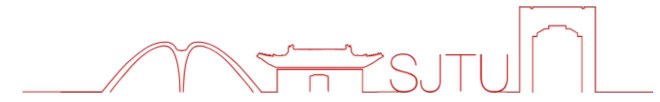
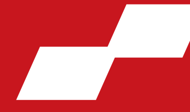


量子机器学习：概念与实践

马汝辉 副教授、博导
计算机科学与工程系
上海交通大学

饮 水 思 源 • 爱 国 荣 校

Credits



- Zaman K, Marchisio A, Hanif M A, et al. A survey on quantum machine learning: Current trends, challenges, opportunities, and the road ahead[J]. arXiv preprint arXiv:2310.10315, 2023.
- Simeone O. An introduction to quantum machine learning for engineers[J]. Foundations and Trends® in Signal Processing, 2022, 16(1-2): 1-223.
- Qisikit Machine Learning. https://qiskit-community.github.io/qiskit-machine-learning/
- TensorFlow Quantum. https://www.tensorflow.org/quantum

1 Quantum Machine Learning

2 Qiskit Machine Learning

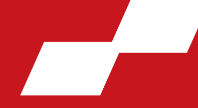
3 TensorFlow Quantum

Overview

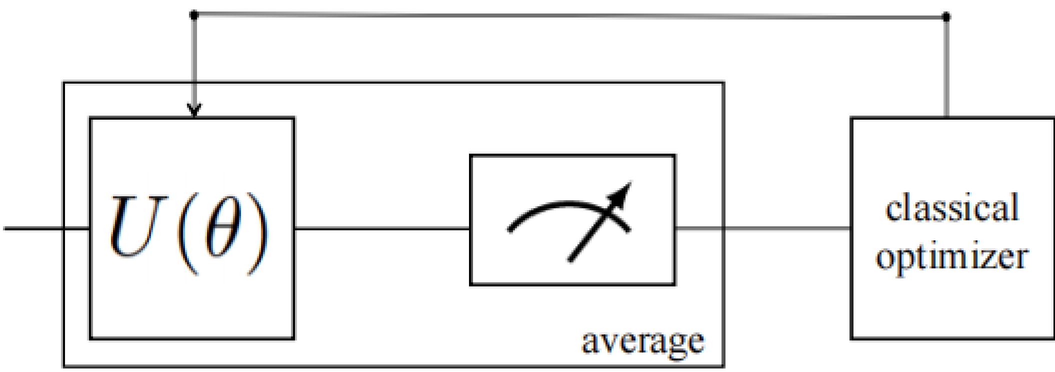
- ML is a class of advanced algorithms that perform a certain task. Given a large number of inputs and desired outputs, an ML model can be trained to make predictions on unseen data. If it is executed on quantum computers, it becomes a quantum ML algorithm.

- Quantum Machine Learning (QML)
  - Fully QML
  - Hybrid QML
    - Quantum Optimization Problems
      - Quantum Approximate Optimization Algorithm
      - Variational Quantum Eigensolver
      - Quantum Annealing
    - Quantum Kernels
      - Quantum Support Vector Machines
      - Quantum Clustering
      - Quantum Autoencoders
    - Quantum Neural Networks
      - Quantum Convolutional Neural Networks
      - Quanvolutional Neural Networks
      - Quantum Recurrent Neural Networks
      - Quantum Generative Adversarial Networks
    - Parametrized Quantum Circuits

Overview

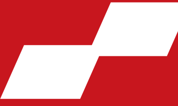


- **Selection of the architecture of a parametric quantum circuit (PQC), also known as ansatz.**
  - Select the architecture of a PQC by specifying a sequence of parametrized quantum gates
  - operation of the PQC is defined by a unitary matrix U(θ), which is dependent on a vector of free parameters θ
- **Parametric optimization**
  - The optimizer is fed measurements of the quantum state produced by the PQC, typically in the form of estimated expectations of observables; and it produces updates to the parameter vector θ.

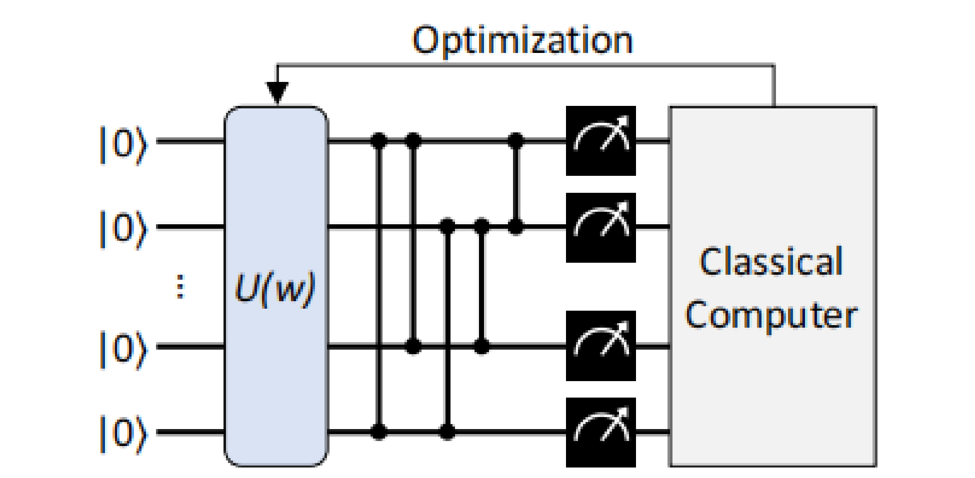


SJTU

Parametrized Quantum Circuits

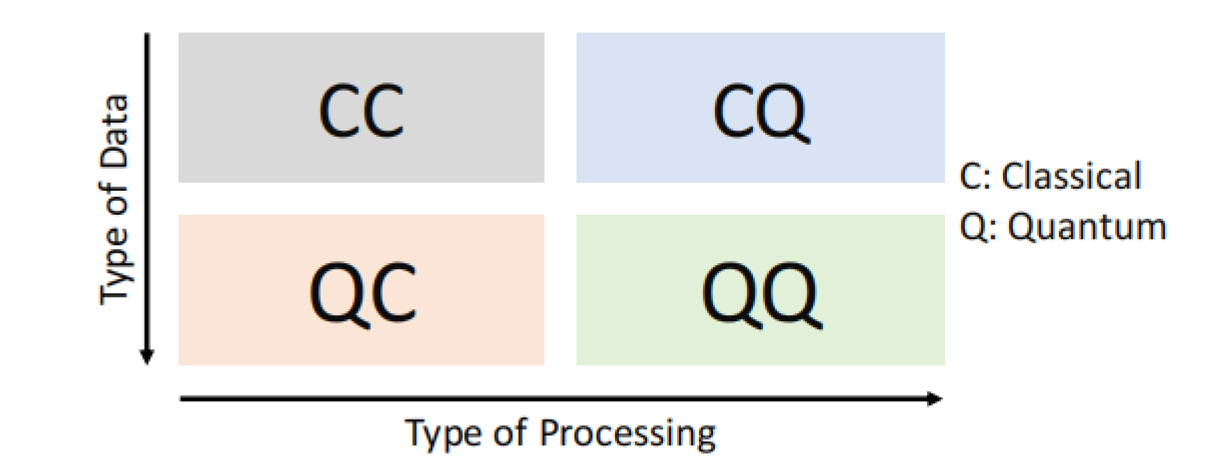


- Variational or Parametrized Quantum Circuits (PQCs) are specific types of quantum algorithms that depend on free parameters.
- PQCs allow us to utilize the existing quantum computers to their full extent.
- In the context of QML, PQCs are used either to encode the data, where the parameters are determined by the data being encoded, or as a quantum model, where the parameters are determined by an optimization process.

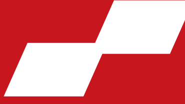


Categorization of QML Approaches

- Before diving into the details of QML algorithms, it is important to characterize different approaches based on the type of data and type of processor used to solve the problem.



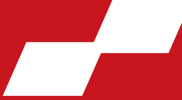
Categorization of QML Approaches



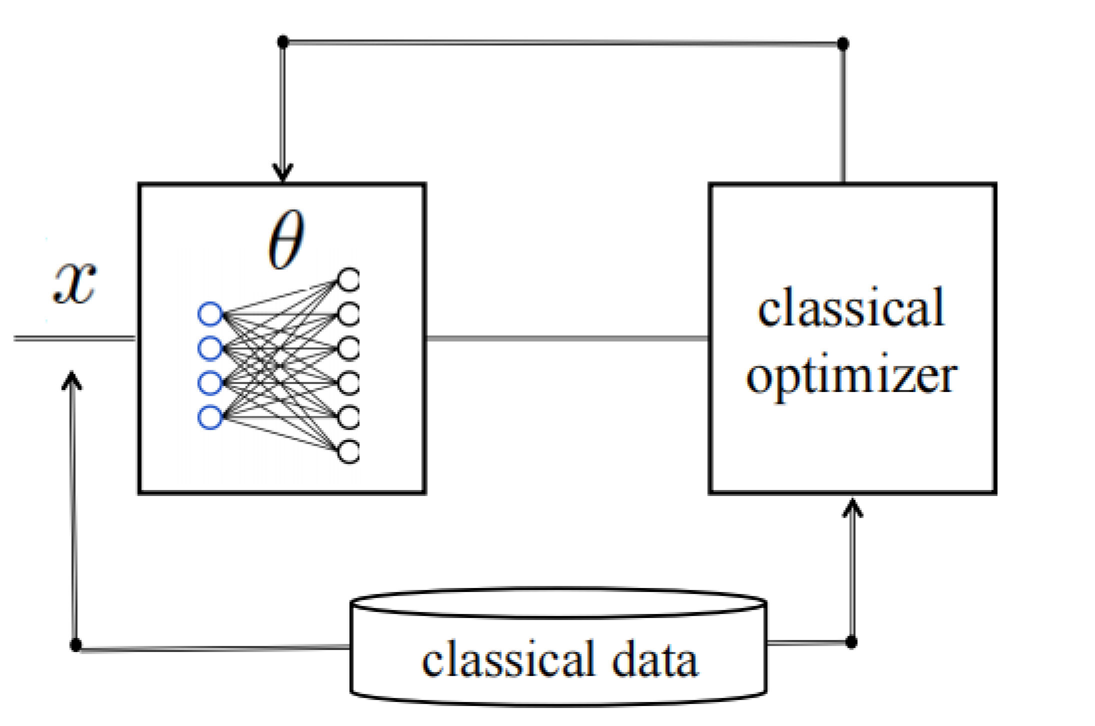
- **CC** refers to processing Classical data using Classical computers, but using algorithms inspired by quantum computing.
- **CQ** refers to processing Classical data using Quantum machine learning algorithms.
  - Main focus
- **QC** refers to processing Quantum data using Classical machine learning algorithms.
  - Active area
- **QQ** refers to processing Quantum data using Quantum machine learning algorithms. It is also known as Fully Quantum Machine Learning (FQML).
  - Future area

SJTU

Categorization of QML Approaches



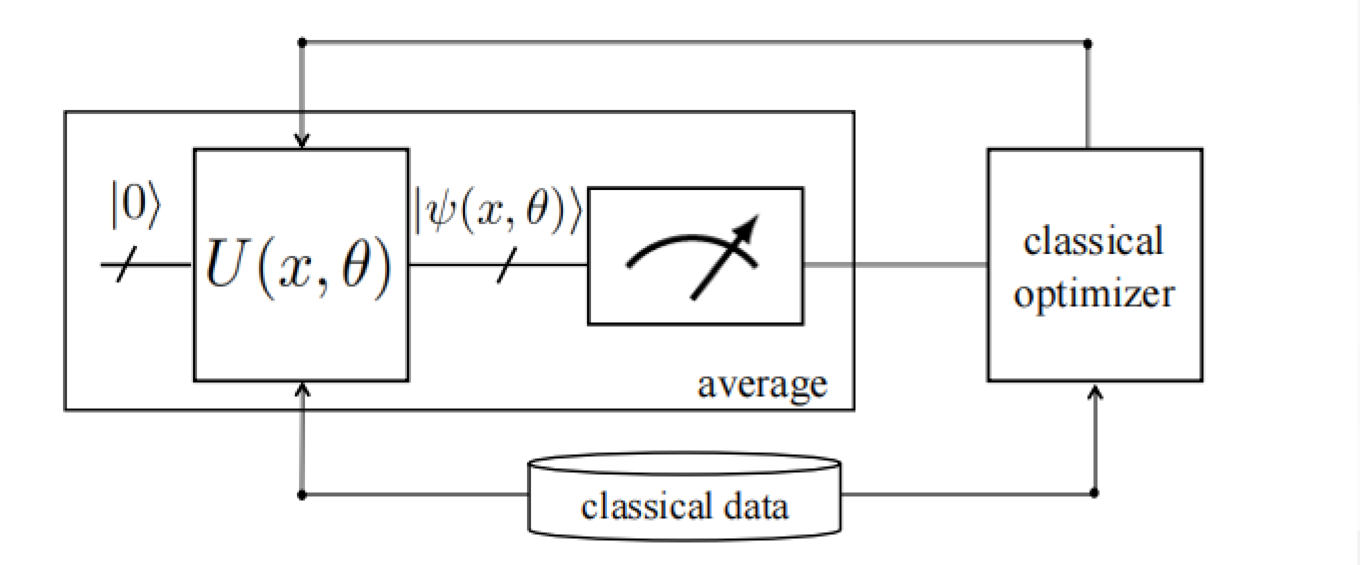
- CC refers to processing Classical data using Classical computers, but using algorithms inspired by quantum computing.



SJTU

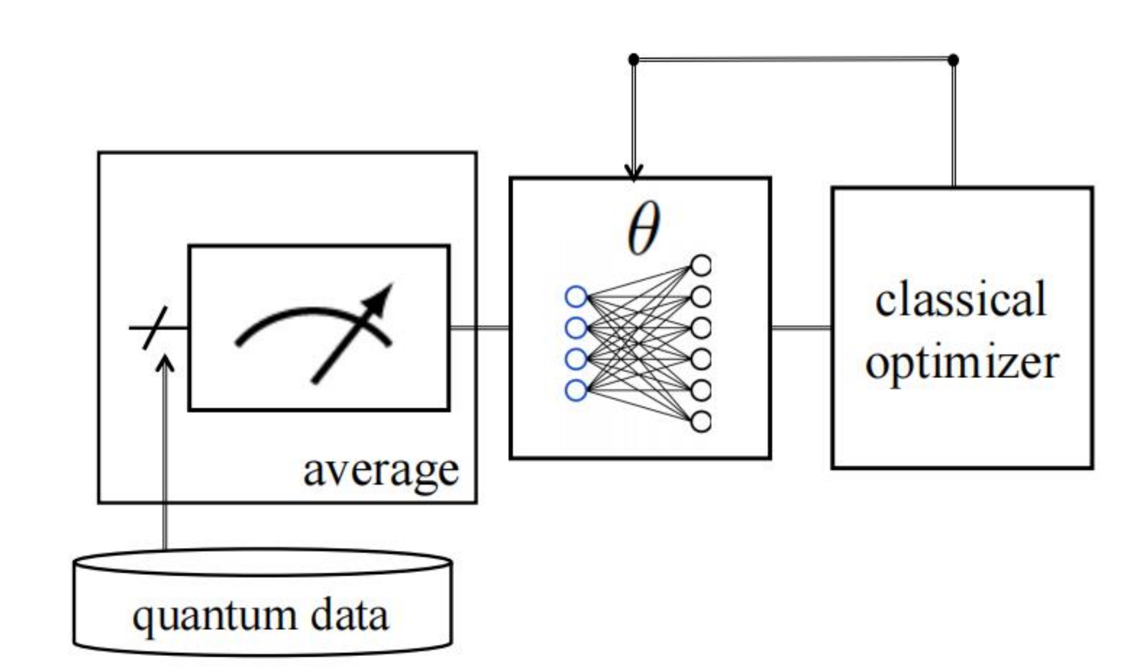
Categorization of QML Approaches

- **CQ refers to processing Classical data using Quantum machine learning algorithms.**
  - Main focus

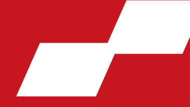


Categorization of QML Approaches

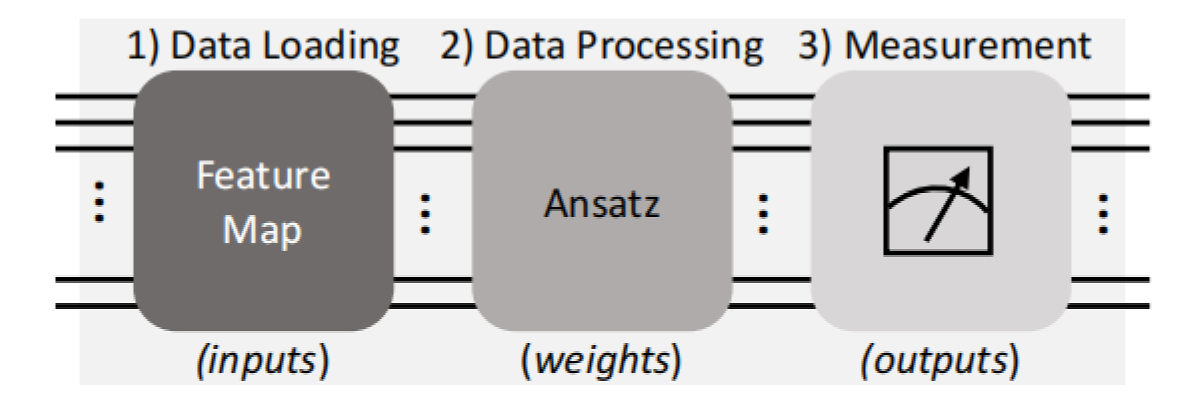
- **QC refers to processing Quantum data using Classical machine learning algorithms.**
  - In the QC case, quantum data are first measured, and then the classical measurement outputs are processed by a classical machine learning model.



