{ket}\{|\Psi\rangle\rangle\rangle$). Система обозначений Дирака позволяет отделить вектора состояния $|\langle\text{ket}\{|\Psi\rangle\rangle\rangle$ от самого состояния $|\langle|\Psi\rangle\rangle\rangle$ или конкретного математического описания (в виде волновой функции или вектора).

O

Операторы Паули

Также используется термин "матрицы Паули", англ. Pauli operators или Pauli matrices

P

Правило сдвига параметров

(*) Англ. Parameter-shift rule

C

Смешанные состояния

Англ. Mixed states

Собственный вектор

Англ. Eigenvector

Собственное значение

Англ. Eigenvalue

Соотношение неопределенности

Также используется термин "принцип неопределенности", англ. Uncertainty relation или Uncertainty principle

Суперпозиция

Англ. Superposition

Сфера Блоха

Англ. Bloch sphere

У

Унитарный оператор

Англ. Unitary operator

Ш

«Шумные» квантовые компьютеры промежуточного масштаба (Noisy intermediate-scale quantum, NISQ, NISQ Hardware)

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 термины русский <https://www.osp.ru/os/2019/03/13055130>

Э

Энтропия

Англ. Entropy

By ODS Quantum Community

© Copyright 2021.