# Google Summer of Code - Final Report

---

## Quple

### Quantum Machine Learning Framework for High Energy Physics

https://gitlab.cern.ch/clcheng/quple

## Introduction

The goal of the Quple project is to implement a common framework for applying quantum machine learning algorithms to high energy physics analysis base on the google Cirq [1] and tensorflow quantum [4] libraries. Specifically, a variational quantum classifier (VQC)[3] was developed for applications in binary classification problems such as distinguishing signal and background even ts in high energy physics experiments.

The variational quantum classifier (VQC) is a hybrid quantum-classical algorithm that utilizes both quantum and classical resources for solving classification problems. The VQC is a kind of quantum neural network (QNN) where the model representation of the is based entirely on the quantum processor, with classical heuristics participating only as optimizers for the trainable parameters of the quantum model [4].

More specifically, the VQC defines a quantum model $f(\mathbf{x},\boldsymbol{\theta}) = \langle \mathbf{0}| U^\dagger(\mathbf{x},\boldsymbol{\theta}) M U(\mathbf{x},\boldsymbol{\theta}) |\mathbf{0}\rangle$ as the expectation value of some observable $M$ with respect to a state prepared via the unitary operation $U(\mathbf{x},\boldsymbol{\theta})$ to the initial state $|\mathbf{0}\rangle = |0\rangle^{\otimes n}$ of the quantum computer with an $n$ qubit register. Here $U(\mathbf{x},\boldsymbol{\theta})$ is a quantum circuit that depends on the data input $\mathbf{x}$ and a set of parameters $\boldsymbol{\theta}$ which can be seen as the weights in a neural network. $U(\mathbf{x},\boldsymbol{\theta})$ can be written as a composition of a data encoding circuit $S(\mathbf{x})$ and a variational circuit $W(\boldsymbol{\theta})$, i.e. $U(\mathbf{x},\boldsymbol{\theta}) = S(\mathbf{x})W(\boldsymbol{\theta})$. The circuit is run multiple times and the measurement results are averaged to obtain the expectation value. Optionally a classical activation function $\psi$ may be used to map the results from $f(\mathbf{x},\boldsymbol{\theta})$ to the predicted class label, i.e. $y = \psi(f(\mathbf{x},\boldsymbol{\theta}))$. The parameters $\boldsymbol{\theta}$ are then trained in an iterative manner by a classical optimizer to optimize over some cost function to obtain the best model.

The Quple project is made into a python library under the same name and the full source code can be found at https://gitlab.cern.ch/clcheng/quple.

The next section is a summary of the code implemention of the various components developed by Quple.

# Code Implementation

## Parameterized Quantum Circuit

A parameterized quantum circuit (PQC) [3] is the core building blocks of many quantum machine learning algorithms. In particular, both the data encoding circuit and the variational circuit of a VQC can be implemented as a parameterized quantum circuit. A quantum circuit consists of a system of qubits together with a sequence of unitary operations (quantum gates) applied to the qubits which transform the quantum state of the system. A quantum circuit is parameterized if it contains gate operations with undetermined parameter expressions. For example, the gate operation $Rx(\theta)$ is the Pauli rotation on the qubit about the $x$-axis by an angle parameterized by $\theta$.

In Quple, the construction of parameterized quantum circuits is implemented by the `ParameterizedCircuit` class based on the google Cirq library. It uses a specific quantum circuit design called the circuit-centric [8] architecture which is composed of some $L$ copies (called the circuit depth) of a primary circuit block. For a quantum circuit with $n$ qubits, each circuit block consists of a layer (called the rotation layer) of single qubit gates applied to each qubit followed by a layer (called the entanglement layer) of two (or three) qubit gates to entangle the qubits with a given connectivity (see section 2). It is a natural design that can be easily implemented in actual quantum computers and its strongly entangling nature has various advantages including error mitigation, the ability to efficiently represent the solution space of some machine learning tasks and to capture nontrivial correlation in the quantum data.

A `ParameterizedCircuit` instance can be created by specifying the number of qubits $n$ in the circuit, the single-qubit quantum gates used in the rotation layer, the multi-qubit quantum gates used in the entanglement layer and the number of copies these layers appear in the circuit. Symbolic parameters are implemented as `sympy.Symbol` objects from sympy library which will be automatically filled to the quantum gates in each layer. The symbolic parameters have a common prefix, such as $\theta$, followed by a numeric suffix indicating the order of creation of the symbols. For example, a parameterized circuit with a single layer of $Rx(\theta)$ rotation on 3 qubits with have parameterized symbols $\theta_0, \theta_1$ and $\theta_2$ representing the rotation angles about the $x$-axis for the first, second and third qubit respectively. The symbolic parameters can then be resolved by specifying a symbol-to-value map or an array of values which will be interpreted as the values of the sorted list of symbols in a given circuit. The sorting of symbols are done according to a natural sorting method, e.g. $\theta_1 < \theta_2 < \theta_{10}$. The following code snippet demonstrates how to create a paraterized circuit using `ParameterizedCircuit` and resolve the parameter

values in the circuit.

```
from quple import ParameterizedCircuit
import numpy as np

circuit = ParameterizedCircuit(n_qubit=4, rotation_blocks=['RX','RY'], entanglement_blocks=['ZZ'],
                               entangle_strategy='full', copies=1)

num_params = len(circuit.parameters)
parameter_values = np.linspace(0, 0.01*np.pi*(num_params-1), num_params)
resolved_circuit = circuit.resolve_parameters(parameter_values)
```

The resulting circuits are shown in Figure 1.

