Contents

Приветствие

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

Почему именно этот курс?

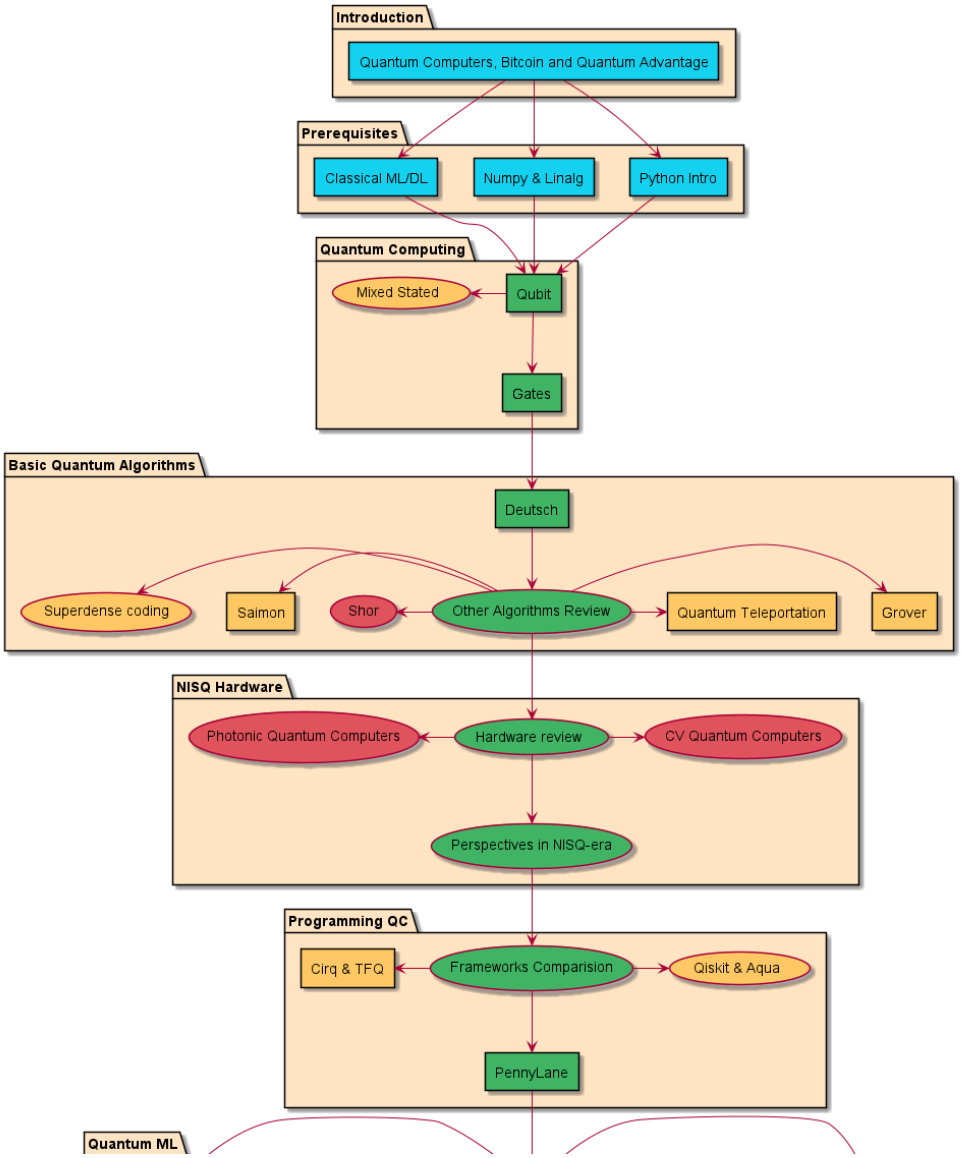
Наш курс отличается от других курсов по квантовым вычислениям:

- 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.