Quantum Neural Networks

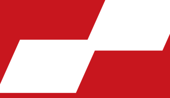


- **Quantum Neural Networks (QNNs) are computational Artificial Neural Network (ANN) models that are based on the principles of quantum mechanics.**
  - The quantum circuit contains a feature map module,
  - an Ansatz module with trainable weights,
  - Measurements are conducted to obtain the outputs.

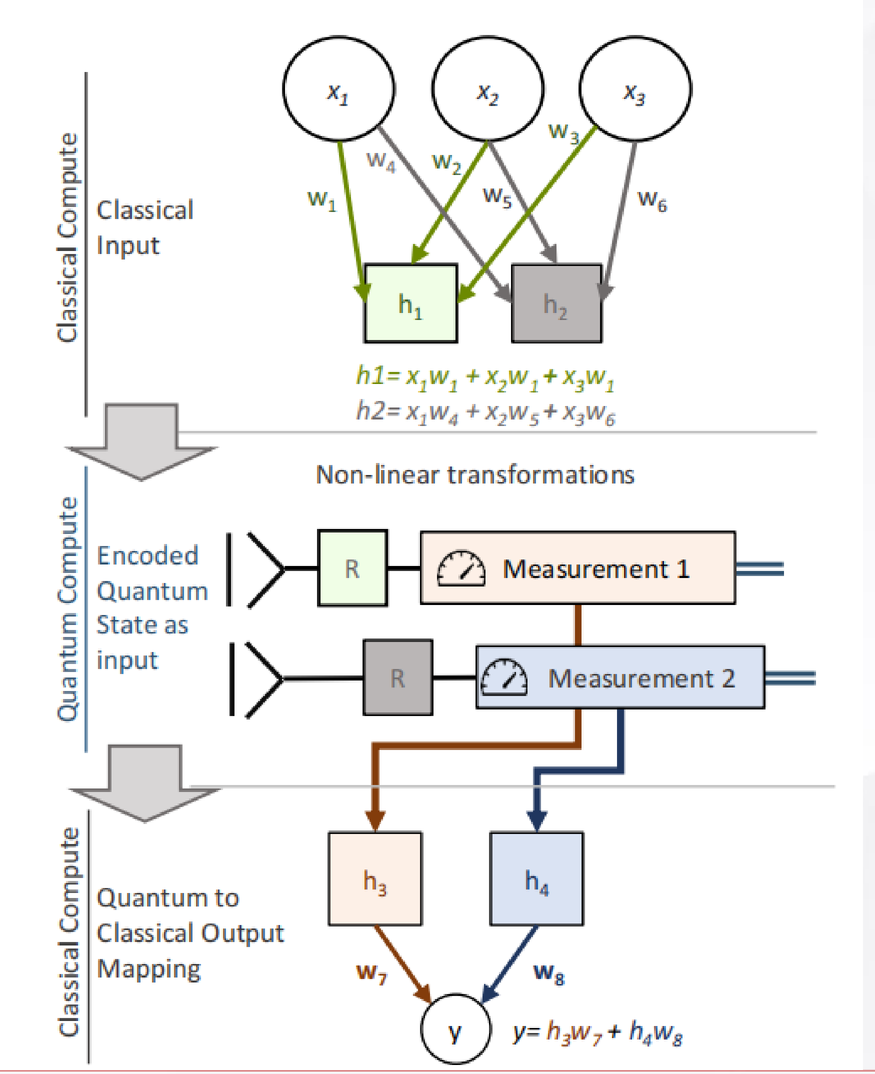


SJTU

Quantum Neural Networks

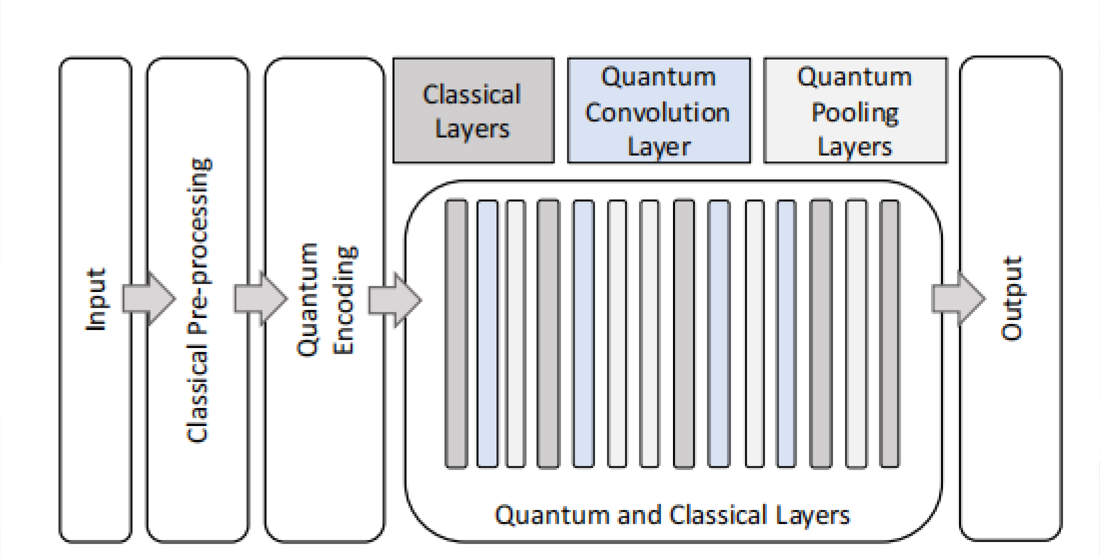


- During the NISQ era, the main focus is on Hybrid Quantum Neural Networks (HQNNs).



Quantum Convolutional Neural Networks

- The structure of a classical CNN consists of applying alternating convolutional layers (with an activation function) and pooling layers, typically followed by fully-connected layers before the output is generated.



02

Qiskit Machine Learning

- Qiskit is pronounced "kiss-kit" 😘, though you may also hear it called "kwis-kit".

Overview

- Qiskit Machine Learning introduces fundamental computational building blocks, such as **Quantum Kernels** and **Quantum Neural Networks**, used in various applications including classification and regression.
- This library is part of the Qiskit Community ecosystem, a collection of high-level codes that are based on the Qiskit software development kit.
- The Qiskit Machine Learning framework aims to be:
  - User-friendly: allowing users to quickly and easily prototype quantum machine learning models without the need of extensive quantum computing knowledge
  - Flexible: providing tools and functionalities to conduct proof-of-concepts and innovative research in quantum machine learning for both beginners and experts
  - Extensible: facilitating the integration of new cutting-edge features leveraging Qiskit's architectures, patterns and related services

What are the main features of Qiskit Machine Learning?

- **Kernel-based methods**
- **Quantum Neural Networks (QNNs)**
  - Qiskit Machine Learning defines a generic interface for neural networks, implemented by two core (derived) primitives: EstimatorQNN and SamplerQNN.
- **Integration with PyTorch**
  - The TorchConnector integrates QNNs with PyTorch.

SJTU

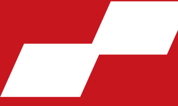
Quantum Neural Networks

Quantum vs. Classical Neural Networks

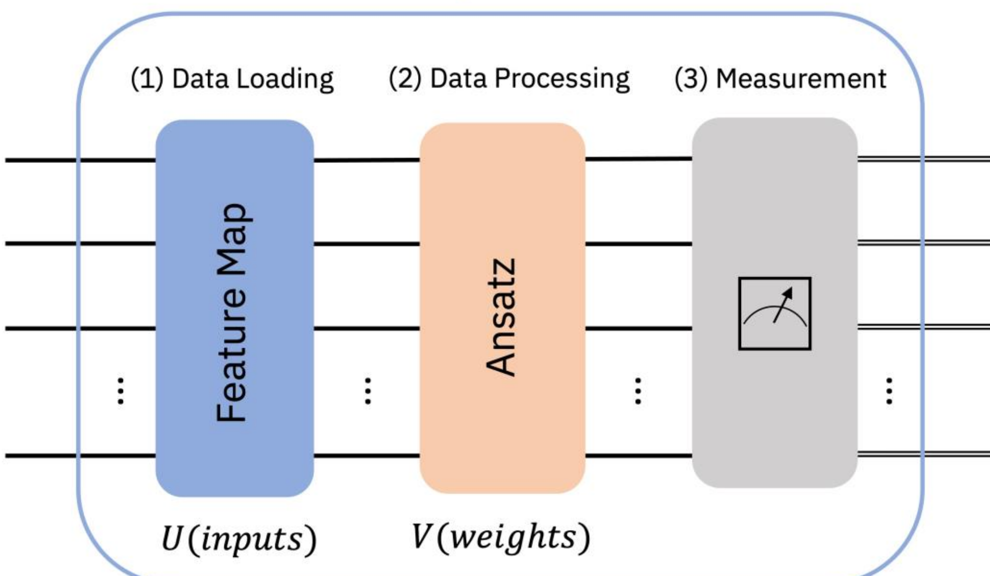
- Classical neural networks are algorithmic models inspired by the human brain that can be trained to recognize patterns in data and learn to solve complex problems.
- The motivation behind quantum machine learning (QML) is to integrate notions from quantum computing and classical machine learning to open the way for new and improved learning schemes.

https://qiskit-community.github.io/qiskit-machine-learning/tutorials/01_neural_networks.html

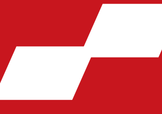
Quantum Neural Networks



- **Because they lie at an intersection between two fields, QNNs can be viewed from two perspectives:**
  - From a machine learning perspective, QNNs are, once again, algorithmic models that can be trained to find hidden patterns in data in a similar manner to their classical counterparts.
  - From a quantum computing perspective, QNNs are quantum algorithms based on parametrized quantum circuits that can be trained in a variational manner using classical optimizers.



Implementation in qiskit-machine-learning



- **The QNNs in qiskit-machine-learning are meant as application-agnostic computational units that can be used for different use cases, and their setup will depend on the application they are needed for. The module contains an interface for the QNNs and two specific implementations:**
  - NeuralNetwork:
    - The interface for neural networks. This is an abstract class all QNNs inherit from.
  - EstimatorQNN:
    - A network based on the evaluation of quantum mechanical observables.
  - SamplerQNN:
    - A network based on the samples resulting from measuring a quantum circuit.

EstimatorQNN

- The **EstimatorQNN** takes in a parametrized quantum circuit as input, as well as an optional quantum mechanical observable, and outputs expectation value computations for the forward pass. The EstimatorQNN also accepts lists of observables to construct more complex QNNs.

[2]:
```
from qiskit.circuit import Parameter
from qiskit import QuantumCircuit

params1 = [Parameter("input1"), Parameter("weight1")]
qc1 = QuantumCircuit(1)
qc1.h(0)
qc1.ry(params1[0], 0)
qc1.rx(params1[1], 0)
qc1.draw("mpl", style="clifford")
```

[2]:

q — H — R_Y (input1) — R_X (weight1) —

EstimatorQNN

- We can now create an observable to define the expectation value computation. If not set, then the EstimatorQNN will automatically create the default observable $Z^{\otimes n}$. Here, n is the number of qubits of the quantum circuit.
- In this example, we will change things up and use the $Y^{\otimes n}$ observable:

[3]:
```
from qiskit.quantum_info import SparsePauliOp

observable1 = SparsePauliOp.from_list([("Y" * qc1.num_qubits, 1)])
```

SJTU

EstimatorQNN

- **Together with the quantum circuit defined above, and the observable we have created, the EstimatorQNN constructor takes in the following keyword arguments:**
  - estimator
  - pass_manager
  - input_params
  - weight_params

```
[4]: from qiskit_machine_learning.neural_networks import EstimatorQNN
     from qiskit.primitives import StatevectorEstimator as Estimator

     estimator = Estimator()
     estimator_qnn = EstimatorQNN(
         circuit=qc1,
         observables=observable1,
         input_params=[params1[0]],
         weight_params=[params1[1]],
         estimator=estimator,
     )
     estimator_qnn
```

```
No gradient function provided, creating a gradient function. If your Estimator requires transpilation, please provide
```

```
[4]: <qiskit_machine_learning.neural_networks.estimator_qnn.EstimatorQNN at 0x7f7317cd6740>
```

SamplerQNN

- The SamplerQNN is instantiated in a similar way to the EstimatorQNN, but because it directly consumes samples from measuring the quantum circuit, it does not require a custom observable.
- Let's create a different quantum circuit for the SamplerQNN. In this case, we will have two input parameters and four trainable weights that parametrize a two-local circuit.