Figure 1: (Top) A parameterized quantum circuit consisting of a single-qubit $R_x(\theta)$ and $R_y(\theta)$ rotation layer and a fully connected two-qubit $ZZ^\theta$ entanglement layer. The two-qubit Ising coupling gate $ZZ^\theta = \exp(-i\pi\theta Z \otimes Z)$ represents a rescaled rotation around the $ZZ$ axis in the two-qubit bloch sphere. (Bottom) A resolved version of the parameterized quantum circuit with $\theta_k = 0.01k\pi$ for $k = 0 \ldots 13$.

A Colab example notebook for the construction of parameterised quantum circuits can be found at

examples/Parameterized_Circuits_Walkthrough.ipynb

### Interaction Graph

An interaction graph determines how the qubits in a quantum circuit are connected by a specific gate operation. In the circuit centric architecture, all qubits must be touched by the gate operation at least once. If the gate operation acts on $m$ qubits for a circuit with $n$ qubits then an interaction graph for that gate operation will consist of a collection of distinct $m$-tuple of qubits created from the $n$ circuit qubits. The most common use cases for an interaction graph is to determine how qubits are entangled by a two-qubit gate operation.

The interaction graph category implemented by Quple are: `nearest_neighbor` (or `linear`), `cyclic` (or `circular`), `star` and `fully_connected` (or simply `full`). In the `nearest_neighbor` interaction, each qubit is connected to its next nearest neighbor in a linear manner. In the `cyclic` interaction, each qubit is connected to its next nearest neighbor in a circular manner. In the `star` interaction, the first qubit is connected to every other qubit. In the `fully_connected` interaction, every distinct ordered tuple of $m$ qubits are connected. The following code snippet shows the resultant interation graphs created from each of the above categories for a circuit with $n = 4$ qubits and a gate operation acting on $m = 2$ qubits.

```
from quple.components.interaction_graphs import nearest_neighbor, cyclic, star, fully_connected

print(nearest_neighbor(n=4, m=2))
# output: [(0, 1), (1, 2), (2, 3)]
print(cyclic(n=4, m=2))
# output: [(0, 1), (1, 2), (2, 3), (3, 0)]
print(fully_connected(n=4, m=2))
# output: [(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
print(star(n=4, m=2) )
# output: [(0, 1), (0, 2), (0, 3)]
```

A practical use of interaction graphs is in the construction of the entanglement layer of `ParameterizedCircuit` in which case the interaction graphs are passed as the `entangle_strategy` argument. The following code snippet shows the construction of various `ParameterizedCircuit` instances with a CNOT entanglement layer using different entangle strategies.

```
from quple import ParameterizedCircuit

A = ParameterizedCircuit(n_qubit=4, entanglement_blocks=['CNOT'], entangle_strategy='linear')
B = ParameterizedCircuit(n_qubit=4, entanglement_blocks=['CNOT'], entangle_strategy='circular')
C = ParameterizedCircuit(n_qubit=4, entanglement_blocks=['CNOT'], entangle_strategy='star')
D = ParameterizedCircuit(n_qubit=4, entanglement_blocks=['CNOT'], entangle_strategy='full')
```

The resulting circuits are shown in Figure 2.

Users can also create their own interaction graph which takes the argument of $n =$ number of qubits in the circuit and $m =$ number of qubits acted on by the gate operation and output a list containing tuples with size $m$.

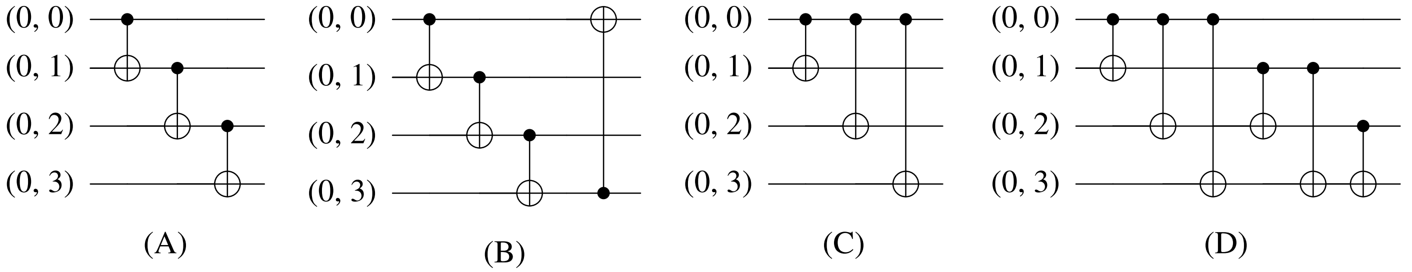



Figure 2: Quantum circuits of 4 qubits with a CNOT entanglement layer constructed with (A) linear, (B) circular, (C) star and (D) full connectivity.

A Colab example notebook for the usage of interaction graphs can be found at

examples/Interaction_Graphs_Walkthrough.ipynb

## Encoding Circuit

A data encoding circuit is a quantum circuit for the encoding of classical data into the quantum state of the circuit qubits. It can be interpreted as a feature map $\mathbf{x} \rightarrow U_{\phi(\mathbf{x})}|0\rangle^{\otimes n}$ to the Hilbert space of $n$ qubits, where $\phi$ is an encoding function which transforms the data vector into the circuit parameters. Specifically, $U_{\phi(\mathbf{x})}$ is implemented as a series of unitary gate operations in the quantum circuit. There are several ways to encode data into qubits and each one provides different expressive power to the original data.

As a system of $n$ qubits will have a quantum state that spans a $2^n$ dimensional Hilbert space, it is desirable to harness this quantum property to embed classical data into a higher dimensional feature space where a specific problem may be easier to solve. In particular, Quple implements a data encoding scheme in which the number of qubits matches the dimension of the data vector. This is in contract to the amplitude encoding method in which a feature map ecnodes a $2^n$ dimensional data vector into the quantum states of $n$ qubits.