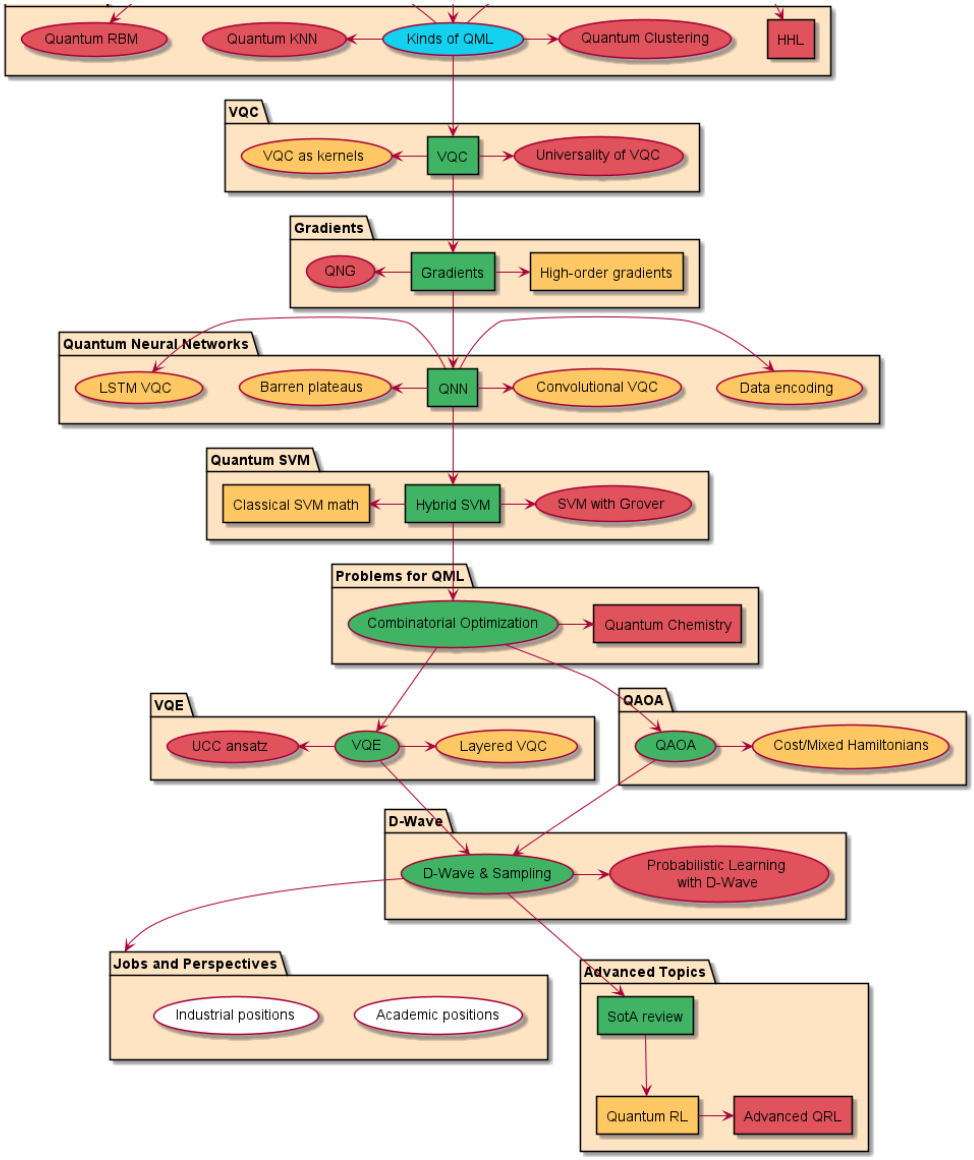


Fig. 1 Программа курса

Как будет проходить этот курс?

Рекомендуем проходить курс в порядке, обозначенном на схеме.

Желаем успехов!

О 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?