SamplerQNN

[5]:

```python
from qiskit.circuit import ParameterVector

inputs2 = ParameterVector("input", 2)
weights2 = ParameterVector("weight", 4)
print(f"input parameters: {[str(item) for item in inputs2.params]}")
print(f"weight parameters: {[str(item) for item in weights2.params]}")

qc2 = QuantumCircuit(2)
qc2.ry(inputs2[0], 0)
qc2.ry(inputs2[1], 1)
qc2.cx(0, 1)
qc2.ry(weights2[0], 0)
qc2.ry(weights2[1], 1)
qc2.cx(0, 1)
qc2.ry(weights2[2], 0)
qc2.ry(weights2[3], 1)

qc2.draw("mpl", style="clifford")
```

```
input parameters: ['input[0]', 'input[1]']
weight parameters: ['weight[0]', 'weight[1]', 'weight[2]', 'weight[3]']
```

[5]:

SamplerQNN

- **Similarly to the EstimatorQNN, we must specify inputs and weights when instantiating the SamplerQNN. In this case, the keyword arguments will be:**
  - Sampler
  - pass_manager
  - input_params
  - weight_params

```
[6]: from qiskit_machine_learning.neural_networks import SamplerQNN
     from qiskit.primitives import StatevectorSampler as Sampler

     sampler = Sampler()
     sampler_qnn = SamplerQNN(circuit=qc2, input_params=inputs2, weight_params=weights2, sampler=sampler)
     sampler_qnn

     No gradient function provided, creating a gradient function. If your Sampler requires transpilation, please provide a

[6]: <qiskit_machine_learning.neural_networks.sampler_qnn.SamplerQNN at 0x7f730236b5b0>
```

How to Run a Forward Pass

EstimatorQNN Example

```
[7]: estimator_qnn_input = algorithm_globals.random.random(estimator_qnn.num_inputs)
     estimator_qnn_weights = algorithm_globals.random.random(estimator_qnn.num_weights)
```

```
[8]: print(
         f"Number of input features for EstimatorQNN: {estimator_qnn.num_inputs} \nInput: {estimator_qnn_input}"
     )
     print(
         f"Number of trainable weights for EstimatorQNN: {estimator_qnn.num_weights} \nWeights: {estimator_qnn_weights}"
     )
```

```
Number of input features for EstimatorQNN: 1
Input: [0.77395605]
Number of trainable weights for EstimatorQNN: 1
Weights: [0.43887844]
```

How to Run a Forward Pass

EstimatorQNN Example

Non-batched Forward Pass

```
[11]: estimator_qnn_forward = estimator_qnn.forward(estimator_qnn_input, estimator_qnn_weights)

print(
    f"Forward pass result for EstimatorQNN: {estimator_qnn_forward}. \nShape: {estimator_qnn_forward.shape}"
)
```

```
Forward pass result for EstimatorQNN: [[0.28127517]].
Shape: (1, 1)
```

Batched Forward Pass

```
[13]: estimator_qnn_forward_batched = estimator_qnn.forward(
    [estimator_qnn_input, estimator_qnn_input], estimator_qnn_weights
)

print(
    f"Forward pass result for EstimatorQNN: {estimator_qnn_forward_batched}.  \nShape: {estimator_qnn_forward_batched.
)
```

```
Forward pass result for EstimatorQNN: [[0.28503768]
 [0.28725616]].
Shape: (2, 1)
```

How to Run a Forward Pass

SamplerQNN Example

```
[9]: sampler_qnn_input = algorithm_globals.random.random(sampler_qnn.num_inputs)
     sampler_qnn_weights = algorithm_globals.random.random(sampler_qnn.num_weights)
```

```
[10]: print(
          f"Number of input features for SamplerQNN: {sampler_qnn.num_inputs} \nInput: {sampler_qnn_input}"
      )
      print(
          f"Number of trainable weights for SamplerQNN: {sampler_qnn.num_weights} \nWeights: {sampler_qnn_weights}"
      )
```

```
Number of input features for SamplerQNN: 2
Input: [0.85859792 0.69736803]
Number of trainable weights for SamplerQNN: 4
Weights: [0.09417735 0.97562235 0.7611397  0.78606431]
```

How to Run a Forward Pass

SamplerQNN Example

Non-batched Forward Pass

```
[12]: sampler_qnn_forward = sampler_qnn.forward(sampler_qnn_input, sampler_qnn_weights)

      print(
          f"Forward pass result for SamplerQNN: {sampler_qnn_forward}.  \nShape: {sampler_qnn_forward.shape}"
      )
```

```
Forward pass result for SamplerQNN: [[0.01171875 0.24316406 0.55175781 0.19335938]].
Shape: (1, 4)
```

Batched Forward Pass

```
[14]: sampler_qnn_forward_batched = sampler_qnn.forward(
          [sampler_qnn_input, sampler_qnn_input], sampler_qnn_weights
      )

      print(
          f"Forward pass result for SamplerQNN: {sampler_qnn_forward_batched}.  \nShape: {sampler_qnn_forward_batched.shape}
      )
```

```
Forward pass result for SamplerQNN: [[0.01171875 0.22949219 0.54003906 0.21875    ]
 [0.01855469 0.265625   0.515625   0.20019531]].
Shape: (2, 4)
```

How to Run a Backward Pass

Backward Pass without Input Gradients

EstimatorQNN

```
[15]: estimator_qnn_input_grad, estimator_qnn_weight_grad = estimator_qnn.backward(
          estimator_qnn_input, estimator_qnn_weights
      )

      print(
          f"Input gradients for EstimatorQNN: {estimator_qnn_input_grad}.  \nShape: {estimator_qnn_input_grad}"
      )
      print(
          f"Weight gradients for EstimatorQNN: {estimator_qnn_weight_grad}.  \nShape: {estimator_qnn_weight_grad.shape}"
      )

      Input gradients for EstimatorQNN: None.
      Shape: None
      Weight gradients for EstimatorQNN: [[[0.63272767]]].
      Shape: (1, 1, 1)
```

How to Run a Backward Pass

Backward Pass without Input Gradients

☐SamplerQNN

```
[16]: sampler_qnn_input_grad, sampler_qnn_weight_grad = sampler_qnn.backward(
          sampler_qnn_input, sampler_qnn_weights
      )

      print(
          f"Input gradients for SamplerQNN: {sampler_qnn_input_grad}.  \nShape: {sampler_qnn_input_grad}"
      )
      print(
          f"Weight gradients for SamplerQNN: {sampler_qnn_weight_grad}.  \nShape: {sampler_qnn_weight_grad.shape}"
      )
```

```
Input gradients for SamplerQNN: None.
Shape: None
Weight gradients for SamplerQNN: [[[ 0.00390625 -0.12451172 -0.06640625 -0.09277344]
  [ 0.21533203 -0.08007812  0.06689453 -0.22705078]
  [-0.48974609  0.32226562 -0.31542969  0.09375   ]
  [ 0.27050781 -0.11767578  0.31494141  0.22607422]]].
Shape: (1, 4, 4)
```

How to Run a Backward Pass

Backward Pass with Input Gradients

```
[17]: estimator_qnn.input_gradients = True
      sampler_qnn.input_gradients = True
```

EstimatorQNN

```
[18]: estimator_qnn_input_grad, estimator_qnn_weight_grad = estimator_qnn.backward(
          estimator_qnn_input, estimator_qnn_weights
      )

      print(
          f"Input gradients for EstimatorQNN: {estimator_qnn_input_grad}.  \nShape: {estimator_qnn_input_grad.shape}"
      )
      print(
          f"Weight gradients for EstimatorQNN: {estimator_qnn_weight_grad}.  \nShape: {estimator_qnn_weight_grad.shape}"
      )
```

```
Input gradients for EstimatorQNN: [[[0.3038852]]].
Shape: (1, 1, 1)
Weight gradients for EstimatorQNN: [[[0.63272767]]].
Shape: (1, 1, 1)
```

SJTU

How to Run a Backward Pass

Backward Pass with Input Gradients

SamplerQNN

```
[19]: sampler_qnn_input_grad, sampler_qnn_weight_grad = sampler_qnn.backward(
          sampler_qnn_input, sampler_qnn_weights
      )

      print(
          f"Input gradients for SamplerQNN: {sampler_qnn_input_grad}.  \nShape: {sampler_qnn_input_grad.shape}"
      )
      print(
          f"Weight gradients for SamplerQNN: {sampler_qnn_weight_grad}.  \nShape: {sampler_qnn_weight_grad.shape}"
      )
```

```
Input gradients for SamplerQNN: [[[-0.05664062 -0.10107422]
  [ 0.38330078 -0.19335938]
  [-0.34375     0.07861328]
  [ 0.01708984  0.21582031]]].
Shape: (1, 4, 2)
Weight gradients for SamplerQNN: [[[ 0.00732422 -0.11376953 -0.07080078 -0.08886719]
  [ 0.21972656 -0.08496094  0.05419922 -0.23193359]
  [-0.48828125  0.32128906 -0.31787109  0.10205078]
  [ 0.26123047 -0.12255859  0.33447266  0.21875   ]]].
Shape: (1, 4, 4)
```

Advanced Functionality

EstimatorQNN with Multiple Observables

```
[20]: observable2 = SparsePauliOp.from_list([("Z" * qc1.num_qubits, 1)])

estimator_qnn2 = EstimatorQNN(
    circuit=qc1,
    observables=[observable1, observable2],
    input_params=[params1[0]],
    weight_params=[params1[1]],
    estimator=estimator,
)
```

```
No gradient function provided, creating a gradient function. If your Estimator requires transpilation, please provide
```

```
[21]: estimator_qnn_forward2 = estimator_qnn2.forward(estimator_qnn_input, estimator_qnn_weights)
estimator_qnn_input_grad2, estimator_qnn_weight_grad2 = estimator_qnn2.backward(
    estimator_qnn_input, estimator_qnn_weights
)

print(f"Forward output for EstimatorQNN1: {estimator_qnn_forward.shape}")
print(f"Forward output for EstimatorQNN2: {estimator_qnn_forward2.shape}")
print(f"Backward output for EstimatorQNN1: {estimator_qnn_weight_grad.shape}")
print(f"Backward output for EstimatorQNN2: {estimator_qnn_weight_grad2.shape}")
```

```
Forward output for EstimatorQNN1: (1, 1)
Forward output for EstimatorQNN2: (1, 2)
Backward output for EstimatorQNN1: (1, 1, 1)
Backward output for EstimatorQNN2: (1, 2, 1)
```

Advanced Functionality

SamplerQNN with custom interpret

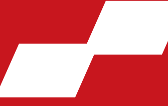
```
[22]: parity = lambda x: "{:b}".format(x).count("1") % 2
      output_shape = 2  # parity = 0, 1

      sampler_qnn2 = SamplerQNN(
          circuit=qc2,
          input_params=inputs2,
          weight_params=weights2,
          interpret=parity,
          output_shape=output_shape,
          sampler=sampler,
      )
```

```
No gradient function provided, creating a gradient function. If your Sampler requires transpilation, please provide a
```

```
[23]: sampler_qnn_forward2 = sampler_qnn2.forward(sampler_qnn_input, sampler_qnn_weights)
      sampler_qnn_input_grad2, sampler_qnn_weight_grad2 = sampler_qnn2.backward(
          sampler_qnn_input, sampler_qnn_weights
      )

      print(f"Forward output for SamplerQNN1: {sampler_qnn_forward.shape}")
      print(f"Forward output for SamplerQNN2: {sampler_qnn_forward2.shape}")
      print(f"Backward output for SamplerQNN1: {sampler_qnn_weight_grad.shape}")
      print(f"Backward output for SamplerQNN2: {sampler_qnn_weight_grad2.shape}")
```

```
Forward output for SamplerQNN1: (1, 4)
Forward output for SamplerQNN2: (1, 2)
Backward output for SamplerQNN1: (1, 4, 4)
Backward output for SamplerQNN2: (1, 2, 4)
```

Neural Network Classifier & Regressor



- **In this tutorial we show how the NeuralNetworkClassifier and NeuralNetworkRegressor are used. Both take as an input a (Quantum) NeuralNetwork and leverage it in a specific context. In both cases we also provide a pre-configured variant for convenience, the Variational Quantum Classifier (VQC) and Variational Quantum Regressor (VQR). The tutorial is structured as follows:**
  - Classification
    - Classification with an EstimatorQNN
    - Classification with a SamplerQNN
    - Variational Quantum Classifier (VQC)
  - Regression
    - Regression with an EstimatorQNN
    - Variational Quantum Regressor (VQR)

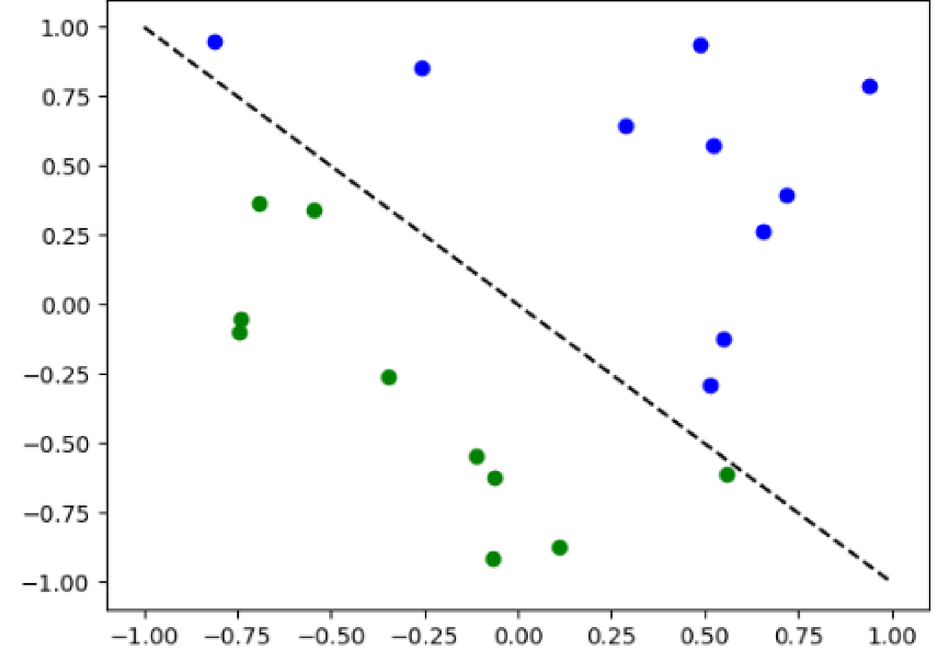
https://qiskit-community.github.io/qiskit-machine-learning/tutorials/02_neural_network_classifier_and_regressor.html

Neural Network Classifier & Regressor

```
[1]: import matplotlib.pyplot as plt
     import numpy as np
     from IPython.display import clear_output
     from qiskit import QuantumCircuit
     from qiskit.circuit import Parameter
     from qiskit.circuit.library import RealAmplitudes, ZZFeatureMap
     from qiskit_machine_learning.optimizers import COBYLA, L_BFGS_B
     from qiskit_machine_learning.utils import algorithm_globals

     from qiskit_machine_learning.algorithms.classifiers import NeuralNetworkClassifier, VQC
     from qiskit_machine_learning.algorithms.regressors import NeuralNetworkRegressor, VQR
     from qiskit_machine_learning.neural_networks import SamplerQNN, EstimatorQNN
     from qiskit_machine_learning.circuit.library import QNNCircuit

     algorithm_globals.random_seed = 42
```

Classification

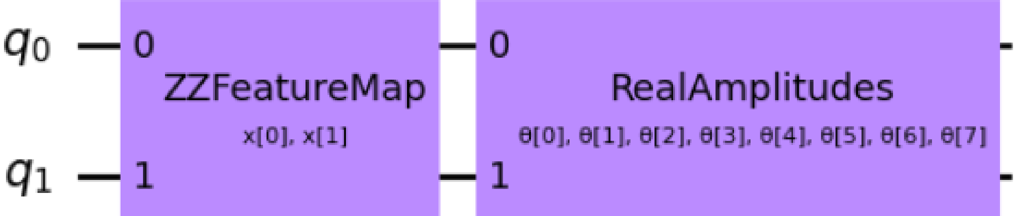
```
[2]: num_inputs = 2
num_samples = 20
X = 2 * algorithm_globals.random.random([num_samples, num_inputs]) - 1
y01 = 1 * (np.sum(X, axis=1) >= 0)  # in { 0,  1}
y = 2 * y01 - 1  # in {-1, +1}
y_one_hot = np.zeros((num_samples, 2))
for i in range(num_samples):
    y_one_hot[i, y01[i]] = 1

for x, y_target in zip(X, y):
    if y_target == 1:
        plt.plot(x[0], x[1], "bo")
    else:
        plt.plot(x[0], x[1], "go")
plt.plot([-1, 1], [1, -1], "--", color="black")
plt.show()
```



Classification with an EstimatorQNN

```
[3]: # construct QNN with the QNNCircuit's default ZZFeatureMap feature map and RealAmplitudes ansatz.
     qc = QNNCircuit(num_qubits=2)
     qc.draw("mpl", style="clifford")
```

[3]:



Create a quantum neural network. As we are performing a local statevector simulation, we will set the `estimator` parameter from `qiskit.primitives.StatevectorEstimator`.

```
[4]: from qiskit.primitives import StatevectorEstimator as Estimator

     estimator = Estimator()
     estimator_qnn = EstimatorQNN(circuit=qc, estimator=estimator)
```

```
No gradient function provided, creating a gradient function. If your Estimator requires transpilation, please provide
```

```
[5]: # QNN maps inputs to [-1, +1]
     estimator_qnn.forward(X[0, :], algorithm_globals.random.random(estimator_qnn.num_weights))
```