In Quple, the construtcion of an encoding circuit is implemented by the `EncodingCircuit` class. Specifically, the derived class `GeneralPauliEncoding` creates an encoding circuit consisting of layers of unitary operators of the form $\exp(i\psi(x)\Sigma)H^{\times n}$ where $\psi$ is a data encoding function, $\Sigma$ is a generalized Pauli operator from the general Pauli group $G_n$ which is an $n$-fold tensor product of Pauli operators on $n$ qubits, and $x = (x_1, \ldots, x_n)$ are the input features to be encoded. Layers of Pauli operators may be repeated several times (called the circuit depth) to the increase frequency spectrum of the final quantum state and thereby the expressivity of a quantum model equipped with the encoding circuit.

To encode data of feature dimension $n$, a set of general Pauli operators are chosen to encode the data into an $n$ qubit circuit. Each Pauli operator will contribute to a unitary operation $\exp\left(i \sum_{s\in S} \psi_s(x_s)\Sigma_s\right)$ where $s$ is the

indices of a subset of all qubit indices $S$. For a general Pauli operator of order $k$, $s$ is a tuple of $k$ elements.

For example Suppose the Pauli operator $\Sigma = Z$ is used, then $S$ is the set $\{1, 2, \ldots, n\}$ and the unitary operation is $\exp\left(i \sum_{j=1}^{n} \psi_j(x_j) Z_j\right) H^{\times n}$, where $Z_j$ is the Pauli $Z$ operator acting on the $j$-th qubit. The following code snippet shows how to construct an encoding circuit with `GeneralPauliEncoding` to encode data vector of feature dimension 3 using the Pauli operator $\Sigma = Z$ with circuit depth 2.

```
from quple.data_encoding import GeneralPauliEncoding
cirquit = GeneralPauliEncoding(feature_dimension=3, paulis=['Z'], copies=2)
```

The encoding circuit created from the above code snippet is shown in Figure 3.

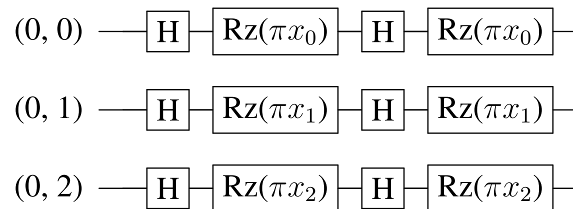



Figure 3: An encoding circuit constructed from `GeneralPauliEncoding` to encode data vector of feature dimension 3 using the Pauli operators $Z$ with circuit depth 2.

If instead the Pauli Operator $\Sigma = Z \otimes Z$ is used, then $S$ is a set of 2-tuple of qubit indices determined by the interaction graph. For a fully connected graph, $S$ is the set of combinations of 2-qubit pairs. Then the unitary operation is $\exp\left(i \sum_{s=(j,k)\in S} \psi_{j,k}(x_j, x_k) Z_j \times Z_k\right) H^{\otimes n}$. The following code snippet shows how to construct an encoding circuit with `GeneralPauliEncoding` to encode data vector of feature dimension 3 using the Pauli operators $Z$ and $ZZ$ with circuit depth 1 with linear entanglement.

```
from quple.data_encoding import GeneralPauliEncoding
cirquit = GeneralPauliEncoding(feature_dimension=4, paulis=['Z','ZZ'], copies=1, entangle_strategy='linear')
```

The encoding circuit created from the above code snippet is shown in Figure 4.

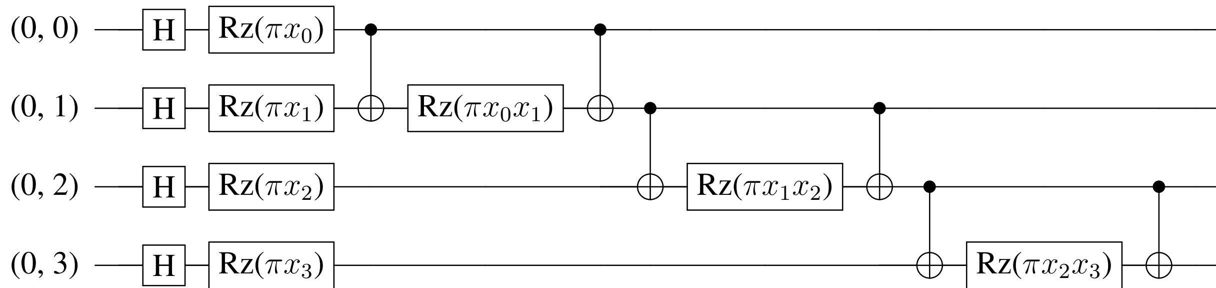



Figure 4: An encoding circuit constructed from GeneralPauliEncoding to encode data vector of feature dimension 3 using the Pauli operators $Z$ and $ZZ$ with circuit depth 1 with linear entanglement.

A Colab example notebook for the construction of data encoding circuits can be found at

examples/Encoding_Circuit_Walkthrough.ipynb

**Encoding Function**

An encoding function $\phi(\mathbf{x})$ specifies how input features $\mathbf{x}$ are encoded into the parameters of a unitary gate operation in an encoding circuit. Usually, the parameters involved in a unitary gate operation are the rotation angles about some axis of the single qubit or multi-qubit Bloch sphere depending on the number of qubits the gate operation is acting on. Therefore, for input features of dimension $n$, the encoding function is a map $f : \mathbb{R}^n \to \mathbb{R}$. It is natural to restrict the range of the encoding function to be within $(0, 2\pi)$ or $(-\pi, \pi)$ to correspond to the effective range of an angle of rotation.