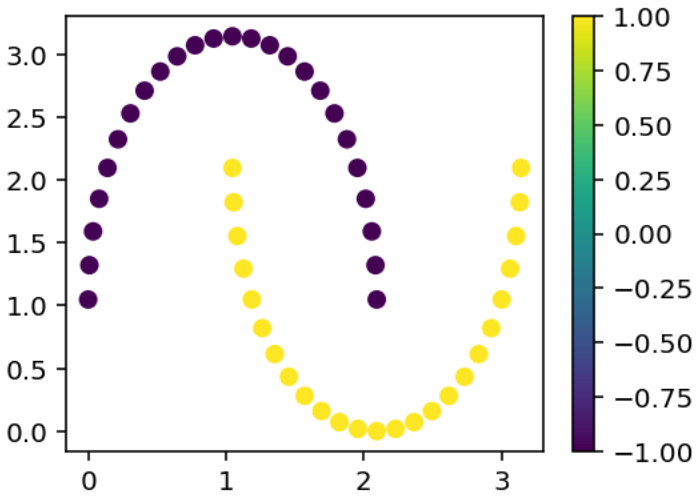
Задача лекции

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 $\{0, 1\} \to \{-1, 1\}$, а признаки $\mathbf{X}$ переведем в $[0, \pi]$:

```
def normalize(x):
    """
    Переводит значения в интервал от 0 до pi
    """
    min_ = x.min()
    max_ = x.max()
    return np.pi * (x - min_) / (max_ - min_)

x[:, 0] = normalize(x[:, 0])
x[:, 1] = normalize(x[:, 1])
y = y * 2 - 1

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



Вариационная схема

Перед тем как мы начнем обучение и будем считать градиенты, нам необходимо определиться с тем, как будет выглядеть наша вариационная схема. Мы посвятим кодированию данных, выбору архитектуры схемы, а также измеряемого оператора еще много занятий. Так что пока просто воспользуемся кодированием признаков \(\mathbf{X}\) вращениями, а сверху применим несколько параметризованных слоев вращений.

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 \(\hat{Rot}(\theta_1^1, \theta_1^2, \theta_1^3)\)
- Вращение 2-го кубита \(\hat{Rot}(\theta_2^1, \theta_2^2, \theta_2^3)\)
- “Запутывающий” оператор, который действует на оба кубита сразу – в нашем случае это \(\hat{CZ}\)

Как видно, на каждый “слой” у нас приходится шесть параметров. Реализуем это в коде:

```
def layer(theta):
    qml.Rot(theta[0, 0], theta[0, 1], theta[0, 2], wires=0)
    qml.Rot(theta[1, 0], theta[1, 1], theta[1, 2], wires=1)
    qml.CZ(wires=[0, 1])
```

Здесь у нас вращения каждого из кубитов по сфере Блоха и двухкубитное взаимодействие \(\hat{CZ}\).

Все вместе

Теперь давайте объединим все это вместе, добавим пару наблюдаемых и оформим как `qml.qnode`:

```
@qml.qnode(dev)
def node(x1, x2, q):
    encoding(x1, x2)
    for q_ in q:
        layer(q_)

    return qml.expval(qml.PauliZ(0) @ qml.PauliY(1))
```

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:

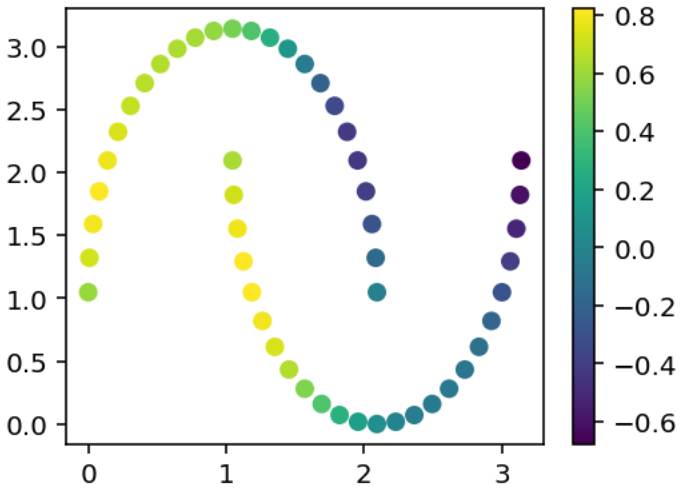
- итерироваться по строкам двумерного массива
- для каждой строки вычислять результат схемы – это функция от \(x_1, x_2, \theta\) – массив параметров \(\theta\) у нас первый элемент `tuple`
- добавлять смещение – это второй элемент `tuple`
- результат добавлять в итоговый массив

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 отклонения. Это не самый лучший выбор для задач классификации, но зато самый простой. Простой вариант – это именно то, что нам нужно в этой лекции:

```
def cost(q, x, y):
    preds = np.array(apply_node(x, q))
    return np.mean(np.square(preds - y))
```

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`

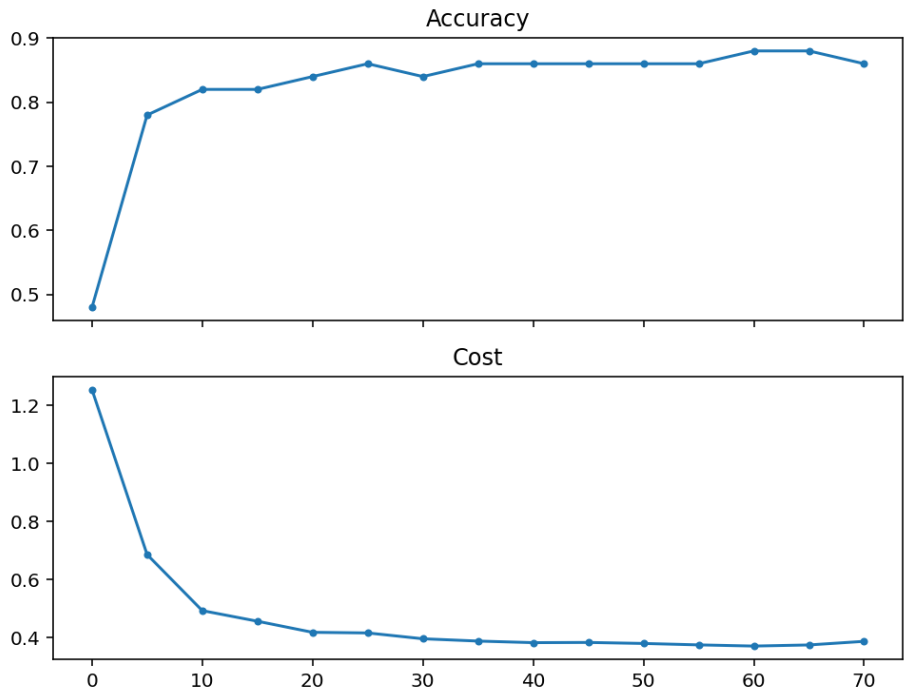
По больше части, на наших занятиях мы будем использовать `NumPy` из-за простоты и привычности. Перед тем как разбираться с тем, как же именно происходит дифференцирование квантовой схемы, давайте посмотрим на весь цикл обучения.

> **Note**
>
> Внимание, процесс обучения на обычном ноутбуке может занять около минуты! Это связано с трудностью симуляции квантового компьютера на классическом.

```
opt = qml.optimize.GradientDescentOptimizer(stepsize=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))
```

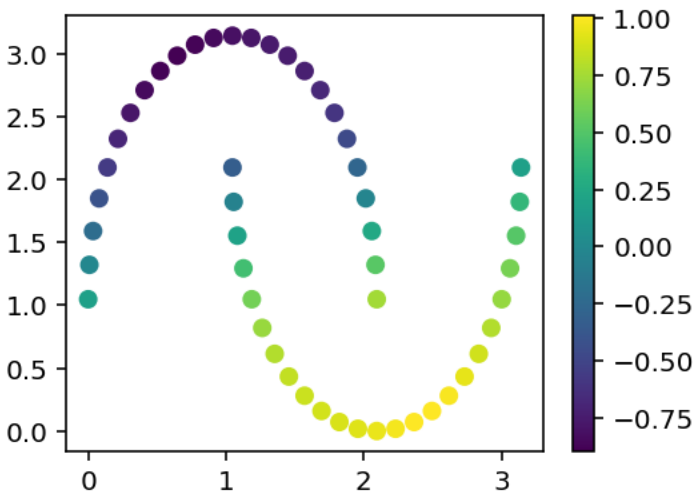
И посмотрим на получившиеся графики точности и функции потерь:

```
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 геометрическом (или визуальной) интерпретацией градиента функции. Правильно, градиент в каждой точке – это касательная. А приближенное значение угла наклона любой прямой можно найти, взяв конечные отрезки:
$$\frac{df}{dx} \simeq \frac{\Delta f}{\Delta x} = \frac{f(x + \Delta x) - f(x)}{\Delta x}$$
Давайте попробуем посчитать градиент нашей квантовой схемы. Для этого инициализируем ее заново случайными параметрами, а потом сравним это с тем значением, которое считает `Autograd`.

```
np.random.seed(42)
q = (np.random.uniform(0, np.pi, size=(4, 2, 3)), 0.0)


def naive_grad(cost, params, x, y, f0, delta):
    return (cost(params, x, y) - f0) / delta

def grad_i(q, f0, cost, x, y, i):
    new_params = q[0].copy().flatten()
    new_params[i] += 0.05

    return naive_grad(cost, (new_params.reshape(q[0].shape), q[1]), x, y, f0, 0.05)
```

В качестве эталона возьмем тот градиент, который нам считает `Autograd`:

```
grad = qml.grad(cost, argnum=0)
```

Сравним первые пять значений `Autograd` с нашим наивным алгоритмом, взяв $\Delta = 0.05$:

Note

Осторожно! Так как мы считаем градиенты очень наивно и на всех точках сразу, то следующий блок кода работает долго!

```
autograd = grad(q, x, y)
f0 = cost(q, x, y)
pretty_print = ""
for i in range(10):
    g_i = grad_i(q, f0, cost, x, y, i)
    pretty_print += f"Naive grad: {g_i:.3f}\tAutograd result: {autograd[0].flatten()
[i]:.3f}\n"

print(pretty_print)
```

```
Naive grad: -0.028      Autograd result: -0.029
Naive grad: 0.081       Autograd result: 0.081
Naive grad: 0.025       Autograd result: 0.025
Naive grad: 0.013       Autograd result: 0.008
Naive grad: -0.000      Autograd result: -0.011
Naive grad: 0.009       Autograd result: 0.004
Naive grad: 0.025       Autograd result: 0.025
Naive grad: 0.002       Autograd result: -0.003
Naive grad: -0.068      Autograd result: -0.069
Naive grad: 0.009       Autograd result: 0.004
```

Можно заметить, что даже с таким большим значением Δ наши оценки получились достаточно близкими к тем, которые получены в `Autograd`. Хотя, конечно, для некоторых значений расхождения заметны и иногда они даже в знаке частной производной.

Parameter-shift rule

Более точная оценка может быть получена методом, который называется **Parameter shift**. Он основан на том, что для квантового “черного ящика” $\hat{U}(\theta)$, которым является наша схема, частная производная по параметру θ_i выражается так:
$$\nabla_{\theta_i}\langle \hat{U} \rangle (\mathbf{\theta}) = \frac{1}{2}[\langle \hat{U} \rangle (\mathbf{\theta} + \frac{\pi}{2}\mathbf{e_i}) - \langle \hat{U} \rangle (\mathbf{\theta} - \frac{\pi}{2}\mathbf{e_i})]$$

> **Note**
>
> Более строгую формулировку, а также вывод правила parameter-shift можно посмотреть в продвиной лекции этого блока про производные высших порядков.