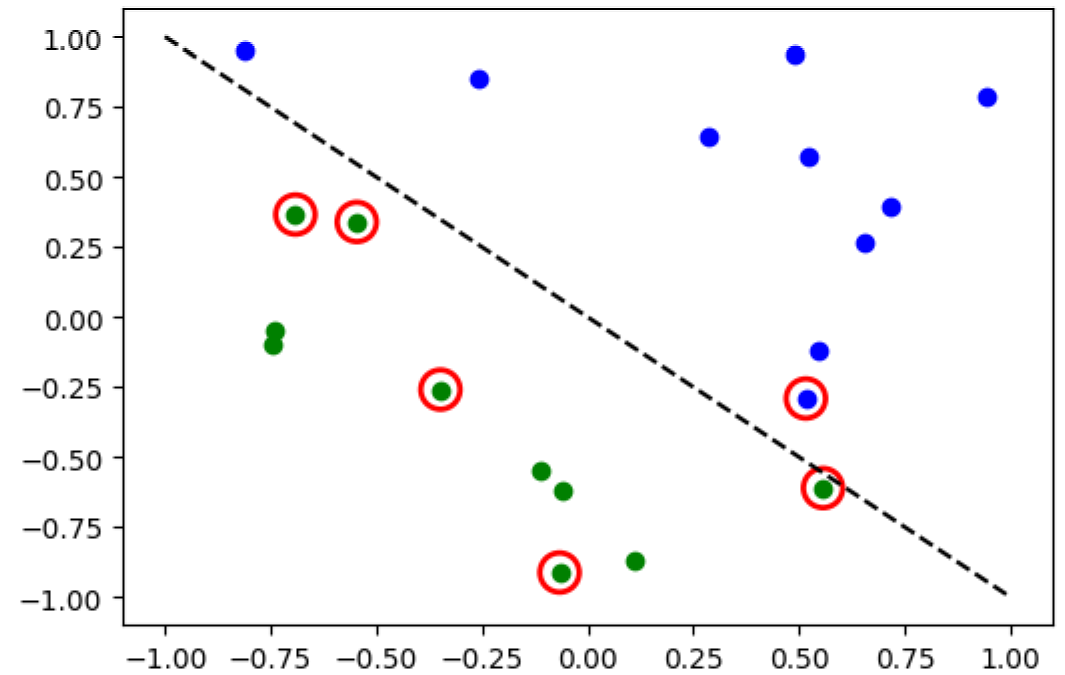
```
[5]: array([[0.23238601]])
```

[6]:
```python
# callback function that draws a live plot when the .fit() method is called
def callback_graph(weights, obj_func_eval):
    clear_output(wait=True)
    objective_func_vals.append(obj_func_eval)
    plt.title("Objective function value against iteration")
    plt.xlabel("Iteration")
    plt.ylabel("Objective function value")
    plt.plot(range(len(objective_func_vals)), objective_func_vals)
    plt.show()
```

[7]:
```python
# construct neural network classifier
estimator_classifier = NeuralNetworkClassifier(
    estimator_qnn, optimizer=COBYLA(maxiter=60), callback=callback_graph
)
```

[8]:
```python
# create empty array for callback to store evaluations of the objective function
objective_func_vals = []
plt.rcParams["figure.figsize"] = (12, 6)

# fit classifier to data
estimator_classifier.fit(X, y)

# return to default figsize
plt.rcParams["figure.figsize"] = (6, 4)

# score classifier
estimator_classifier.score(X, y)
```

```
[9]: # evaluate data points
y_predict = estimator_classifier.predict(X)

# plot results
# red == wrongly classified
for x, y_target, y_p in zip(X, y, y_predict):
    if y_target == 1:
        plt.plot(x[0], x[1], "bo")
    else:
        plt.plot(x[0], x[1], "go")
    if y_target != y_p:
        plt.scatter(x[0], x[1], s=200, facecolors="none", edgecolors="r", linewidths=2)
plt.plot([-1, 1], [1, -1], "--", color="black")
plt.show()
```



```
[11]: # construct a quantum circuit from the default ZZFeatureMap feature map and a customized RealAmplitudes ansatz
qc = QNNCircuit(ansatz=RealAmplitudes(num_inputs, reps=1))
qc.draw("mpl", style="clifford")
```

[11]:

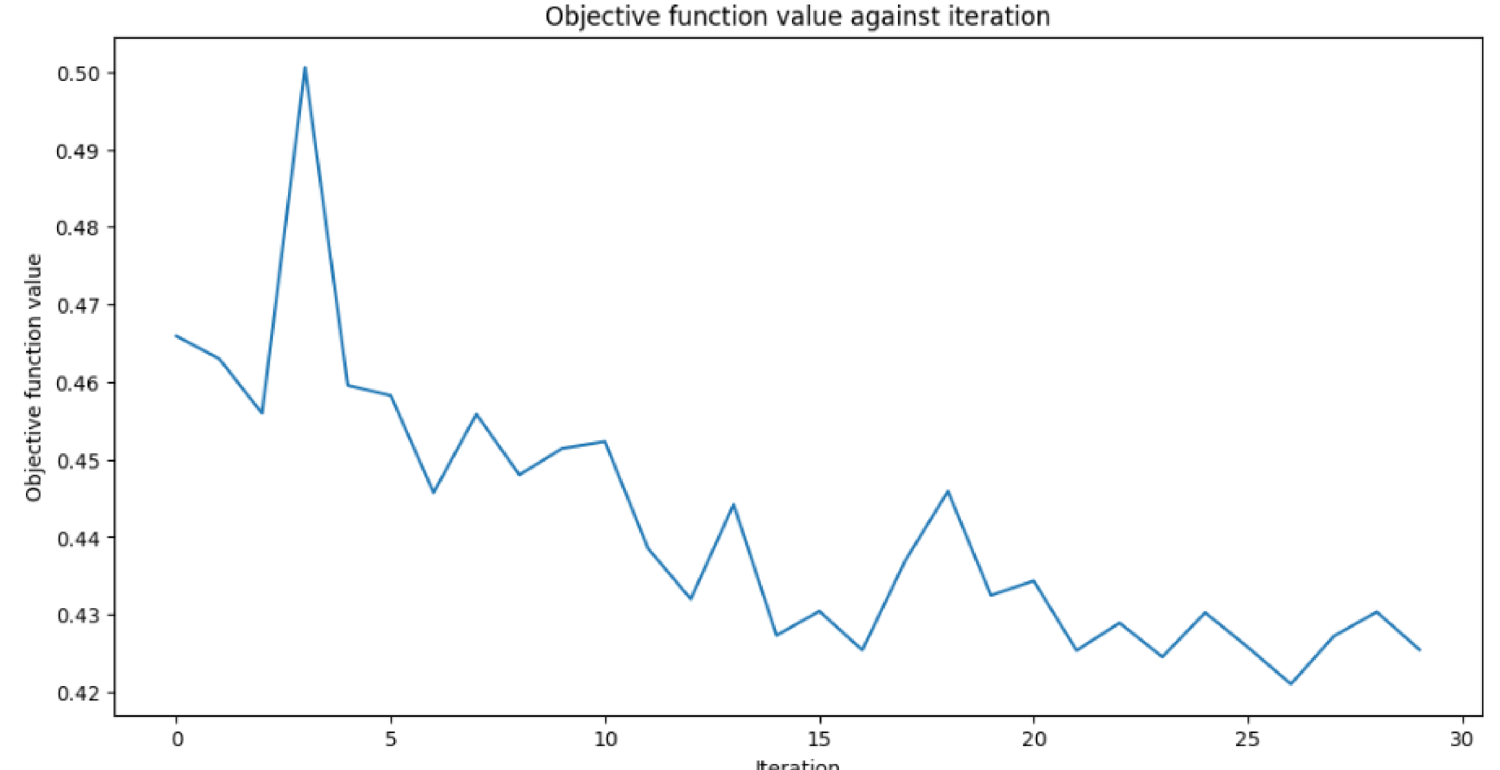

```
[12]: # parity maps bitstrings to 0 or 1
def parity(x):
    return "{:b}".format(x).count("1") % 2


output_shape = 2  # corresponds to the number of classes, possible outcomes of the (parity) mapping.
```

```
[13]: from qiskit.primitives import StatevectorSampler as Sampler

sampler = Sampler()
# construct QNN
sampler_qnn = SamplerQNN(
    circuit=qc,
    interpret=parity,
    output_shape=output_shape,
    sampler=sampler,
)
```

```
No gradient function provided, creating a gradient function. If your Sampler requires transpilation, please provide a
```

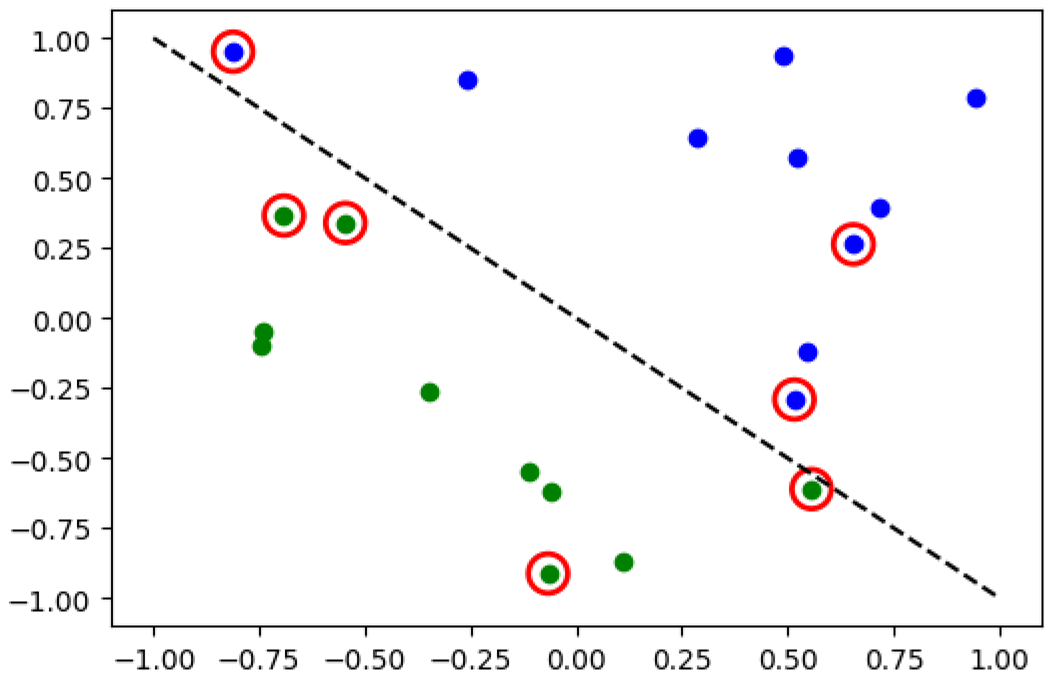
```
[14]: # construct classifier
sampler_classifier = NeuralNetworkClassifier(
    neural_network=sampler_qnn, optimizer=COBYLA(maxiter=30), callback=callback_graph
)
```

```
[15]: # create empty array for callback to store evaluations of the objective function
objective_func_vals = []
plt.rcParams["figure.figsize"] = (12, 6)

# fit classifier to data
sampler_classifier.fit(X, y01)

# return to default figsize
plt.rcParams["figure.figsize"] = (6, 4)

# score classifier
sampler_classifier.score(X, y01)
```



SJTU

[16]:
```python
# evaluate data points
y_predict = sampler_classifier.predict(X)

# plot results
# red == wrongly classified
for x, y_target, y_p in zip(X, y01, y_predict):
    if y_target == 1:
        plt.plot(x[0], x[1], "bo")
    else:
        plt.plot(x[0], x[1], "go")
    if y_target != y_p:
        plt.scatter(x[0], x[1], s=200, facecolors="none", edgecolors="r", linewidths=2)
plt.plot([-1, 1], [1, -1], "--", color="black")
plt.show()
```



SJTU

Variational Quantum Classifier (VQC)

```
[18]: # construct feature map, ansatz, and optimizer
feature_map = ZZFeatureMap(num_inputs)
ansatz = RealAmplitudes(num_inputs, reps=1)

# construct variational quantum classifier
vqc = VQC(
    feature_map=feature_map,
    ansatz=ansatz,
    loss="cross_entropy",
    optimizer=COBYLA(maxiter=30),
    callback=callback_graph,
    sampler=sampler,
)
```

```
No gradient function provided, creating a gradient function. If your Sampler requires transpilation, please provide a
```

```
[19]: # create empty array for callback to store evaluations of the objective function
objective_func_vals = []
plt.rcParams["figure.figsize"] = (12, 6)

# fit classifier to data
vqc.fit(X, y_one_hot)

# return to default figsize
plt.rcParams["figure.figsize"] = (6, 4)

# score classifier
vqc.score(X, y_one_hot)
```

```
[19]: # create empty array for callback to store evaluations of the objec
objective_func_vals = []
plt.rcParams["figure.figsize"] = (12, 6)

# fit classifier to data
vqc.fit(X, y_one_hot)

# return to default figsize
plt.rcParams["figure.figsize"] = (6, 4)

# score classifier
vqc.score(X, y_one_hot)
```

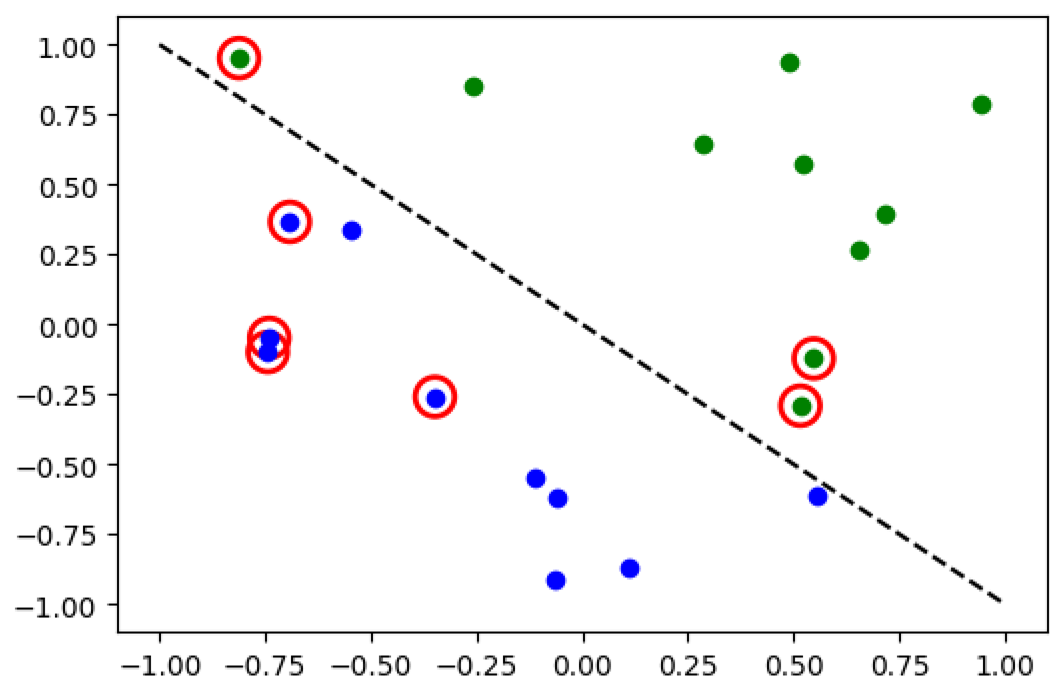
Back to top

Objective function value against iteration


```
[19]: 0.65
```

SJTU

```python
[20]: # evaluate data points
y_predict = vqc.predict(X)

# plot results
# red == wrongly classified
for x, y_target, y_p in zip(X, y_one_hot, y_predict):
    if y_target[0] == 1:
        plt.plot(x[0], x[1], "bo")
    else:
        plt.plot(x[0], x[1], "go")
    if not np.all(y_target == y_p):
        plt.scatter(x[0], x[1], s=200, facecolors="none", edgecolors="r", linewidths=2)
plt.plot([-1, 1], [1, -1], "--", color="black")
plt.show()
```



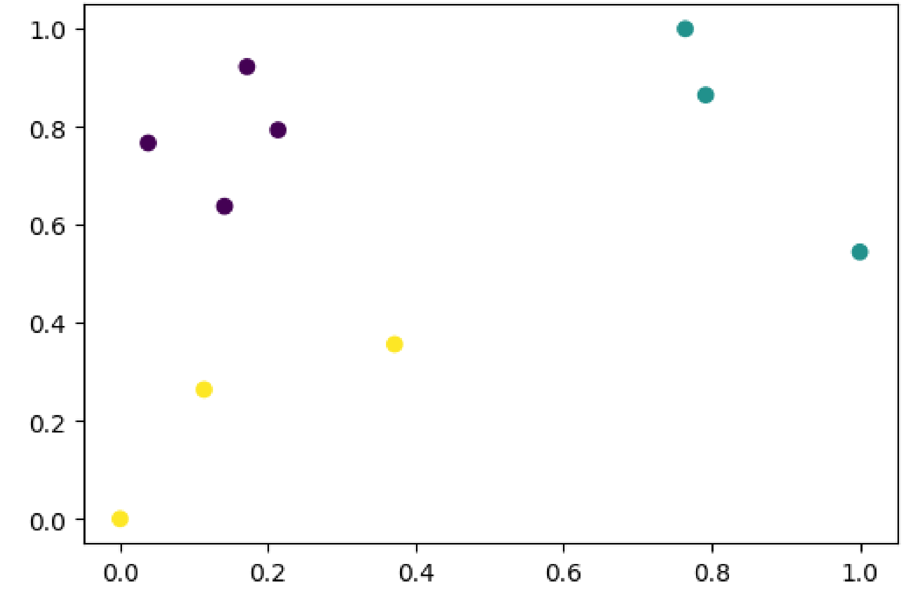
Multiple classes with VQC

[21]:
```python
from sklearn.datasets import make_classification
from sklearn.preprocessing import MinMaxScaler

X, y = make_classification(
    n_samples=10,
    n_features=2,
    n_classes=3,
    n_redundant=0,
    n_clusters_per_class=1,
    class_sep=2.0,
    random_state=algorithm_globals.random_seed,
)
X = MinMaxScaler().fit_transform(X)
```

Let's see how our dataset looks like.

