Contents

Приветствие

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

Почему именно этот курс?

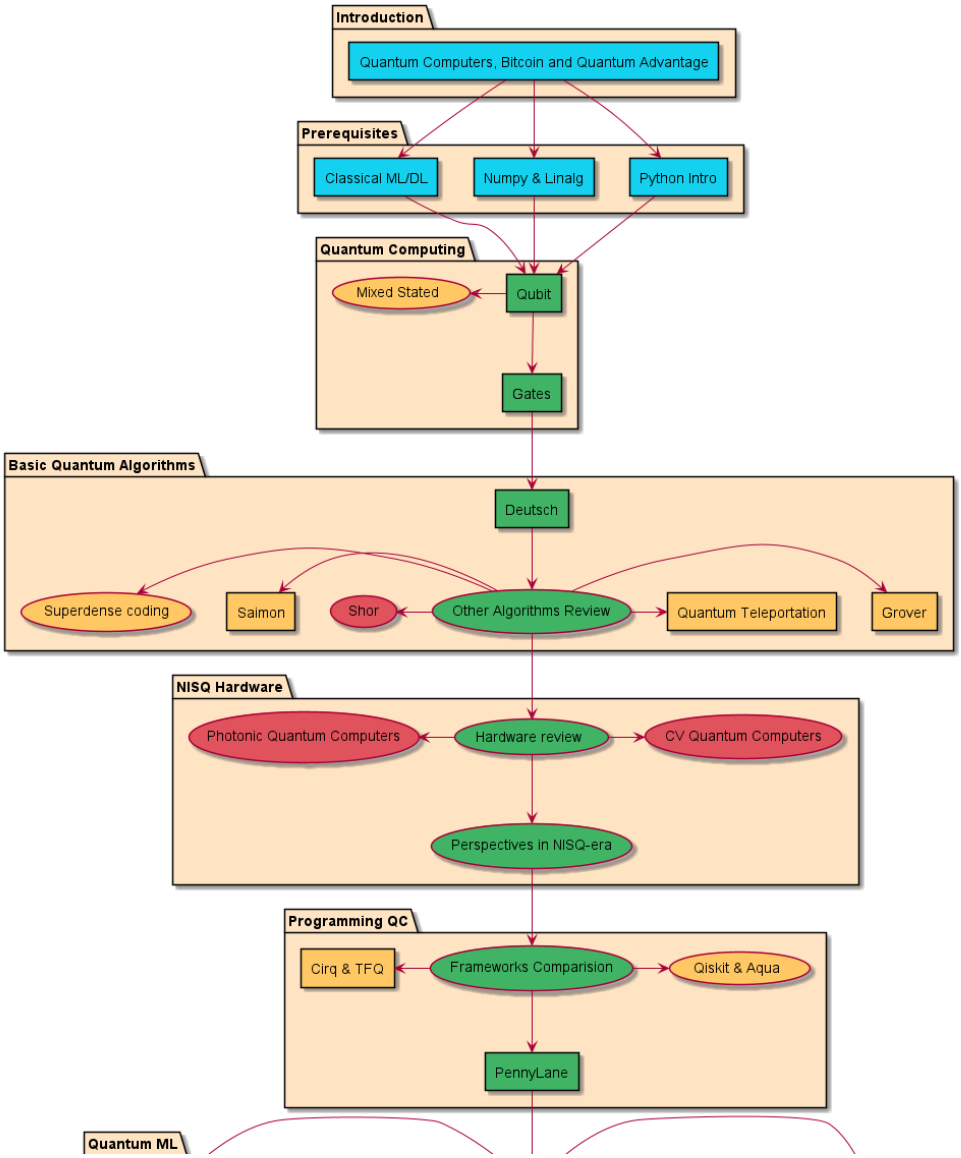
Наш курс отличается от других курсов по квантовым вычислениям:

- 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.