Если по-простому, то оценка частной производной по i-му параметру может быть получена вычислением сначала ожидаемого значения схемы с параметром θ_i, смещенным на $\frac{\pi}{2}$ в одну сторону, а потом – в другую. Давайте запишем это в коде, но перед этим давайте вспомним, как будет выглядеть производная функции потерь (а именно она нам нужна):
$$\frac{\partial L}{\partial \theta_i} = \frac{\partial (\langle \hat{U} \rangle (\theta) - y)^2}{\partial \theta_i} = 2 (\langle \hat{U} \rangle - y) \frac{\partial \langle \hat{U} \rangle}{\partial \theta_i}$$
Реализуем явно и наивно эту формулу в коде:

```python
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:

```python
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:
$$ U_{j...k} = U_j, ..., U_k \ket{\Psi} $$

Производная от измерения

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

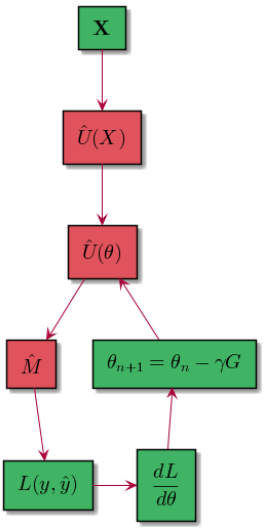


Fig. 2 Квантово-классическая схема

Видно, что мы хотим считать производную не от самой параметризованной схемы \(U_{j...k}\), 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:
\[G_i = \frac{\partial \bra{U_{j...k}\Psi}\hat{M}\ket{U_{j...k}\Psi}}{\partial \theta_i} \]
То есть нам важно посчитать производную от результата измерения, так как именно результат измерения у нас будет определять “предсказание” нашей квантовой нейронной сети. Причем нам нужно уметь считать производную от любого параметра \(\theta_i\) в цепочке \(\theta_j, ...\theta_i, ...\theta_k\).

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 \(\Psi_i\), которое пришло на вход \(i\)-го гейта в нашей последовательности:
\[\langle M(\theta) \rangle = Tr(M U_{k, ..., 1} \rho_i U_{k, ..., 1}^\dagger) \]
где \(\rho\) это матрица плотности (\(\ket{\Psi}\bra{\Psi}\)). Подробнее о матрицах плотности можно почитать в ранней продвинутой лекции про смешанные состояния.

Тогда частная производная от математического ожидания по \(i\)-му параметру \(\theta_i\) записывается (подробнее в [MNKF18]) через коммутатор исходного состояния \(\rho\), которое “пришло” на вход гейта \(U_i\) и того оператора Паули \(P_i\), который мы используем в \(U_i\):
\[\frac{\partial \langle M \rangle}{\partial \theta_i} = -\frac{i}{2}Tr(M U_{k, ..., i}[P_i, U_{i-1, ..., 1}\rho_i U_{i-1, ..., 1}^\dagger]U_{k, ..., i}^\dagger) \]
Этот коммутатор может быть переписан следующим образом:
\[[P_i, \rho] = i[U_i \Bigl(\frac{\pi}{2} \Bigr) \rho_i U_i^\dagger \Bigl (\frac{\pi}{2} \Bigr) - U_i \Bigl(-\frac{\pi}{2} \Bigr) \rho_i U_i^\dagger \Bigl (-\frac{\pi}{2} \Bigr)] \]
Тогда соответствующий градиент \(\frac{\partial}{\partial \theta_i}\) можно записать через смещения на \(\pm\frac{\pi}{2}\):
\[\begin{split} \begin{gathered} \frac{\partial \langle M \rangle}{\partial \theta_i} = \frac{\langle M_i^+ \rangle - \langle M_i^-\rangle}{2} \\ \langle M_i^{\pm} \rangle = \frac{1}{2} Tr [M U_{k, ..., i+1} U_i(\pm \frac{\pi}{2})\rho_i^` U_i^\dagger (\pm \frac{\pi}{2}) U_{k, ..., i+1}^\dagger] \\ \rho_i^` = U_{j, ..., 1} \rho_i U_{j, ..., 1}^\dagger \end{gathered} \end{split}\]
По аналогии с классическими нейронными сетями и *backpropagation* (для тех, кто забыл это понятие, рекомендуем вернуться к вводным лекциями про классическое машинное обучение) тут явно можно выделить *forward* проход со смещением \(\theta_i\) на значения \(\frac{\pi}{2}\) и *backward* со смещением на \(-\frac{\pi}{2}\).