[22]:
```python
plt.scatter(X[:, 0], X[:, 1], c=y)
```

[22]: <matplotlib.collections.PathCollection at 0x7f8bd8786ad0>



```
[24]: vqc = VQC(
          num_qubits=2,
          optimizer=COBYLA(maxiter=30),
          callback=callback_graph,
          sampler=sampler,
      )
```

```
No gradient function provided, creating a gradient function. If your Sampler requires transpilation, please provide a
```

```
[25]: # create empty array for callback to store evaluations of the objective function
      objective_func_vals = []
      plt.rcParams["figure.figsize"] = (12, 6)

      # fit classifier to data
      vqc.fit(X, y_cat)

      # return to default figsize
      plt.rcParams["figure.figsize"] = (6, 4)

      # score classifier
      vqc.score(X, y_cat)
```

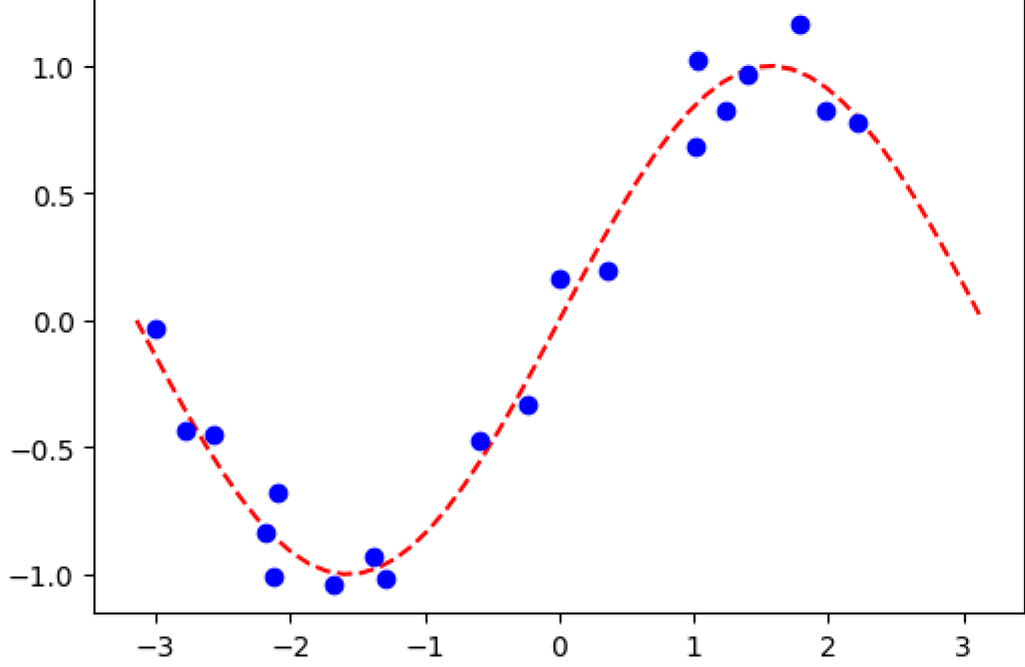
Objective function value against iteration

(Chart: y-axis "Objective function value", ticks 10.40, 10.45, 10.50, 10.55, 10.60, 10.65, 10.70, 10.75, 10.80; x-axis "Iteration", ticks 0, 5, 10, 15, 20, 25, 30)

```
[25]: 0.6
```

Regression

```
[27]: num_samples = 20
      eps = 0.2
      lb, ub = -np.pi, np.pi
      X_ = np.linspace(lb, ub, num=50).reshape(50, 1)
      f = lambda x: np.sin(x)

      X = (ub - lb) * algorithm_globals.random.random([num_samples, 1]) + lb
      y = f(X[:, 0]) + eps * (2 * algorithm_globals.random.random(num_samples) - 1)

      plt.plot(X_, f(X_), "r--")
      plt.plot(X, y, "bo")
      plt.show()
```



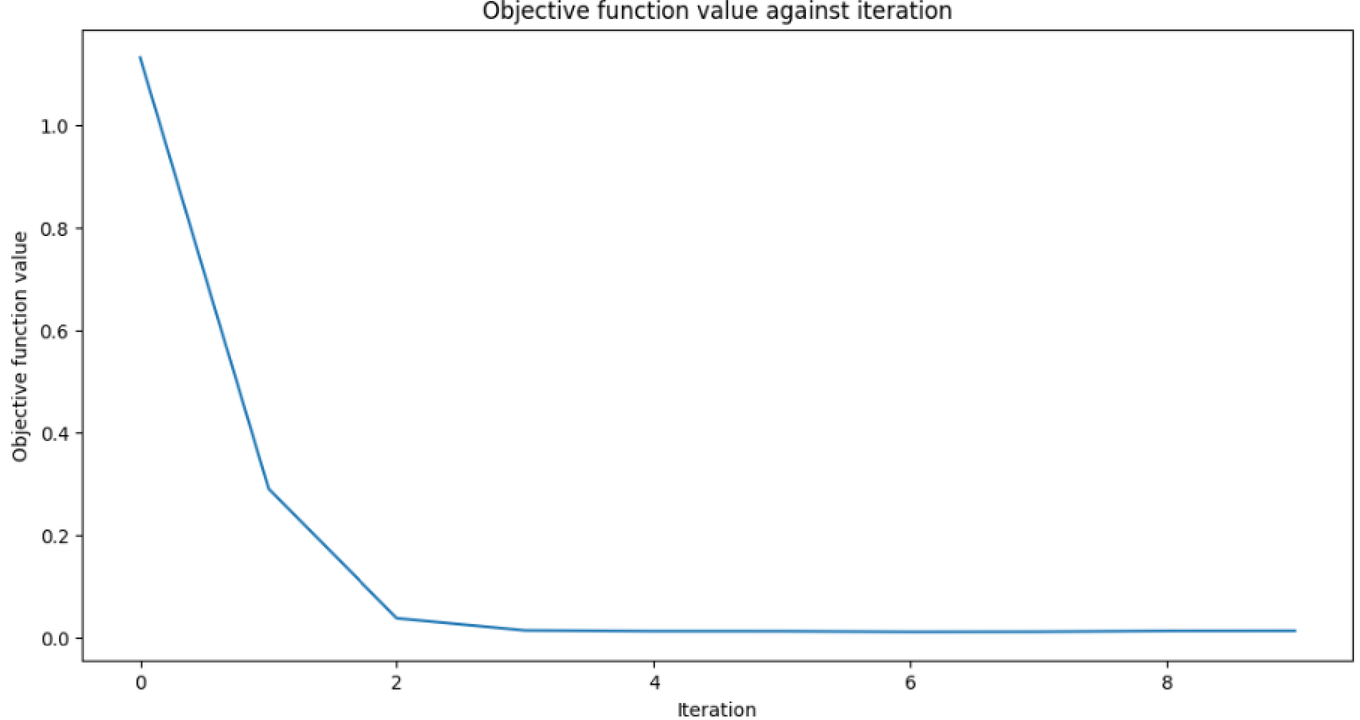
Regression with an EstimatorQNN

[28]:
```
# construct simple feature map
param_x = Parameter("x")
feature_map = QuantumCircuit(1, name="fm")
feature_map.ry(param_x, 0)

# construct simple ansatz
param_y = Parameter("y")
ansatz = QuantumCircuit(1, name="vf")
ansatz.ry(param_y, 0)

# construct a circuit
qc = QNNCircuit(feature_map=feature_map, ansatz=ansatz)

# construct QNN
regression_estimator_qnn = EstimatorQNN(circuit=qc, estimator=estimator)
```

```
No gradient function provided, creating a gradient function. If your Estimator requires transpilation, please provide
```

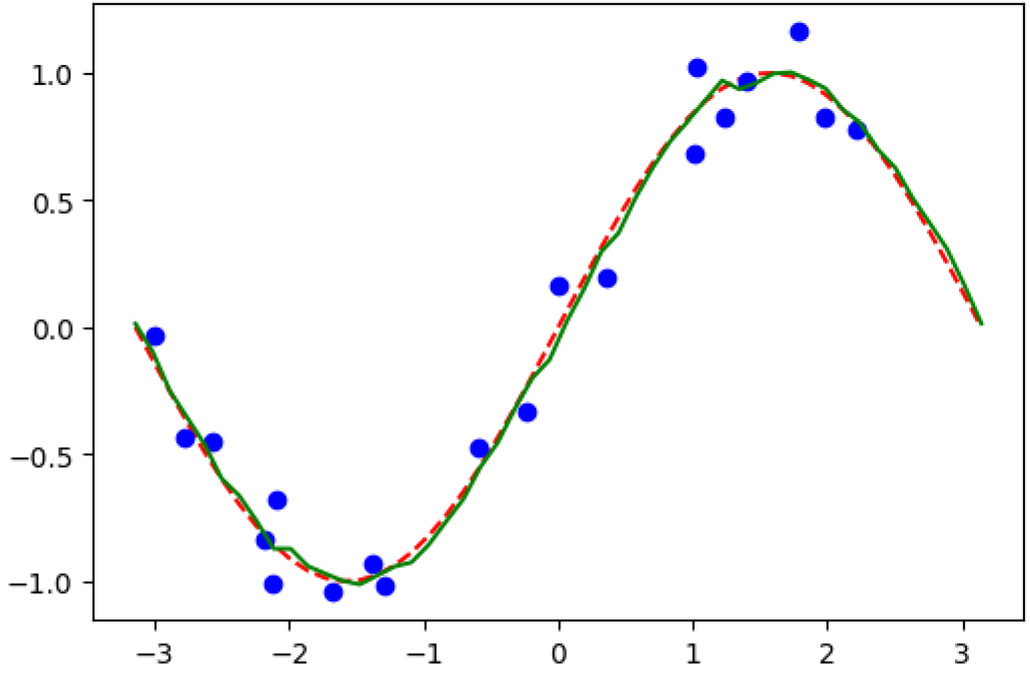
[29]:
```
# construct the regressor from the neural network
regressor = NeuralNetworkRegressor(
    neural_network=regression_estimator_qnn,
    loss="squared_error",
    optimizer=L_BFGS_B(maxiter=5),
    callback=callback_graph,
)
```

```
[30]: # create empty array for callback to store evaluations of the objective function
objective_func_vals = []
plt.rcParams["figure.figsize"] = (12, 6)

# fit to data
regressor.fit(X, y)

# return to default figsize
plt.rcParams["figure.figsize"] = (6, 4)

# score the result
regressor.score(X, y)
```




```
[30]: 0.9746977891041729
```

[31]:
```
# plot target function
plt.plot(X_, f(X_), "r--")

# plot data
plt.plot(X, y, "bo")

# plot fitted line
y_ = regressor.predict(X_)
plt.plot(X_, y_, "g-")
plt.show()
```



Regression with the Variational Quantum Regressor (VQR)

```
[33]: vqr = VQR(
          feature_map=feature_map,
          ansatz=ansatz,
          optimizer=L_BFGS_B(maxiter=5),
          callback=callback_graph,
          estimator=estimator,
      )
```

```
No gradient function provided, creating a gradient function. If your Estimator requires transpilation, please provide
```

```
[34]: # create empty array for callback to store evaluations of the objective function
      objective_func_vals = []
      plt.rcParams["figure.figsize"] = (12, 6)

      # fit regressor
      vqr.fit(X, y)

      # return to default figsize
      plt.rcParams["figure.figsize"] = (6, 4)

      # score result
      vqr.score(X, y)
```

Objective function value against iteration

Objective function value

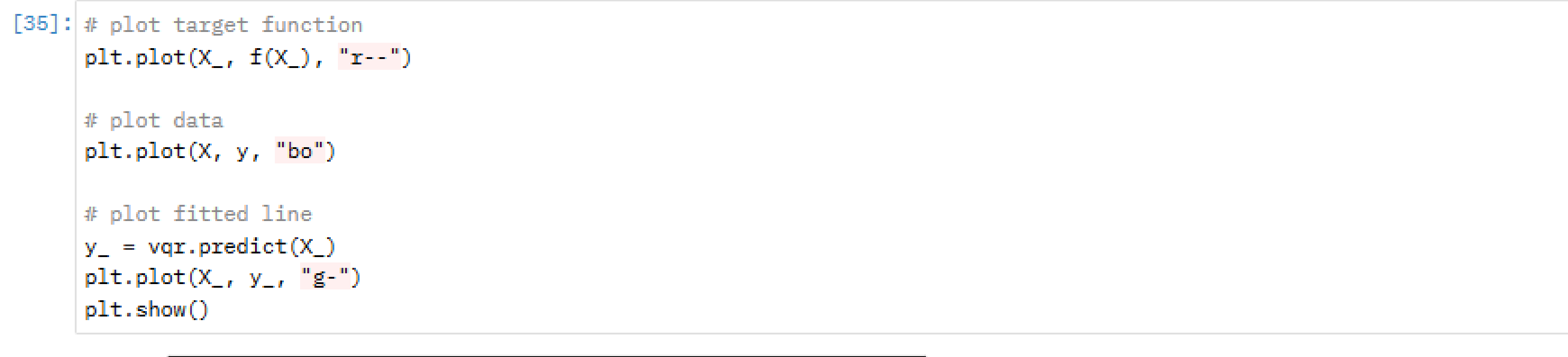
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0

0 1 2 3 4 5

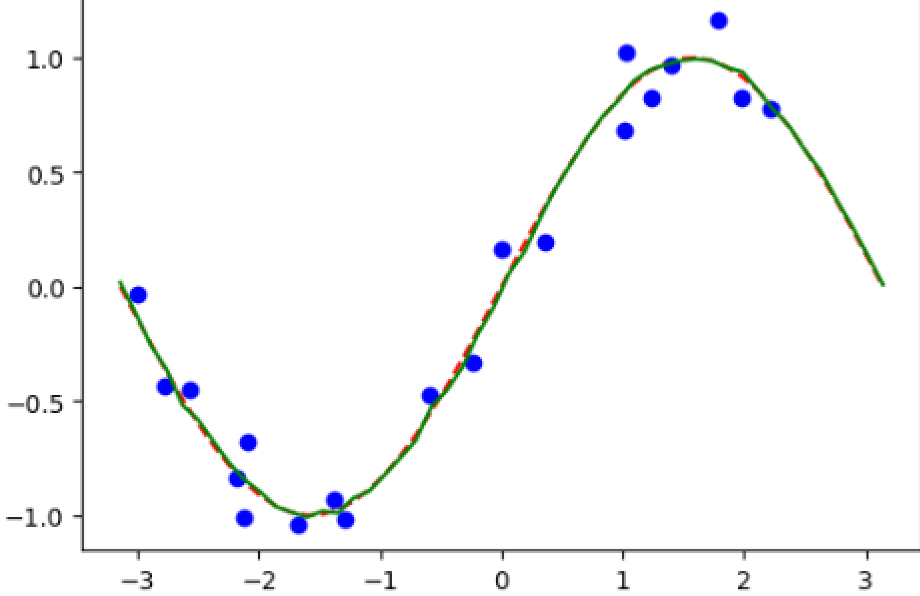
Iteration

[34]: 0.9764887827914851

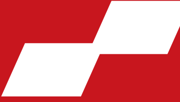
SJTU



```
[35]: # plot target function
plt.plot(X_, f(X_), "r--")

# plot data
plt.plot(X, y, "bo")

# plot fitted line
y_ = vqr.predict(X_)
plt.plot(X_, y_, "g-")
plt.show()
```



Training a Quantum Model on a Real Dataset



Exploratory Data Analysis

```
[1]: from sklearn.datasets import load_iris

     iris_data = load_iris()
```

```
[3]: features = iris_data.data
     labels = iris_data.target
```

```
[4]: from sklearn.preprocessing import MinMaxScaler

     features = MinMaxScaler().fit_transform(features)
```

- There are 150 samples (instances) in the dataset.
- There are four features (attributes) in each sample.
- There are three labels (classes) in the dataset.
- The dataset is perfectly balanced, as there are the same number of samples (50) in each class.