In Quple, encoding circuits from the EncodingCircuit class will have its gate operations parameterized by the expressions of the form $\pi\phi(\mathbf{x})$ with the $\pi$ factor extracted out by default which restricts the range of $\phi(\mathbf{x})$ to be within $[0, 2]$ or $[-1, 1]$. There are a number of encoding functions that are implemented in Quple which also put a restriction on the value of each input feature to be within $[-1, +1]$ to make sure the encoding functions will map input features of arbitrary dimension to a value of the required range. A table of the encoding functions implemented in Quple is shown in Figure 1. Users can also create their own encoding function as long as it takes an array like input $\mathbf{x}$ and output a number that is within the required range.

| Encoding Function | Formula ($n = 1$) | Formula ($n > 1$) | Domain | Range |
|---|---|---|---|---|
| self product | $x_0$ | $\prod_{i=0}^{n} x_i$ | [-1, +1] | [-1, +1] |
| cosine product | $x_0$ | $\prod_{i=0}^{n}(\cos(\pi(x_i+1)/2))$ | [-1, +1] | [1, +1] |
| distance measure | $x_0$ | $\prod_{i<j}(x_i - x_j)/2^{\text{pairs}}$ | [-1, +1] | [-1, +1] |
| one norm distance | $x_0$ | $\sum_{i<j} \lvert x_i - x_j \rvert/\text{pairs}$ | [-1, +1] | [0, +2] |
| two norm distance | $x_0$ | $[\sum_{i<j}(x_i - x_j)^2/\text{pairs}]^{1/2}$ | [-1, +1] | [0, +2] |
| arithmetic mean | $x_0$ | $\sum_{i=0}^{n} x_i/n$ | [-1, +1] | [-1, +1] |
| second moment | $x_0$ | $[\sum_{i=0}^{n}(x_i+1)^2/n]^{1/2}$ | [-1, +1] | [-1, +1] |
| cube sum | $x_0$ | $\sum_{i=0}^{n} x_i^3/n$ | [-1, +1] | [-1, +1] |
| exponential square sum | $x_0$ | $2\exp[(\sum_{i=0}^{n} x_i^2/n) - 1]$ | [-1, +1] | [$2\exp(-1)$, +2] |
| exponential cube sum | $x_0$ | $2\exp[(\sum_{i=0}^{n} x_i^3/n) - 1]$ | [-1, +1] | [$2\exp(-2)$, +2] |

Table 1: A table of encoding functions implemented by Quple.

The following code snippet shows how to construct an encoding circuit with `GeneralPauliEncoding` using the Paulis $Z$ and $ZZ$ for encoding data of feature dimension 3 using the cube sum encoding function.

```
from quple.data_encoding.encoding_maps import cube_sum
from quple.data_encoding import GeneralPauliEncoding

encoding_circuit = GeneralPauliEncoding(feature_dimension=3, paulis=['Z', 'ZZ'],
                                        encoding_map=cube_sum, entangle_strategy='linear', copies=1)
```

The encoding circuit created from the above code snippet is shown is Figure 5.

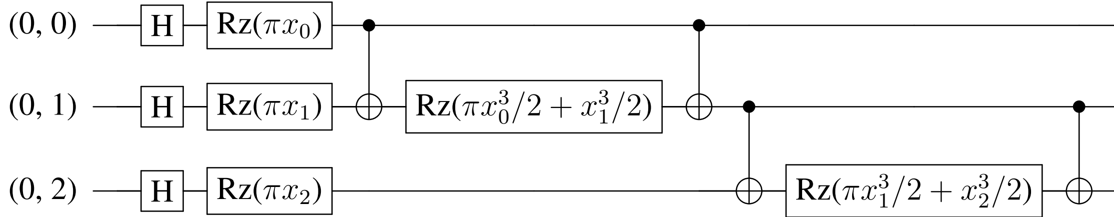



Figure 5: The `GeneralPauliEncoding` circuit with the Paulis $Z$ and $ZZ$ for encoding data of feature dimension 3 using the cube sum encoding function

A Colab example notebook for the usage of encoding functions can be found at

`examples/Encoding_Function_Walkthrough.ipynb`

## Variational Circuit

A variational (or model) circuit is a parameterized quantum circuit representing a unitary $U(\theta)$ parameterized by a set of free parameters $\theta$ which are treated as the weights in a machine learing model. The variational circuit implemented by Quple is based on a circuit-centric design [8]. In this architecture, a variational circuit of $n$ qubits

is composed of $L$ copies (i.e. the circuit depth) of a primary circuit block. Each circuit block consists of a layer of single qubit gates (the rotation layer) applied to each of the qubits, followed by a layer of two qubit gates (the entanglement layer) to entangle all qubits according to a given interaction graph. A final rotation layer is added to the circuit so that measurement on any of the data qubits will effectively include the effect of all the two qubit gates in the entanglement layer. It is an example of a strongly entangling circuit which has the advantage of capturing correlations between the data features at all ranges for a short range circuit.

Currently, Quple has implemented four variational circuit design: `RealAmplitudes`, `EfficientSU2`, `ExcitationPreserv` and `IsingCoupling`. The first three designs are based on the implementation from the Qiskit [2] library.

The `RealAmplitudes` circuit consists of a layer of single qubit Pauli Y rotations acting on each qubit followed by a layer of CNOT entanglement on pairs of qubits under a given interaction graph. The resultant quantum state from the RealAmplitudes circuit will only have real amplitudes with zero complex part. It is a hardware efficient circuit as it uses entangling interactions that are naturally available on hardware and do not require compilation. The following code snippet shows how to construct a `RealAmplitudes` circuit with linear entanglement and circuit depth 2.

```
from quple.circuits.variational_circuits import RealAmplitudes
circuit = RealAmplitudes(n_qubit=4, copies=2, entangle_strategy='linear')
```

The variational circuit created from the above code snippet is shown in Figure 6.