Обобщенный parameter-shift

Предложенное в [MNKF18] выражение может быть на самом деле получено в более общем виде из других соображений. Так, выражение для нашей наблюдаемой $\langle M \rangle$ может всегда быть представлено [MBK21] как сумма вида:
$$ \bra{U_i(\theta_i)}\hat{M}\ket{U_i(\theta_i)} = \hat{A} + \hat{B}\cos{\theta_i} + \hat{C}\sin{\theta_i} $$
где $\hat{A}, \hat{B}, \hat{C}$ – операторы, не зависящие от параметра θ_i.

> **Note**
>
> Действительно, явно выписав выражение для наблюдаемой и вспомнив формулы для косинуса и синуса двойного угла, а также воспользовавшись тем, что $U(\theta) = e^{-\frac{1}{2}H\theta} = \cos{\frac{\theta}{2}}\mathbf{1} - i\sin{\frac{\theta}{2}}H$, получаем:
> $$\begin{split} \begin{gathered} (\cos{\frac{\theta}{2}}\mathbf{1} + i\sin{\frac{\theta}{2}}H)\hat{M}(\cos{\frac{\theta}{2}}\mathbf{1} - i\sin{\frac{\theta}{2}}H) = \\ \cos^2{\frac{\theta}{2}}\hat{M} + i \sin{\frac{\theta}{2}}\cos{\frac{\theta}{2}}H\hat{M} - i \sin {\frac{\theta}{2}}\cos{\frac{\theta}{2}}\hat{M}H + \sin^2{\frac{\theta}{2}}H\hat{M}H = \\ \frac{1}{2} \cos{\theta}\hat{M}+\frac{1}{2}\hat{M}+\frac{i}{2} \sin{\theta}H\hat{M} - \frac{i}{2}\sin{\theta}\hat{M}H + \frac{1}{2}H\hat{M}H - \frac{1}{2}\cos{\theta}H\hat{M}H = \\ \frac{1}{2}(\hat{M} + H\hat{M}H) + \frac{1}{2}(\hat{M} - H\hat{M}H)\cos{\theta} + \frac{i}{2}(H\hat{M} - \hat{M}H)\sin{\theta} \end{gathered} \end{split}$$

Тогда можно воспользоваться правилами тригонометрии, а именно, тем что для любого $s \neq k\pi, \text{ } k \in \{1, 2, ..., \}$ справедливо:
$$\begin{split} \begin{gathered} \frac{d \cos {\theta}}{d \theta} = \frac{\cos (\theta + s) - \cos (\theta - s)}{2\sin{s}} \\ \frac{d \sin {\theta}}{d \theta} = \frac{\sin (\theta + s) - \sin (\theta - s)}{2\sin{s}} \\ \end{gathered} \end{split}$$
И подставим это в выражение для $\frac{\partial \langle M \rangle}{\partial \theta_i}$:
$$ \frac{\partial \langle M(\theta_i) \rangle}{\partial \theta_i} = \frac{\langle M(\theta_i + s) \rangle - \langle M(\theta_i - s) \rangle}{2\sin{s}} $$
Легко заметить, что подстановка сюда $s = \frac{\pi}{2}$ дает нам классический *parameter shift*, описанный в [MNKF18].

Наконец, запишем полученное выражение в более удобном виде, который позволит нам более эффективно выписывать производные высших порядков. Для этого введем вектор $\mathbf{e_i}$ – единичный вектор для i-го параметра, то есть вектор, где все компоненты кроме i-й равны нулю, а i-я равна 1. Тогда наше финальное выражение для обобщенного *parameter shift* примет следующий вид:
$$ \boxed{\frac{\partial f(\mathbf{\theta})}{\partial \theta_i} = \frac{f(\mathbf{\theta} + s\mathbf{e_i}) - f(\mathbf{\theta} - s\mathbf{e_i})}{2\sin{s}}} $$

Вторая производная и гессиан

В классической теории оптимизации, также как и в машинном обучении, очень часто на первый план выходят так называемые методы 2-го порядка. Эти методы похожи на обычный градиентный спуск, но для ускорения сходимости они также используют информацию из матрицы вторых производных, которая называется гессианом. Более подробно про методы 2-го порядка и гессиан можно посмотреть в вводных лекциях курса.