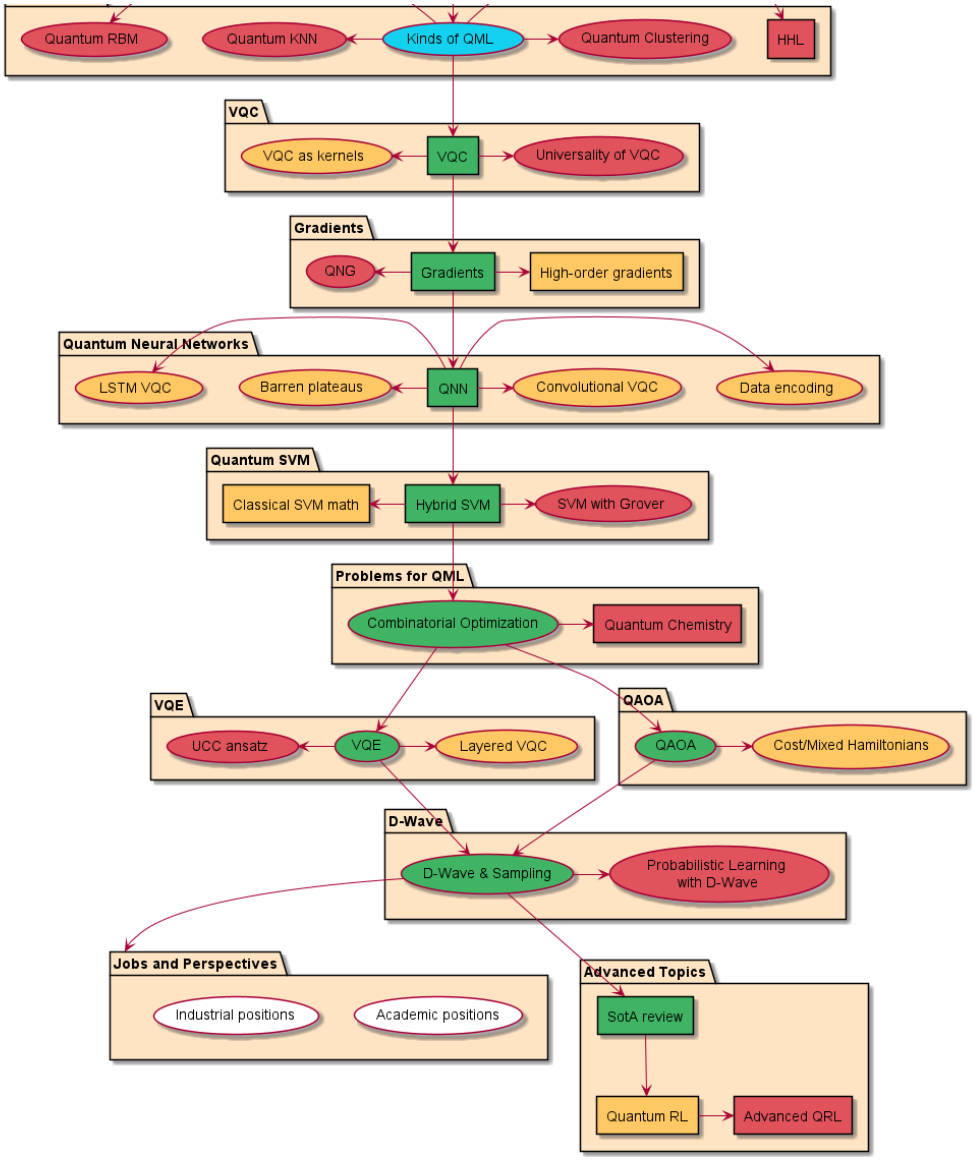


Fig. 1 Программа курса

Как будет проходить этот курс?

Рекомендуем проходить курс в порядке, обозначенном на схеме.

Желаем успехов!