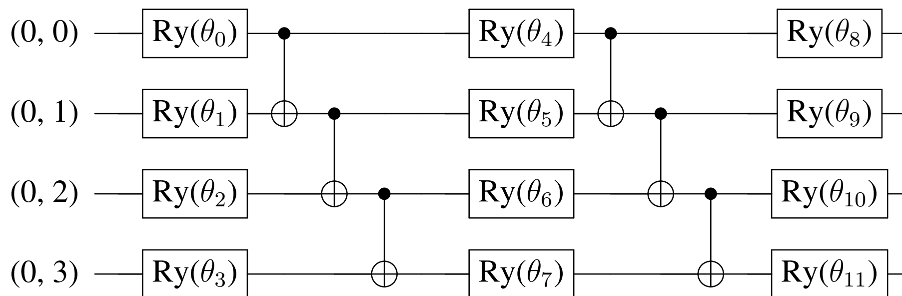



Figure 6: The RealAmplitudes variational circuit with linear entanglement and circuit depth 2.

The `EfficientSU2` circuit consists of a layer of single qubit operations spanned by SU(2) (such as the Pauli X, Y and Y operations and their rotations) acting on each qubit and a layer of CNOT entanglement on pairs of qubits under a given interaction graph. It is a hardware efficient circuit as it uses entangling interactions that are naturally available on hardware and do not require compilation. The following code snippet shows how to construct an `EfficientSU2` circuit with linear entanglement and circuit depth 2.

```
from quple.circuits.variational_circuits import EfficientSU2
circuit = EfficientSU2(n_qubit=4, copies=2, entangle_strategy='linear')
```

The variational circuit created from the above code snippet is shown in Figure 7.

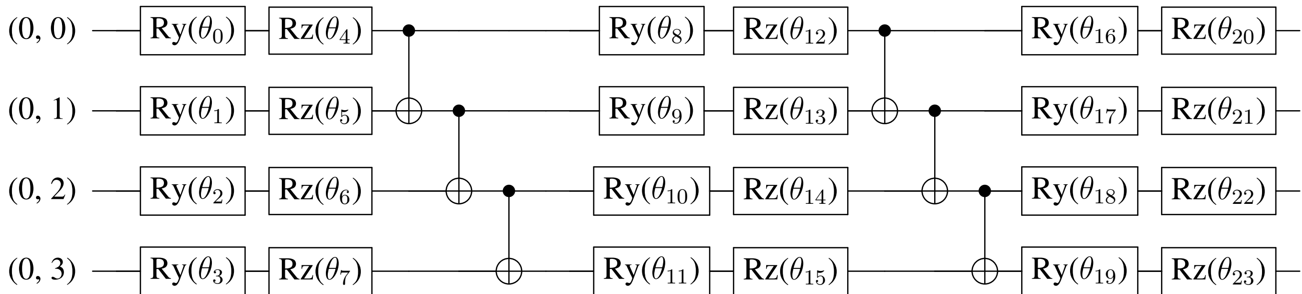



Figure 7: The EfficientSU2 variational circuit with linear entanglement and circuit depth 2.

The `ExcitationPreserving` circuit consists of a layer of single qubit Pauli Z rotation acting on each qubit and a layer of two qubit gates from the Fermionic simulation, or fSim, gate set acting on pairs of qubits under a given interaction graph. Under this gate set, the $\sigma_X\sigma_X$ and $\sigma_Y\sigma_Y$ couplings between the qubits have equal coefficients which conserves the number of excitations of the qubits [6]. Algorithms performed with just Pauli Z rotations and fSim gates enable error mitigation techiques including post selection and zero noise extrapolation [5]. The following code snippet shows how to construct an `ExcitationPreserving` circuit with linear entanglement and circuit depth 1.

```
from quple.circuits.variational_circuits import ExcitationPreserving
circuit = ExcitationPreserving(n_qubit=4, copies=1, entangle_strategy='linear')
```

The variational circuit created from the above code snippet is shown in Figure 8.

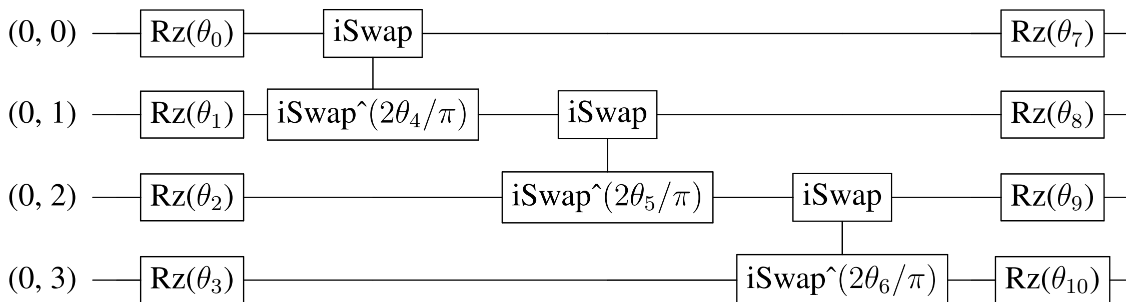



Figure 8: The ExcitationPreserving variational circuit with linear entanglement and circuit depth 1.

The IsingCoupling circuit consists of a layer of single qubit Pauli rotations acting on each qubit and a layer of two qubit XX Ising coupling gates which is a rotation around the $XX$ axis in the two-qubit bloch sphere. The following code snippet shows how to construct an IsingCoupling circuit with linear entanglement and circuit depth 1.

```
from quple.circuits.variational_circuits import IsingCoupling

circuit = IsingCoupling(n_qubit=4, copies=1, entangle_strategy='linear')
```

The variational circuit created from the above code snippet is shown in Figure 9.