Training a Classical Machine Learning Model

```
[6]: from sklearn.model_selection import train_test_split
     from qiskit_machine_learning.utils import algorithm_globals

     algorithm_globals.random_seed = 123
     train_features, test_features, train_labels, test_labels = train_test_split(
         features, labels, train_size=0.8, random_state=algorithm_globals.random_seed
     )
```

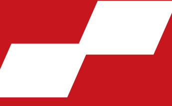
We train a classical Support Vector Classifier from scikit-learn. For the sake of simplicity, we don't tweak any parameters and rely on the default values.

```
[7]: from sklearn.svm import SVC

     svc = SVC()
     _ = svc.fit(train_features, train_labels)  # suppress printing the return value
```

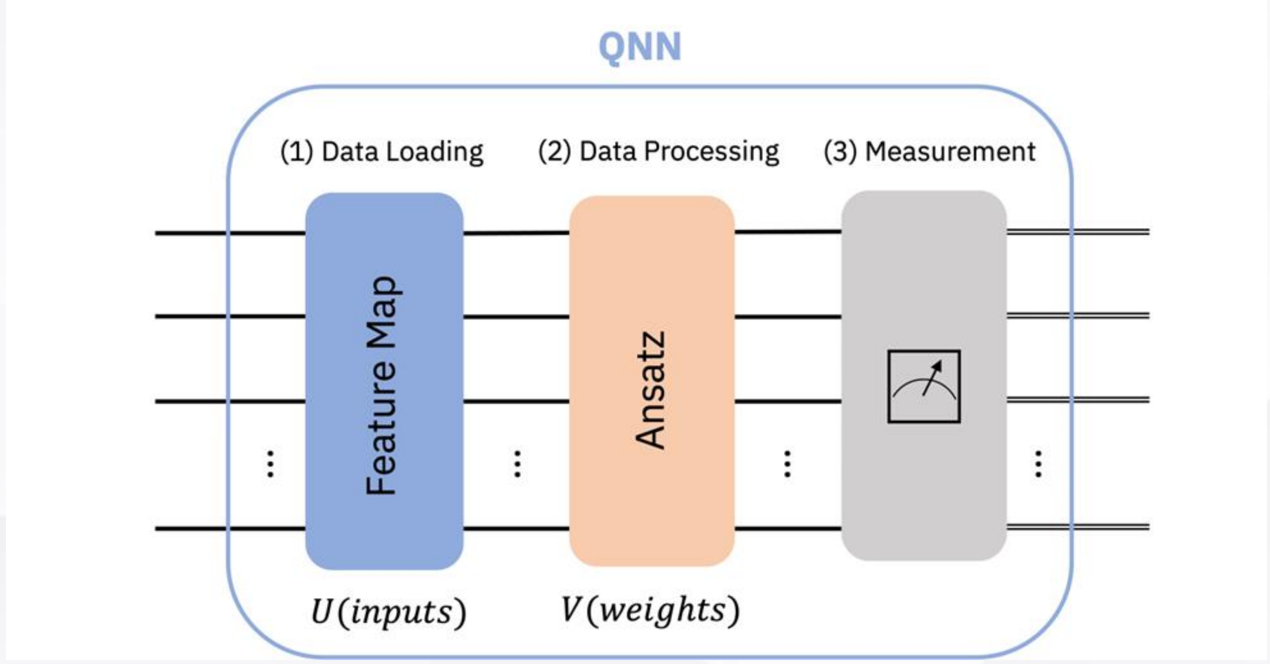
Now we check out how well our classical model performs. We will analyze the scores in the conclusion section.

```
[8]: train_score_c4 = svc.score(train_features, train_labels)
     test_score_c4 = svc.score(test_features, test_labels)

     print(f"Classical SVC on the training dataset: {train_score_c4:.2f}")
     print(f"Classical SVC on the test dataset:     {test_score_c4:.2f}")

     Classical SVC on the training dataset: 0.99
     Classical SVC on the test dataset:     0.97
```

Training a Quantum Machine Learning Model

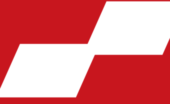


- Our data is classical, meaning it consists of a set of bits, not qubits. We need a way to encode the data as qubits.
- Once the data is loaded, we must immediately apply a parameterized quantum circuit.



SJTU

Training a Quantum Machine Learning Model



Data loading

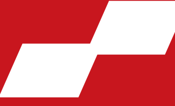
[9]:
```
from qiskit.circuit.library import ZZFeatureMap

num_features = features.shape[1]

feature_map = ZZFeatureMap(feature_dimension=num_features, reps=1)
feature_map.decompose().draw(output="mpl", style="clifford", fold=20)
```

[9]:

q0
q1
q2
q3
H
P 2.0*x[0]
P 2.0*x[1]
P 2.0*x[2]
P 2.0*x[3]
P 2.0*(π − x[0])*(π − x[1])
P 2.0*(π − x[0])*(π − x[2])
P 2.0*(π − x[1])*(π − x[2])
P 2.0*(π − x[0])*(π − x[3])
P 2.0*(π − x[1])*(π − x[3])
P 2.0*(π − x[2])*(π − x[3])

SJTU

Training a Quantum Machine Learning Model



Ansatz

[10]:
```python
from qiskit.circuit.library import RealAmplitudes

ansatz = RealAmplitudes(num_qubits=num_features, reps=3)
ansatz.decompose().draw(output="mpl", style="clifford", fold=20)
```

[10]:

SJTU

Training a Quantum Machine Learning Model

Optimizer

```
[11]: from qiskit_machine_learning.optimizers import COBYLA

optimizer = COBYLA(maxiter=100)
```

Sampler

```
[12]: from qiskit.primitives import StatevectorSampler as Sampler

sampler = Sampler()
```

```
[13]: from matplotlib import pyplot as plt
from IPython.display import clear_output

objective_func_vals = []
plt.rcParams["figure.figsize"] = (12, 6)


def callback_graph(weights, obj_func_eval):
    clear_output(wait=True)
    objective_func_vals.append(obj_func_eval)
    plt.title("Objective function value against iteration")
    plt.xlabel("Iteration")
    plt.ylabel("Objective function value")
    plt.plot(range(len(objective_func_vals)), objective_func_vals)
    plt.show()
```

Training a Quantum Machine Learning Model

```
[14]: import time
      from qiskit_machine_learning.algorithms.classifiers import VQC

      vqc = VQC(
          sampler=sampler,
          feature_map=feature_map,
          ansatz=ansatz,
          optimizer=optimizer,
          callback=callback_graph,
      )

      # clear objective value history
      objective_func_vals = []

      start = time.time()
      vqc.fit(train_features, train_labels)
      elapsed = time.time() - start

      print(f"Training time: {round(elapsed)} seconds")
```

```
[15]: train_score_q4 = vqc.score(train_features, train_labels)
      test_score_q4 = vqc.score(test_features, test_labels)

      print(f"Quantum VQC on the training dataset: {train_score_q4:.2f}")
      print(f"Quantum VQC on the test dataset:     {test_score_q4:.2f}")
```

Training a Quantum Machine Learning Model

Objective function value against iteration


```
Training time: 103 seconds
```

```
Quantum VQC on the training dataset: 0.62
Quantum VQC on the test dataset:     0.53
```

Reducing the Number of Features

```
[16]: from sklearn.decomposition import PCA

      features = PCA(n_components=2).fit_transform(features)

      plt.rcParams["figure.figsize"] = (6, 6)
      sns.scatterplot(x=features[:, 0], y=features[:, 1], hue=labels, palette="tab10")
```

```
[16]: <Axes: >
```

Reducing the Number of Features

```
[16]: from sklearn.decomposition import PCA

      features = PCA(n_components=2).fit_transform(features)

      plt.rcParams["figure.figsize"] = (6, 6)
      sns.scatterplot(x=features[:, 0], y=features[:, 1], hue=labels, palette="tab10")
```

```
[16]: <Axes: >
```

Reducing the Number of Features

[18]:
```python
num_features = features.shape[1]

feature_map = ZZFeatureMap(feature_dimension=num_features, reps=1)
ansatz = RealAmplitudes(num_qubits=num_features, reps=3)
```

[19]:
```python
optimizer = COBYLA(maxiter=40)
```

[20]:
```python
vqc = VQC(
    sampler=sampler,
    feature_map=feature_map,
    ansatz=ansatz,
    optimizer=optimizer,
    callback=callback_graph,
)

# clear objective value history
objective_func_vals = []

# make the objective function plot look nicer.
plt.rcParams["figure.figsize"] = (12, 6)


start = time.time()
vqc.fit(train_features, train_labels)
elapsed = time.time() - start

print(f"Training time: {round(elapsed)} seconds")
```

Reducing the Number of Features

```
Training time: 29 seconds
```

```
[21]: train_score_q2_ra = vqc.score(train_features, train_labels)
      test_score_q2_ra = vqc.score(test_features, test_labels)

      print(f"Quantum VQC on the training dataset using RealAmplitudes: {train_score_q2_ra:.2f}")
      print(f"Quantum VQC on the test dataset using RealAmplitudes:     {test_score_q2_ra:.2f}")
```

```
Quantum VQC on the training dataset using RealAmplitudes: 0.49
Quantum VQC on the test dataset using RealAmplitudes:     0.33
```

SJTU

Reducing the Number of Features

[22]:
```python
from qiskit.circuit.library import EfficientSU2

ansatz = EfficientSU2(num_qubits=num_features, reps=3)
optimizer = COBYLA(maxiter=40)

vqc = VQC(
    sampler=sampler,
    feature_map=feature_map,
    ansatz=ansatz,
    optimizer=optimizer,
    callback=callback_graph,
)

# clear objective value history
objective_func_vals = []

start = time.time()
vqc.fit(train_features, train_labels)
elapsed = time.time() - start

print(f"Training time: {round(elapsed)} seconds")
```

[23]:
```python
train_score_q2_eff = vqc.score(train_features, train_labels)
test_score_q2_eff = vqc.score(test_features, test_labels)

print(f"Quantum VQC on the training dataset using EfficientSU2: {train_score_q2_eff:.2f}")
print(f"Quantum VQC on the test dataset using EfficientSU2:     {test_score_q2_eff:.2f}")
```

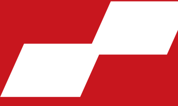
```
Quantum VQC on the training dataset using EfficientSU2: 0.65
Quantum VQC on the test dataset using EfficientSU2:     0.60
```

SJTU

Conclusion

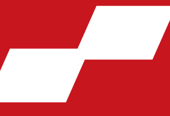
```
[24]: print(f"Model                             | Test Score | Train Score")
      print(f"SVC, 4 features                   | {train_score_c4:10.2f} | {test_score_c4:10.2f}")
      print(f"VQC, 4 features, RealAmplitudes | {train_score_q4:10.2f} | {test_score_q4:10.2f}")
      print(f"----------------------------------------------------------")
      print(f"SVC, 2 features                   | {train_score_c2:10.2f} | {test_score_c2:10.2f}")
      print(f"VQC, 2 features, RealAmplitudes | {train_score_q2_ra:10.2f} | {test_score_q2_ra:10.2f}")
      print(f"VQC, 2 features, EfficientSU2   | {train_score_q2_eff:10.2f} | {test_score_q2_eff:10.2f}")

      Model                             | Test Score | Train Score
      SVC, 4 features                   |       0.99 |       0.97
      VQC, 4 features, RealAmplitudes   |       0.62 |       0.53
      ----------------------------------------------------------
      SVC, 2 features                   |       0.97 |       0.90
      VQC, 2 features, RealAmplitudes   |       0.49 |       0.33
      VQC, 2 features, EfficientSU2     |       0.65 |       0.60
```

The Quantum Convolution Neural Network



- **Throughout this tutorial, we discuss a Quantum Convolutional Neural Network (QCNN). We implement such a QCNN on Qiskit by modeling both the convolutional layers and pooling layers using a quantum circuit. After building such a network, we train it to differentiate horizontal and vertical lines from a pixelated image. The following tutorial is thus divided accordingly;**
  - ☐ Differences between a QCNN and CCNN
  - ☐ Components of a QCNN
  - ☐ Data Generation
  - ☐ Building a QCNN
  - ☐ Training our QCNN
  - ☐ Testing our QCNN

Differences between a QCNN and CCNN



Classical Convolutional Neural Networks

- Classical Convolutional Neural Networks (CCNNs) are a subclass of artificial neural networks which have the ability to determine particular features and patterns of a given input.

C P C P FC

Input image

j

i

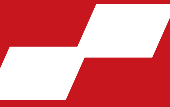
$(\ell-1)$ (ℓ) $(\ell+1)$

=

Cat Dog

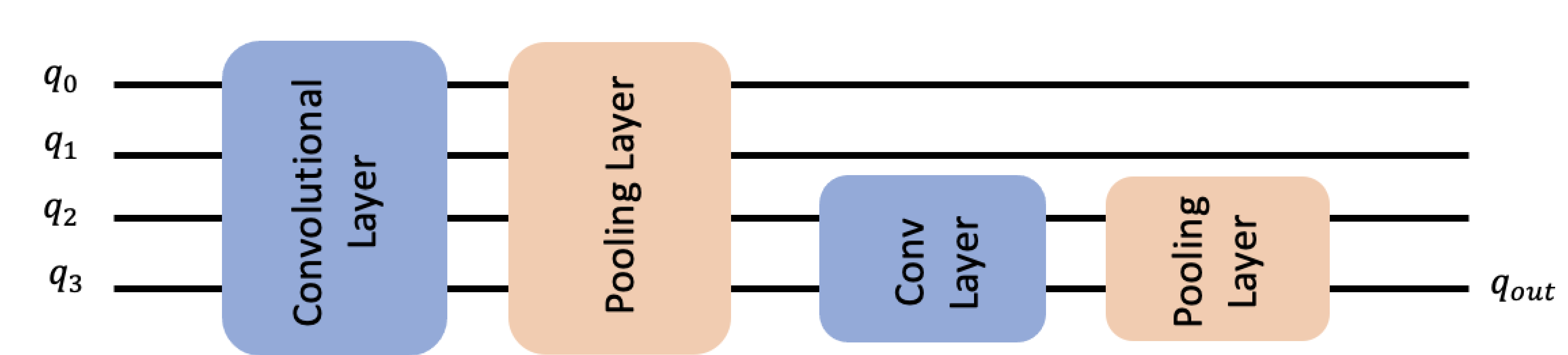
SJTU

Differences between a QCNN and CCNN



Quantum Convolutional Neural Networks

- Quantum Convolutional Neural Networks (QCNN) behave in a similar manner to CCNNs.
- First, we encode our pixelated image into a quantum circuit using a given feature map.
- After encoding our image, we apply alternating convolutional and pooling layers.