О блоке

Этот блок включает в себя обзор 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 \(|0\rangle\) 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──┌U0──┤ 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`:

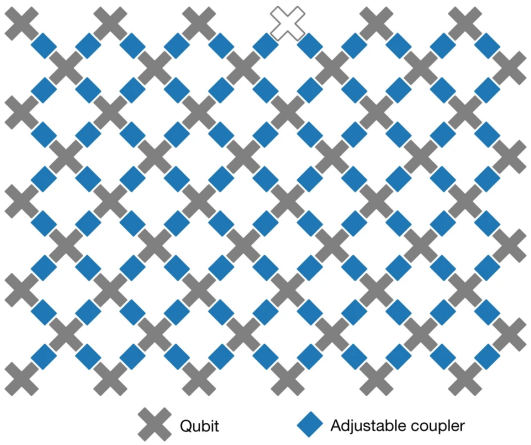


Fig. 2 Изображение из [AAB+19]

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 \(\ket{0}\) и \(\ket{1}\) 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 \(\ket{0}\), то есть если мы будем сэмплировать, то должны получать \(\sim 0.5\). 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 tfq
```

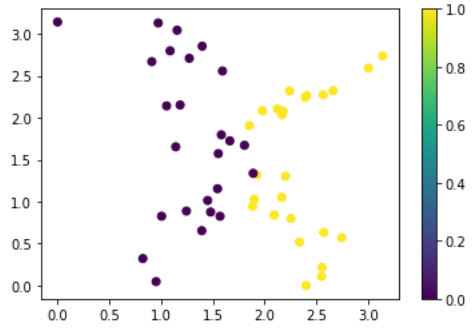
Задача

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 \(\pi\).

```
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” параметризированные 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)
```

```
(0, 0): ──H──Ry(w1)──@──H──Rx(w3)──@──
                     │              │
(0, 1): ──H──Ry(w2)──X──H──Rx(w4)──X──
```

Наблюдаемые

В качестве операторов, которые мы будем измерять, воспользуемся парой \(\hat{XY}\) и \(\hat{YX}\) 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

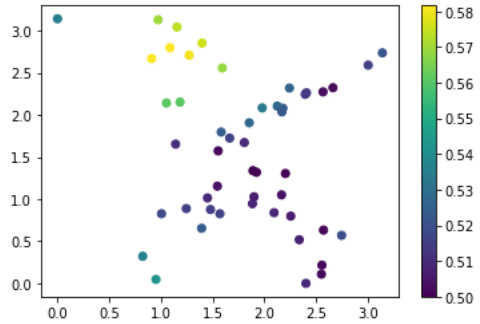
```
model = tf.keras.Model(inputs=cirq_inputs, outputs=output_layer)
model.compile(
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.1),
    loss=tf.keras.losses.BinaryCrossentropy(),
    metrics=[
        tf.keras.metrics.BinaryAccuracy(),
        tf.keras.metrics.BinaryCrossentropy(),
    ]
)
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

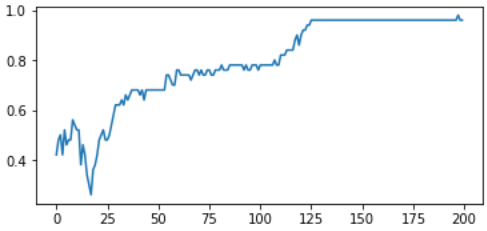
- запустим обучение

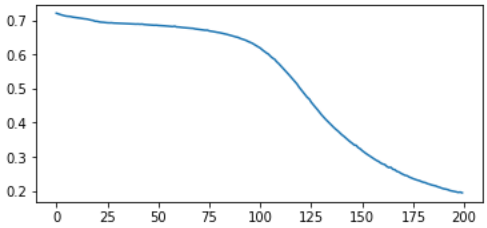
```
model.fit(x=x_input, y=y, epochs=200, verbose=0)
```

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

- визуализируем логи обучения

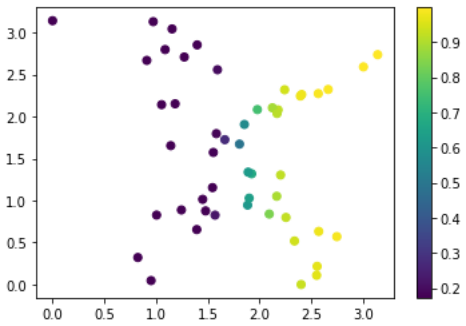
```
f, ax = plt.subplots(2, figsize=(6, 6))
ax[0].plot(model.history.history["binary_accuracy"])
ax[1].plot(model.history.history["binary_crossentropy"])
plt.show()
```





- 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`.