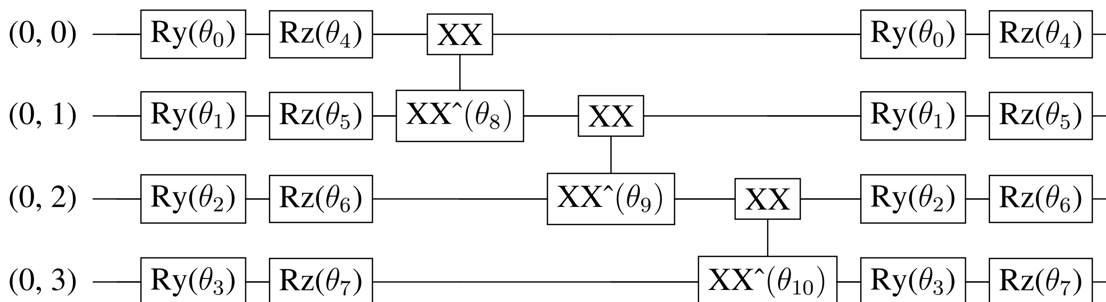



Figure 9: The IsingCoupling variational circuit with linear entanglement and circuit depth 1.

Users can also build their customized variational circuit based on the ParmaterisedCircuit class using the circuit-centric design.

In addition, Quple has implemented the add_readout method under the ParmaterisedCircuit class which allows the addition of an extra readout qubit that is entangled to all data qubits via a custom two-qubit gate operation which is used as the qubit for measurement when training a VQC model. The following code snippet shows how to construct an EfficientSU2 circuit with linear entanglement, circuit depth 1 equipped with a readout qubit that entangled to all data qubits via the two-qubit $XX^{\theta}$ gate.

```
from quple.circuits.variational_circuits import EfficientSU2

circuit = EfficientSU2(n_qubit=5, copies=1, entangle_strategy='linear')
circuit.add_readout('XX')
```

The variational circuit created from the above code snippet is shown in Figure 7.

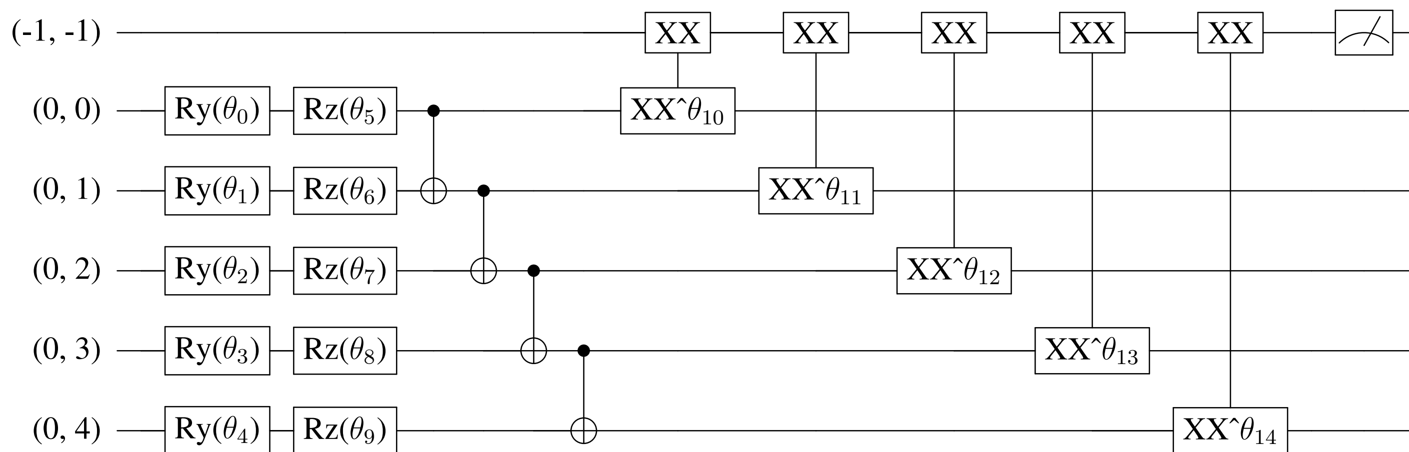



Figure 10: The EfficientSU2 variational circuit with linear entanglement and circuit depth 1. A readout qubit is entangled to all data qubits via the two-qubit $XX^\theta$ gate. The last rotation layer is not shown to save space.

A Colab example notebook for the construction of variational circuits can be found at

examples/Variational_Circuits_Walkthrough.ipynb

## Variational Quantum Classifier

In Quple, the variational quantum classifier is implemented as the quple.classifiers.VQC class. It is derived from the tf.keras.Sequential model defined by an input layer consisting of the data encoding circuit, a tfq.layers.PQC layer consisting of the variational circuit and a final output layer with a custom activation function. The output layer is a classical dense layer with non-trainable weights and only serves to transform the expectation value from the variational circuit measurement. Users can specify the activation function used in the output layer, the optimizer (which should be an instance of tf.keras.Optimizer object) for updating the model weights, the loss function and the metrics in a similar manner as a tensorflow keras model. The following code snippet shows how to construct an VQC model using the GeneralPauliEncoding encoding circuit and the EfficientSU2 variational circuit.

```
import tensorflow as tf
from quple.data_encoding import GeneralPauliEncoding
from quple.circuits.variational_circuits import EfficientSU2
from quple.classifiers import VQC

n_qubit = 5
encoding_circuit = GeneralPauliEncoding(n_qubit, paulis='Z', copies = 2)
variational_circuit = EfficientSU2(n_qubit, copies=2, entangle_strategy='full')
                                   variational_circuit.add_readout('XX')

vqc = VQC(encoding_circuit, variational_circuit, activation='sigmoid',
          optimizer=tf.keras.optimizers.Adam(),
          metrics=['binary_accuracy'], loss='mse',
          readout=[variational_circuit.readout_measurement()])
```