Components of a QCNN

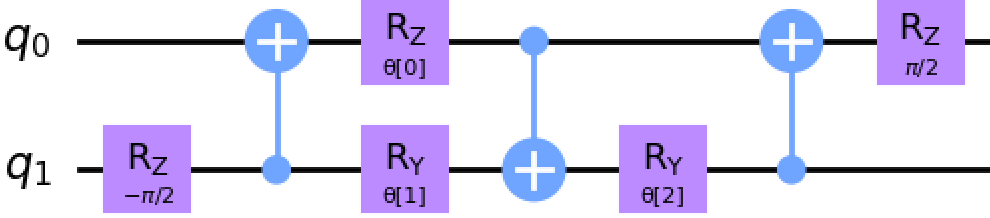
- In theory, one could apply any parametrized circuit for both the convolutional and pooling layers of our network.
- Here, we take a different approach and form our parametrized circuit based on the two qubit unitary. This states that every unitary matrix in U(4) can be decomposed such that
  - $U=(A_1 \otimes A_2)\cdot N(\alpha,\beta,\gamma)\cdot(A_3 \otimes A_4)$
- where $A_j \in SU(2)$, $\otimes$ is the tensor product, and $N(\alpha,\beta,\gamma)=\exp(i[\alpha\sigma_x\sigma_x+\beta\sigma_y\sigma_y+\gamma\sigma_z\sigma_z])$, where α,β,γ are the parameters that we can adjust.
- From this, it is evident that each unitary depends on 15 parameters and implies that in order for the QCNN to be able to span the whole Hilbert space, each unitary in our QCNN must contain 15 parameters each.

SJTU

Convolutional Layer

[2]:
```python
# We now define a two qubit unitary as defined in [3]
def conv_circuit(params):
    target = QuantumCircuit(2)
    target.rz(-np.pi / 2, 1)
    target.cx(1, 0)
    target.rz(params[0], 0)
    target.ry(params[1], 1)
    target.cx(0, 1)
    target.ry(params[2], 1)
    target.cx(1, 0)
    target.rz(np.pi / 2, 0)
    return target


# Let's draw this circuit and see what it looks like
params = ParameterVector("θ", length=3)
circuit = conv_circuit(params)
circuit.draw("mpl", style="clifford")
```

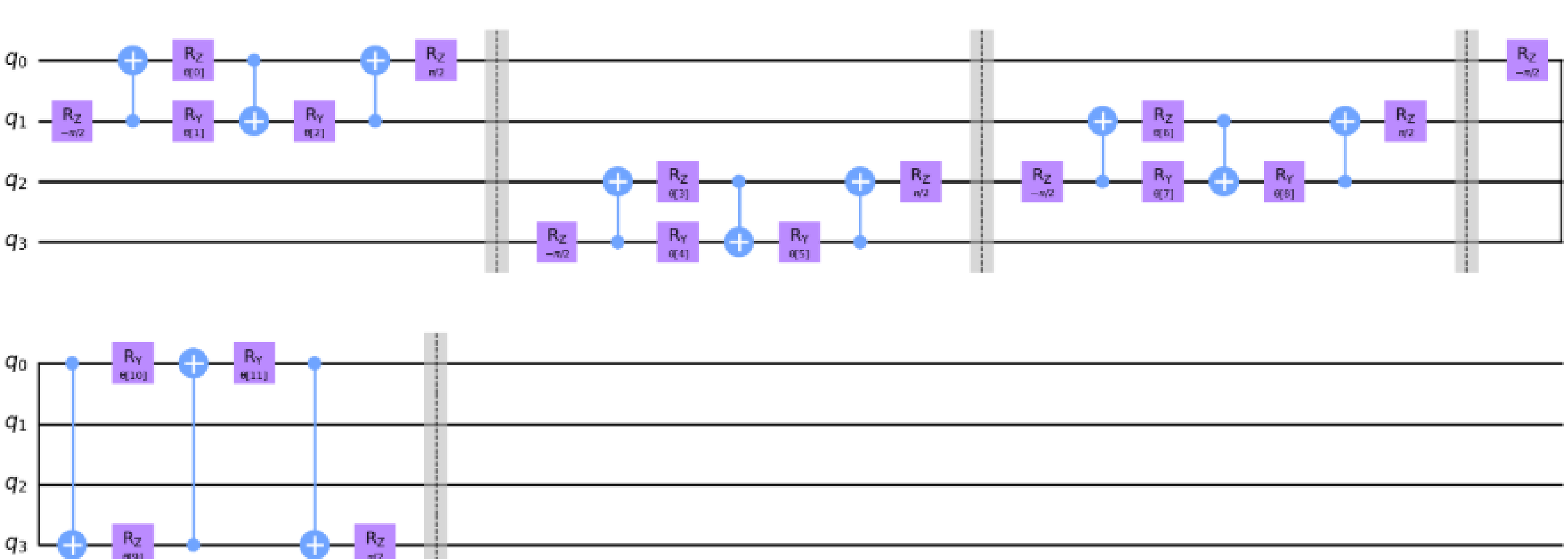
[2]:



Convoluti

[3]:
```python
def conv_layer(num_qubits, param_prefix):
    qc = QuantumCircuit(num_qubits, name="Convolutional Layer")
    qubits = list(range(num_qubits))
    param_index = 0
    params = ParameterVector(param_prefix, length=num_qubits * 3)
    for q1, q2 in zip(qubits[0::2], qubits[1::2]):
        qc = qc.compose(conv_circuit(params[param_index : (param_index + 3)]), [q1, q2])
        qc.barrier()
        param_index += 3
    for q1, q2 in zip(qubits[1::2], qubits[2::2] + [0]):
        qc = qc.compose(conv_circuit(params[param_index : (param_index + 3)]), [q1, q2])
        qc.barrier()
        param_index += 3

    qc_inst = qc.to_instruction()

    qc = QuantumCircuit(num_qubits)
    qc.append(qc_inst, qubits)
    return qc


circuit = conv_layer(4, "θ")
circuit.decompose().draw("mpl", style="clifford")
```

[3]:



Pooling Layer

[4]:
```python
def pool_circuit(params):
    target = QuantumCircuit(2)
    target.rz(-np.pi / 2, 1)
    target.cx(1, 0)
    target.rz(params[0], 0)
    target.ry(params[1], 1)
    target.cx(0, 1)
    target.ry(params[2], 1)

    return target


params = ParameterVector("θ", length=3)
circuit = pool_circuit(params)
circuit.draw("mpl", style="clifford")
```

[4]:

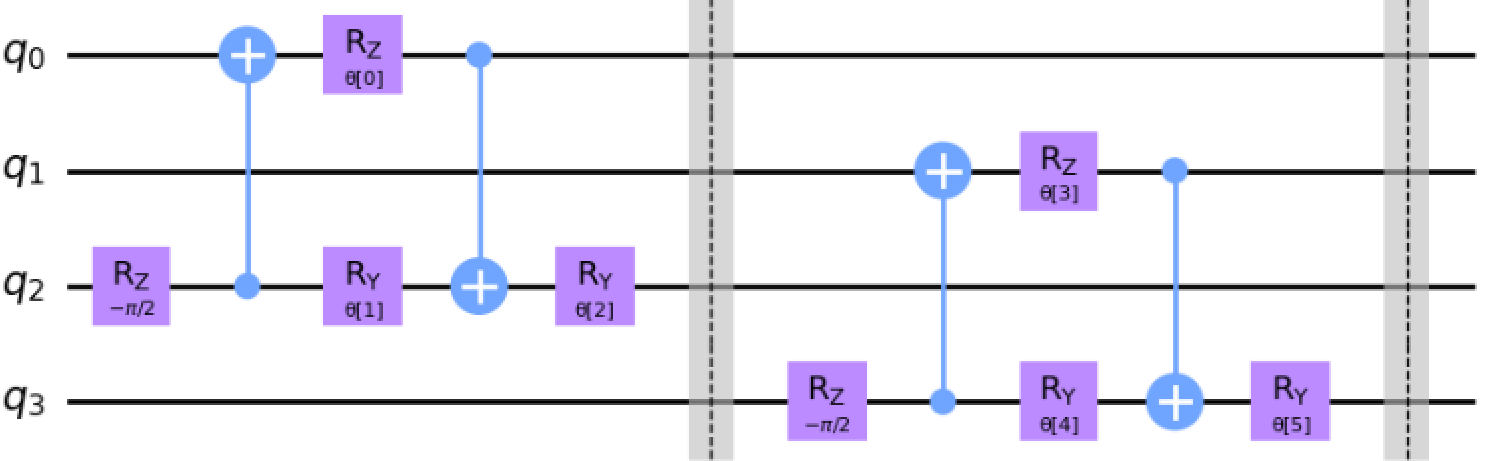
q_0 ——— ⊕ — R_Z θ[0] — ● ———

q_1 — R_Z −π/2 — ● — R_Y θ[1] — ⊕ — R_Y θ[2] —

Pooling L

```
[5]: def pool_layer(sources, sinks, param_prefix):
         num_qubits = len(sources) + len(sinks)
         qc = QuantumCircuit(num_qubits, name="Pooling Layer")
         param_index = 0
         params = ParameterVector(param_prefix, length=num_qubits // 2 * 3)
         for source, sink in zip(sources, sinks):
             qc = qc.compose(pool_circuit(params[param_index : (param_index + 3)]), [source, sink])
             qc.barrier()
             param_index += 3

         qc_inst = qc.to_instruction()

         qc = QuantumCircuit(num_qubits)
         qc.append(qc_inst, range(num_qubits))
         return qc


     sources = [0, 1]
     sinks = [2, 3]
     circuit = pool_layer(sources, sinks, "θ")
     circuit.decompose().draw("mpl", style="clifford")
```

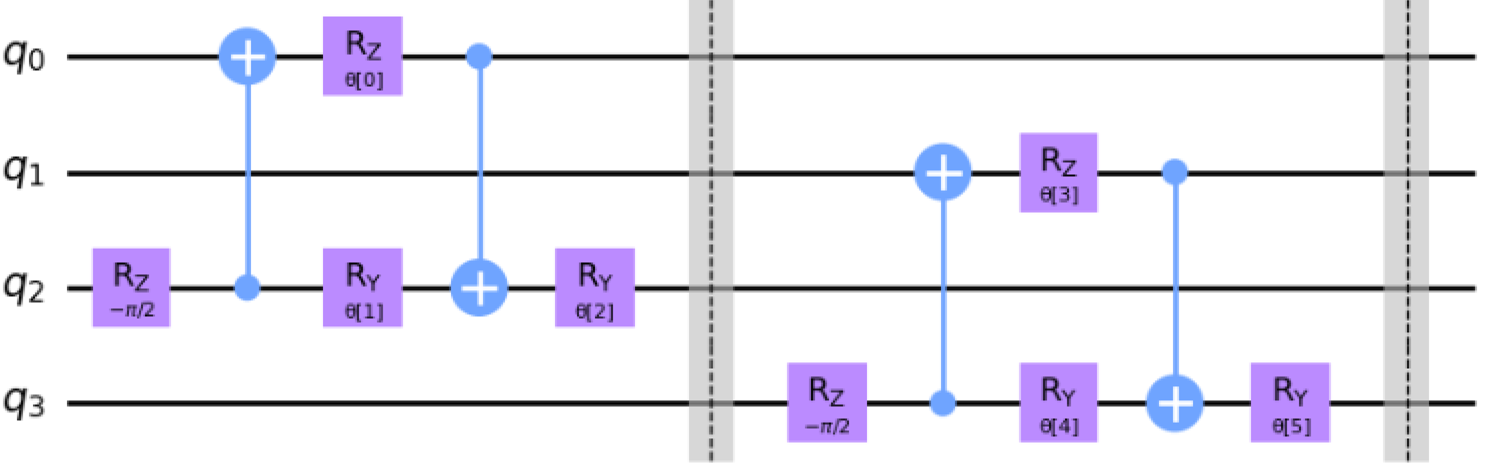
[5]:



Pooling L

[5]:
```python
def pool_layer(sources, sinks, param_prefix):
    num_qubits = len(sources) + len(sinks)
    qc = QuantumCircuit(num_qubits, name="Pooling Layer")
    param_index = 0
    params = ParameterVector(param_prefix, length=num_qubits // 2 * 3)
    for source, sink in zip(sources, sinks):
        qc = qc.compose(pool_circuit(params[param_index : (param_index + 3)]), [source, sink])
        qc.barrier()
        param_index += 3

    qc_inst = qc.to_instruction()

    qc = QuantumCircuit(num_qubits)
    qc.append(qc_inst, range(num_qubits))
    return qc


sources = [0, 1]
sinks = [2, 3]
circuit = pool_layer(sources, sinks, "θ")
circuit.decompose().draw("mpl", style="clifford")
```

[5]:



Data Generation

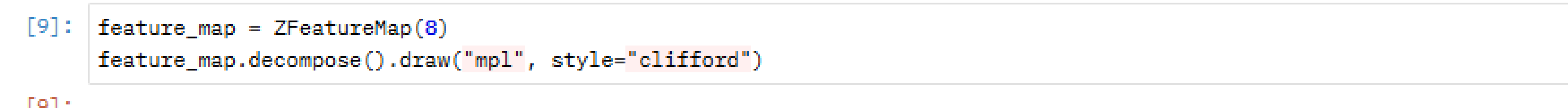
```
[7]: images, labels = generate_dataset(50)

     train_images, test_images, train_labels, test_labels = train_test_split(
         images, labels, test_size=0.3, random_state=246
     )
```

Let's see some examples in our dataset

```
[8]: fig, ax = plt.subplots(2, 2, figsize=(10, 6), subplot_kw={"xticks": [], "yticks": []})
     for i in range(4):
         ax[i // 2, i % 2].imshow(
             train_images[i].reshape(2, 4),  # Change back to 2 by 4
             aspect="equal",
         )
     plt.subplots_adjust(wspace=0.1, hspace=0.025)
```