Методы второго порядка требуют больше вызовов, чтобы вычислить гессиан, но взамен они обеспечивают гораздо лучшую сходимость, а также менее склонны “застревать” в локальных минимумах. Это обеспечивает, в итоге, более быстрое обучение. В классических нейронных сетях вычисление гессиана это часто проблема, так как это матрица размерности $\sim O(N^2)$, где N – число весов нейронной сети, и эта матрица получается слишком большой. Но, как мы помним, основная “фича” **VQC** это их экспоненциальная экспрессивность – возможность линейным числом параметров (и гейтов) обеспечить преобразование, эквивалентное экспоненциальному числу весов классической нейронной сети. А значит, для них проблема размерности гессиана не стоит так остро. При этом использование гессиана теоретически позволит в итоге обучить **VQC** за меньшее число вызовов. Именно поэтому методы второго порядка потенциально очень интересны в квантово-классическом обучении. Но для начала нам необходимо разобраться, как именно можно посчитать матрицу вторых производных.

Пользуясь обобщенным правилом *parameter shift*, можно выписать выражение для второй производной [MBK21]:
$$ \frac{\partial_2 f}{\partial \theta_i \theta_j} = \frac{f(\mathbf{\theta} + s_1\mathbf{e_i} + s_2\mathbf{e_j}) + f(\mathbf{\theta} - s_1\mathbf{e_i} - s_2 \mathbf{e_j}) - f(\mathbf{\theta} + s_1 \mathbf{e_i} - s_2 \mathbf{e_j}) - f(\mathbf{\theta} - s_1 \mathbf{e_i} + s_2 \mathbf{e_j})}{4\sin{s_1}\sin{s_2}} $$
Взяв $s_1 = s_2$, можно упростить это выражение к следующему виду:

\[\begin{split} \begin{gathered} \frac{f(\mathbf{\theta} + s\mathbf{a}) + f(\mathbf{\theta} + s\mathbf{b}) - f(\mathbf{\theta} + s\mathbf{c}) - f(\mathbf{\theta} + s\mathbf{d})}{(2\sin{s}) ^2} \\ \mathbf{a} = \mathbf{e_i} + \mathbf{e_j} \\ \mathbf{b} = -\mathbf{e_i} - \mathbf{e_j} \\ \mathbf{c} = \mathbf{e_i} - \mathbf{e_j} \\ \mathbf{d} = -\mathbf{e_i} + \mathbf{e_j} \end{gathered} \end{split}\]
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 \(s_g\) при вычислении вектора градиента, а также такое значение \(s_h\) при вычислении гессиана, чтобы максимально переиспользовать результаты квантовых вызовов и уменьшить их общее количество.

Внимательно взглянув на выражение для 2-х производных, можно заметить, что оптимизация там возможна при расчете диагональных элементов гессиана. Давайте выпишем выражение для диагонального элемента явно:
\[\frac{f(\mathbf{\theta} + 2s\mathbf{e_i}) + f(\mathbf{\theta} - 2s\mathbf{e_i}) - 2 f(\mathbf{\theta})}{(2\sin{s})^2} \]
Можно заметить, что, например, использование \(s = \frac{\pi}{4}\) для гессиана, а также “стандартного” \(s=\frac{\pi}{2}\) для градиента позволит полностью переиспользовать в диагональных элементах гессиана значения, которые мы получили при расчете градиента. А значение \(f(\mathbf{\theta})\) вообще считается один раз для всех диагональных вызовов.

> **Note**
>
> На самом деле, диагональные элементы гессиана можно использовать и сами по себе, например для квазиньютоновских методов оптимизации, где матрица Гессе аппроксимируется какой-то другой матрицей, чтобы не считать все вторые производные. Например, она может быть аппроксимирована диагональной матрицей, как в работе [And19].

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.