Quple also provides the `quple.components.data_preparation import prepare_train_val_test` method for splitting a dataset into the training, validation and test datasets and carry out data preprocessing using the sklearn [7] libraries. The following code snippet shows how to preprocess a given dataset and train the VQC model using the preprocessed dataset.

```
import numpy as np
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.decomposition import PCA

n_qubit = 5
n_event = 100
batch_size = 64
epochs = 100

# load input data
x, y = np.load('data.npy')

# preprocess input data:
# 1. Apply pca to reduce dimension of data vector to match the number of qubits
# 2. Apply MinMaxScaler to restrict data to the range [-1, +1]
reprocessors = [PCA(n_components=n_qubit), MinMaxScaler((-1,1))]
x_train, x_val, x_test, y_train, y_val, y_test = prepare_train_val_test(
                        x, y, train_size=n_event, val_size=n_event, test_size=n_event,
                        preprocessors=preprocessors, stratify=y)

# train the vqc model
vqc.fit(x_train, y_train, x_val, y_val, x_test, y_test,
        batch_size=batch_size, epochs=epochs)
```

A Colab example notebook for the construction of VQC models can be found at

`examples/Variational_Quantum_Classifier_Walkthrough.ipynb`

## Circuit Descriptors

In designing the suitable data encoding circuit and variational circuit for a VQC model, it is important to have some quantitative description about certain properties of the circuit to serve as indicators for the potential performance of a VQC constructed built upon these circuits. To this mean, Quple has implemented two circuit descriptors, namely expressibility and entangling capability, according to []. They are applicable to a `QuantumCircuit` object which contain parameterized gate operations.

A data encoding circuit is deemed good if it can well preserve the features in the original data and a good variational circuit should be able to well represent the solution space while maintaining a low circuit depth and number of parameters. In both cases, a circuit with the ability to generate states that are well representative of the Hilbert space is needed and this property is characterised by the expressibility of a circuit. The quantity can be expressed as the Kullback–Leibler divergence between the sampled state fidelity distribution of the parameterized circuit and the distribution of state fidelities generated by a parameterized circuit using that of Haar random states.

In the context of variational algorithms, potential advantages of generating highly entangled states with low-depth circuits include the ability to efficiently represent the solution space for data classification, and to capture nontrivial correlation in the quantum data. This is where the entangling capability of a circuit come in to play. The quantity can be expressed as the average Meyer Wallach measure of the sampled final states of the parameterized circuit.

The following code snippet demonstrates some examples how to calculate the expressibility and entangling capability of various parameterized circuits.

```
from quple.circuits.variational_circuits import ExcitationPreserving, RealAmplitudes, EfficientSU2
from quple.data_encoding import GeneralPauliEncoding
from quple import circuit_fidelity_plot, circuit_entangling_measure, circuit_expressibility_measure

A = GeneralPauliEncoding(feature_dimension=4, copies=2, paulis=['Z'])
B = GeneralPauliEncoding(feature_dimension=4, copies=2, paulis=['Z', 'ZZ'],
                         entangle_strategy='linear')
C = RealAmplitudes(n_qubit=4, copies=2, entangle_strategy='linear')
D = EfficientSU2(n_qubit=4, copies=2, entangle_strategy='linear')

# Calculate the expressibility measure for each of the circuits
print(circuit_expressibility_measure(A, samples=3000))
# output: 0.9377672194574366
print(circuit_expressibility_measure(B, samples=3000))
# output: 0.06422864349728435
print(circuit_expressibility_measure(C, samples=3000))
# output: 0.1825387273863395
print(circuit_expressibility_measure(D, samples=3000))
# output: 0.011474480876725163

# Calculate the entangling measure for each of the circuits
print(circuit_entangling_measure(A, samples=3000))
# output: 2.971446059546891e-15
print(circuit_entangling_measure(B, samples=3000))
# output: 0.4816055124378446
print(circuit_entangling_measure(C, samples=3000))
# output: 0.5585023165647457
print(circuit_entangling_measure(D, samples=3000))
# output: 0.7089928898427679

plot_A = circuit_fidelity_plot(A, samples=3000, bins=100)
plot_B = circuit_fidelity_plot(B, samples=3000, bins=100)
plot_C = circuit_fidelity_plot(C, samples=3000, bins=100)
plot_D = circuit_fidelity_plot(E, samples=3000, bins=100)
```

The plots of fidelity distributions of the parameterized circuits defined above in comparison to that of the Haar random states are shown in Figure 11.