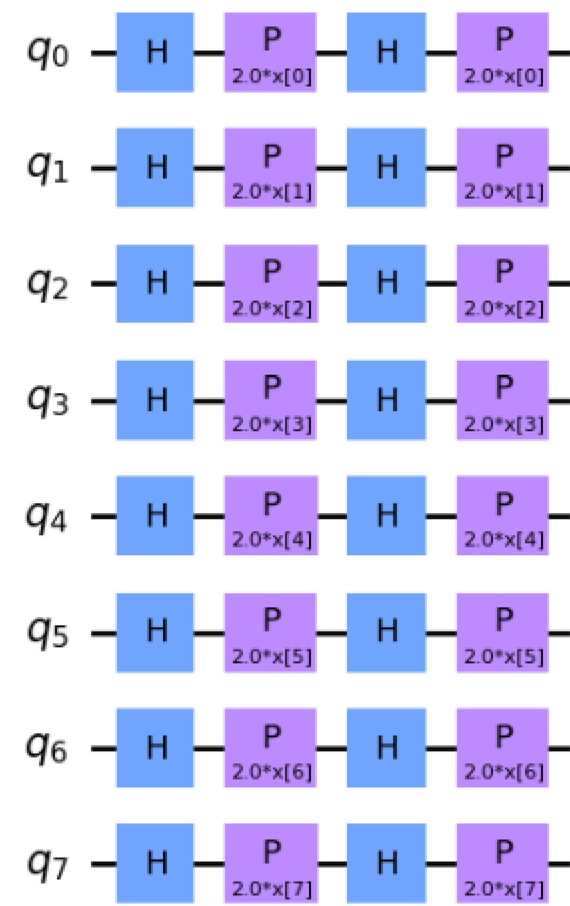
Modeling our QCNN

Data embedding



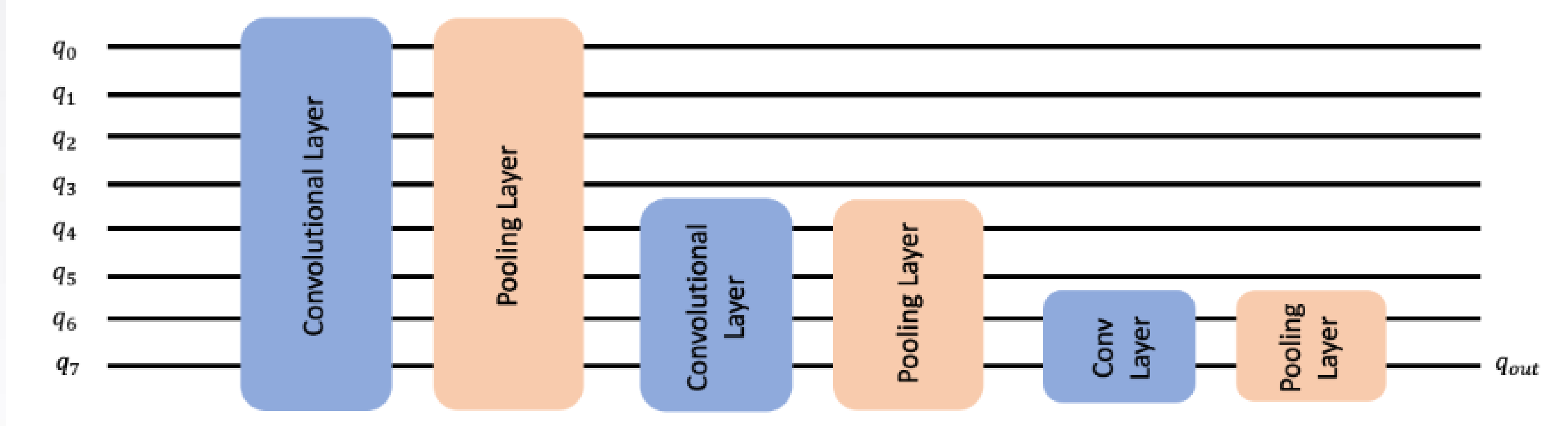
```
[9]: feature_map = ZFeatureMap(8)
     feature_map.decompose().draw("mpl", style="clifford")
```

[9]:



Modeling our QCNN

Ansatz



Modeling our QCNN

Ansatz

```
[10]: feature_map = ZFeatureMap(8)

ansatz = QuantumCircuit(8, name="Ansatz")

# First Convolutional Layer
ansatz.compose(conv_layer(8, "c1"), list(range(8)), inplace=True)

# First Pooling Layer
ansatz.compose(pool_layer([0, 1, 2, 3], [4, 5, 6, 7], "p1"), list(range(8)), inplace=True)

# Second Convolutional Layer
ansatz.compose(conv_layer(4, "c2"), list(range(4, 8)), inplace=True)

# Second Pooling Layer
ansatz.compose(pool_layer([0, 1], [2, 3], "p2"), list(range(4, 8)), inplace=True)

# Third Convolutional Layer
ansatz.compose(conv_layer(2, "c3"), list(range(6, 8)), inplace=True)

# Third Pooling Layer
ansatz.compose(pool_layer([0], [1], "p3"), list(range(6, 8)), inplace=True)

# Combining the feature map and ansatz
circuit = QuantumCircuit(8)
circuit.compose(feature_map, range(8), inplace=True)
circuit.compose(ansatz, range(8), inplace=True)

observable = SparsePauliOp.from_list([("Z" + "I" * 7, 1)])

# we decompose the circuit for the QNN to avoid additional data copying
qnn = EstimatorQNN(
    circuit=circuit.decompose(),
    observables=observable,
    input_params=feature_map.parameters,
    weight_params=ansatz.parameters,
    estimator=estimator,
)

No gradient function provided, creating a gradient function. If your Estimator requires transpilation, please provide
```

SJTU

Modeling our QCNN

Ansatz

[11]:
```
circuit.draw("mpl", style="clifford")
```

[11]:

q_0 q_1 q_2 q_3 q_4 q_5 q_6 q_7

ZFeatureMap

Convolutional Layer

Pooling Layer

q_0 q_1 q_2 q_3 q_4 q_5 q_6 q_7

Convolutional Layer

Pooling Layer

Convolutional Layer

Pooling Layer

SJTU

Training our QCNN

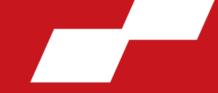
```
[12]: def callback_graph(weights, obj_func_eval):
          clear_output(wait=True)
          objective_func_vals.append(obj_func_eval)
          plt.title("Objective function value against iteration")
          plt.xlabel("Iteration")
          plt.ylabel("Objective function value")
          plt.plot(range(len(objective_func_vals)), objective_func_vals)
          plt.show()
```

```
[13]: with open("11_qcnn_initial_point.json", "r") as f:
          initial_point = json.load(f)

      classifier = NeuralNetworkClassifier(
          qnn,
          optimizer=COBYLA(maxiter=200),  # Set max iterations here
          callback=callback_graph,
          initial_point=initial_point,
      )
```

```
[14]: x = np.asarray(train_images)
      y = np.asarray(train_labels)

      objective_func_vals = []
      plt.rcParams["figure.figsize"] = (12, 6)
      classifier.fit(x, y)

      # score classifier
      print(f"Accuracy from the train data : {np.round(100 * classifier.score(x, y), 2)}%")
```

SJTU

Training our QCNN



Objective function value against iteration

Objective function value

0.80
0.75
0.70
0.65
0.60
0.55
0.50

0 25 50 75 100 125 150 175 200

Iteration

```
Accuracy from the train data : 94.29%
```

Testing

```
[15]: y_predict = classifier.predict(test_images)
x = np.asarray(test_images)
y = np.asarray(test_labels)
print(f"Accuracy from the test data : {np.round(100 * classifier.score(x, y), 2)}%")

# Let's see some examples in our dataset
fig, ax = plt.subplots(2, 2, figsize=(10, 6), subplot_kw={"xticks": [], "yticks": []})
for i in range(0, 4):
    ax[i // 2, i % 2].imshow(test_images[i].reshape(2, 4), aspect="equal")
    if y_predict[i] == -1:
        ax[i // 2, i % 2].set_title("The QCNN predicts this is a Horizontal Line")
    if y_predict[i] == +1:
        ax[i // 2, i % 2].set_title("The QCNN predicts this is a Vertical Line")
plt.subplots_adjust(wspace=0.1, hspace=0.5)
```

```
Accuracy from the test data : 93.33%
```

The QCNN predicts this is a Vertical Line

The QCNN predicts this is a Vertical Line

The QCNN predicts this is a Vertical Line

The QCNN predicts this is a Horizontal Line

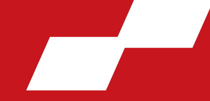
03

TensorFlow Quantum

TensorFlow Quantum

- **TensorFlow Quantum (TFQ) is a Python framework for quantum machine learning.**
- **TensorFlow Quantum implements the components needed to integrate TensorFlow with quantum computing hardware. To that end, TensorFlow Quantum introduces two datatype primitives:**
  - Quantum circuit —This represents a Cirq-defined quantum circuit within TensorFlow. Create batches of circuits of varying size, similar to batches of different real-valued datapoints.
  - Pauli sum —Represent linear combinations of tensor products of Pauli operators defined in Cirq. Like circuits, create batches of operators of varying size.

MNIST classification



1. Load the Data

- Loads the raw data from Keras.
- Filters the dataset to only 3s and 6s.
- Downscales the images so they fit can fit in a quantum computer.
- ~~Removes any contradictory examples.~~
- Converts the binary images to Cirq circuits.
- Converts the Cirq circuits to TensorFlow Quantum circuits.

MNIST classification

- Loads the raw data from Keras.

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Rescale the images from [0,255] to the [0.0,1.0] range.
x_train, x_test = x_train[..., np.newaxis]/255.0, x_test[..., np.newaxis]/255.0

print("Number of original training examples:", len(x_train))
print("Number of original test examples:", len(x_test))
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
Number of original training examples: 60000
Number of original test examples: 10000
```

Filters the dataset to only 3s and 6s.

```python
def filter_36(x, y):
    keep = (y == 3) | (y == 6)
    x, y = x[keep], y[keep]
    y = y == 3
    return x,y
```

```python
x_train, y_train = filter_36(x_train, y_train)
x_test, y_test = filter_36(x_test, y_test)

print("Number of filtered training examples:", len(x_train))
print("Number of filtered test examples:", len(x_test))
```

```
Number of filtered training examples: 12049
Number of filtered test examples: 1968
```

MNIST

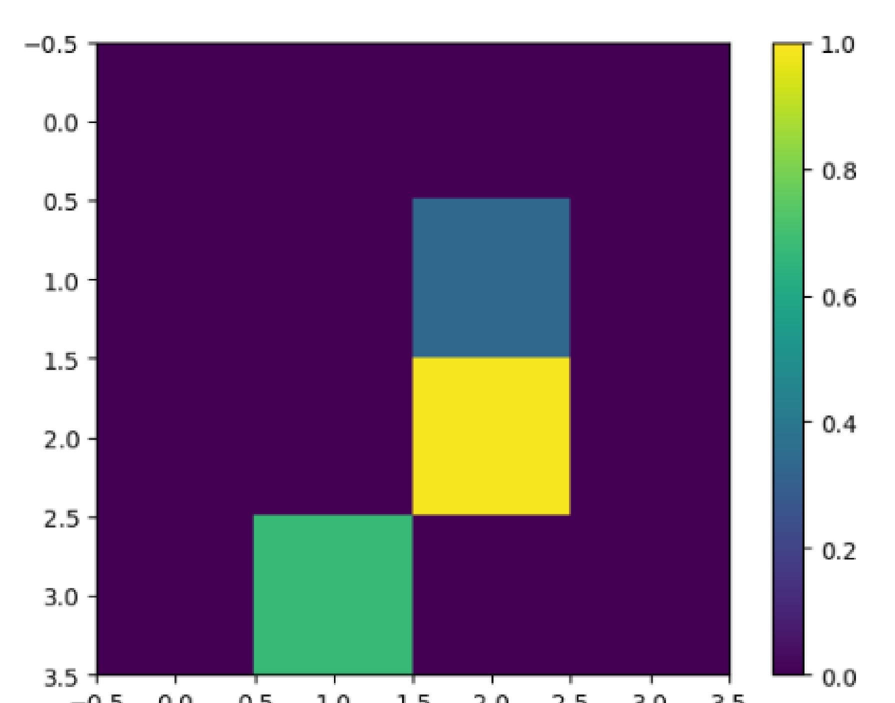
Downscales t

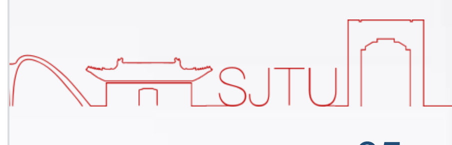
```
x_train_small = tf.image.resize(x_train, (4,4)).numpy()
x_test_small = tf.image.resize(x_test, (4,4)).numpy()
```

Again, display the first training example—after resize:

```
print(y_train[0])

plt.imshow(x_train_small[0,:,:,0], vmin=0, vmax=1)
plt.colorbar()
```

```
True
<matplotlib.colorbar.Colorbar at 0x7f8c5a54b4f0>
```





MNIST classification

Encode the data as quantum circuits

```
def convert_to_circuit(image):
    """Encode truncated classical image into quantum datapoint."""
    values = np.ndarray.flatten(image)
    qubits = cirq.GridQubit.rect(4, 4)
    circuit = cirq.Circuit()
    for i, value in enumerate(values):
        if value:
            circuit.append(cirq.X(qubits[i]))
    return circuit


x_train_circ = [convert_to_circuit(x) for x in x_train_bin]
x_test_circ = [convert_to_circuit(x) for x in x_test_bin]
```

```
SVGCircuit(x_train_circ[0])
```

(2, 2): ─ X ─

(3, 1): ─ X ─

```
x_train_tfcirc = tfq.convert_to_tensor(x_train_circ)
x_test_tfcirc = tfq.convert_to_tensor(x_test_circ)
```

SJTU

MNIST classification

2. Quantum

```
class CircuitLayerBuilder():
    def __init__(self, data_qubits, readout):
        self.data_qubits = data_qubits
        self.readout = readout

    def add_layer(self, circuit, gate, prefix):
        for i, qubit in enumerate(self.data_qubits):
            symbol = sympy.Symbol(prefix + '-' + str(i))
            circuit.append(gate(qubit, self.readout)**symbol)
```

Build an example circuit layer to see how it looks:

```
demo_builder = CircuitLayerBuilder(data_qubits = cirq.GridQubit.rect(4,1),
                                   readout=cirq.GridQubit(-1,-1))

circuit = cirq.Circuit()
demo_builder.add_layer(circuit, gate = cirq.XX, prefix='xx')
SVGCircuit(circuit)
```

(-1, -1):	XX	XX	XX	XX
(0, 0):	XX^(xx-0)			
(1, 0):		XX^(xx-1)		
(2, 0):			XX^(xx-2)	
(3, 0):				XX^(xx-3)

2. Quantum neural network

```
def create_quantum_model():
    """Create a QNN model circuit and readout operation to go along with it."""
    data_qubits = cirq.GridQubit.rect(4, 4)  # a 4x4 grid.
    readout = cirq.GridQubit(-1, -1)         # a single qubit at [-1,-1]
    circuit = cirq.Circuit()

    # Prepare the readout qubit.
    circuit.append(cirq.X(readout))
    circuit.append(cirq.H(readout))

    builder = CircuitLayerBuilder(
        data_qubits = data_qubits,
        readout=readout)

    # Then add layers (experiment by adding more).
    builder.add_layer(circuit, cirq.XX, "xx1")
    builder.add_layer(circuit, cirq.ZZ, "zz1")

    # Finally, prepare the readout qubit.
    circuit.append(cirq.H(readout))

    return circuit, cirq.Z(readout)
```

```
model_circuit, model_readout = create_quantum_model()
```

MNIST classification

3. Train the

```
EPOCHS = 3
BATCH_SIZE = 32

NUM_EXAMPLES = len(x_train_tfcirc)
```

```
x_train_tfcirc_sub = x_train_tfcirc[:NUM_EXAMPLES]
y_train_hinge_sub = y_train_hinge[:NUM_EXAMPLES]
```

Training this model to convergence should achieve >85% accuracy on the test set.

```
qnn_history = model.fit(
      x_train_tfcirc_sub, y_train_hinge_sub,
      batch_size=32,
      epochs=EPOCHS,
      verbose=1,
      validation_data=(x_test_tfcirc, y_test_hinge))

qnn_results = model.evaluate(x_test_tfcirc, y_test)
```

```
Epoch 1/3
324/324 [==============================] - 56s 172ms/step - loss: 0.6782 - hinge_accuracy: 0.7792 - val_los
Epoch 2/3
324/324 [==============================] - 56s 171ms/step - loss: 0.3630 - hinge_accuracy: 0.8503 - val_los
Epoch 3/3
324/324 [==============================] - 56s 171ms/step - loss: 0.3502 - hinge_accuracy: 0.8776 - val_los
62/62 [==============================] - 2s 32ms/step - loss: 0.3321 - hinge_accuracy: 0.8725
```

谢谢！

饮 水 思 源　爱 国 荣 校