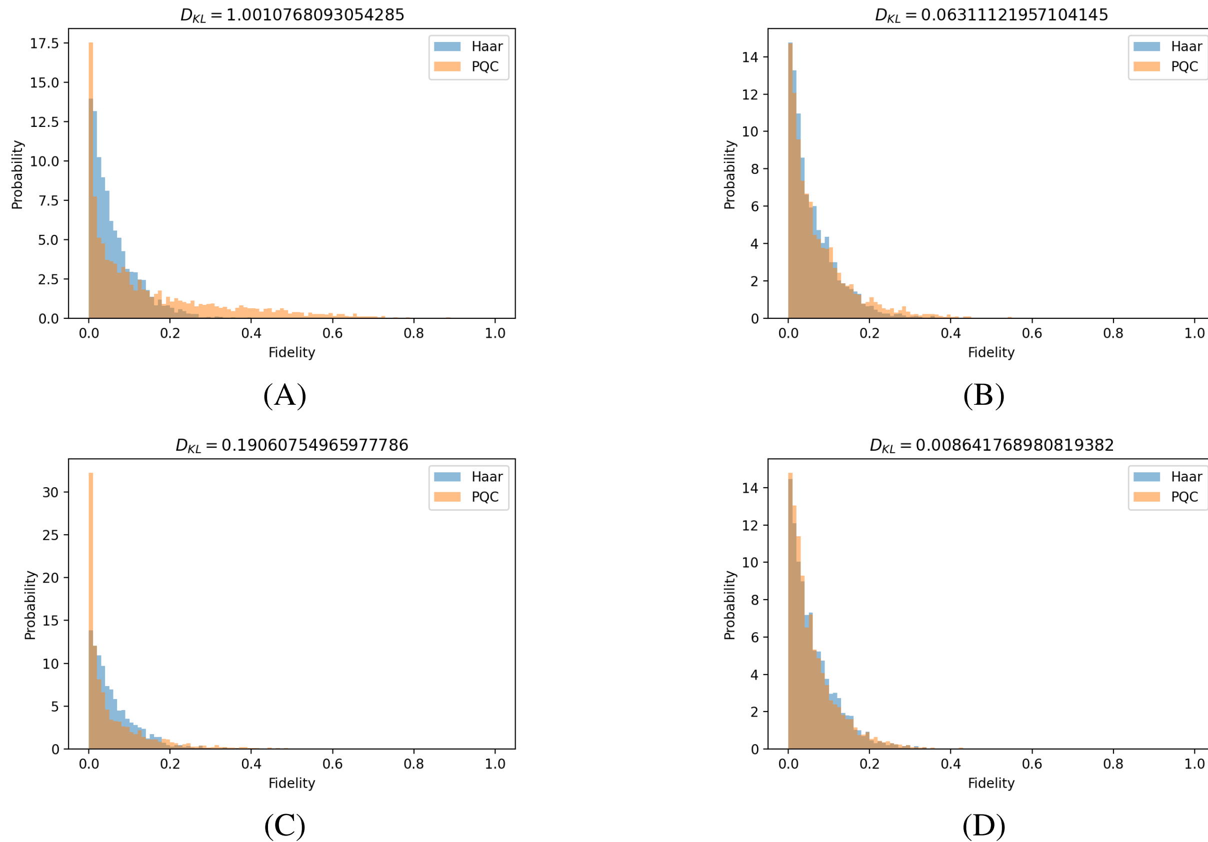



Figure 11: Plots of fidelity distributions of various parameterized quantum circuits. The distribution in orange is the fidelity distribution sampled from (A) the `GeneralPauliEncoding` circuit of 4 qubits and circuit depth 2 using Pauli string 'Z', (B) the `GeneralPauliEncoding` circuit of 4 qubits and circuit depth 2 using Pauli strings 'Z' and 'ZZ' with linear entanglement, (C) the `RealAmplitudes` circuit of 4 qubssits and circuit depth 2 with linear entanglement and (D) the `RealAmplitudes` circuit of 4 qubits and circuit depth 2 with linear entanglement. The distribution in blue is the fidelity distribution sampled from the Haar random states. Here $D_{KL}$ is the expressibility evaluated for the parameterized quantum circuit.

A Colab example notebook for the usage of circuit descriptors can be found at

`examples/Circuit_Descriptors_Walkthrough.ipynb`

# Application to High Energy Physics

The variational quantum classifier (VQC) implemented by the Quple library was tested on two of the Large Hadron Collider (LHC) high energy physics experiments, namely the $H \rightarrow \mu^+\mu^-$ and $ttH$ Higgs boson decay channels. The main goal of the classifier is to distinguish between signal and background events based on the kinematic variables of a collision event. The dataset for each of the experiments is preprocessed to reduce the feature dimension of each data point to suit the number of qubits used by the quantum classifier using the method of principal component anylsis (PCA). The performance of the VQC will be compared with common classical machine learning algorithms, namely the classical neural netowrk, the classical support vector machine (SVM)

and the classical boost decision tree (BDT) based on the same dataset with reduced feature dimension. The input data used in the VQC is additionaly rescaled to the range of [-1, +1] to fit the requirement of the data encoding circuit.

In both tests, the VQC model is configured with 5, 7 and 10 qubits corresponding to input data with feature dimension of 5, 7 and 10 respectively. The encoding circuit used is the `GeneralPauliEncoding` circuit with Pauli string 'Z' and self product as the encoding function. The variational circuit used is the `EfficientSU2` circuit with a readout qubit entangled to all input qubits via an XX parity gate operation raised to a parameterized power. A Pauli Z operator is acted on the readout qubit as measurement. An additional sigmoid activation function is applied to the output epecteation value from the readout qubit measurement. The Adam optimizer implemented by tensorflow Keras is used as the classical optimizer for tuning the parameters of the variational circuit during the training. An example Quple code for constructing the VQC model is shown below.

```
import tensorflow as tf
from quple.data_encoding import GeneralPauliEncoding
from quple.circuits.variational_circuits import EfficientSU2
from quple.classifiers import VQC

n_qubit = 5
n_event = 100
encoding_circuit = GeneralPauliEncoding(n_qubit, paulis='Z', copies = 2)
variational_circuit = EfficientSU2(n_qubit, copies=2, entangle_strategy='full')
                                   variational_circuit.add_readout('XX')

vqc = VQC(encoding_circuit, variational_circuit, activation='sigmoid',
          optimizer=tf.keras.optimizers.Adam(),
          metrics=['binary_accuracy','AUC'], loss='mse',
          readout=[variational_circuit.readout_measurement()])
```

For the classical classifiers, the model used for the classical neural network is implemented by the tensorflow keras library and consists of two dense layers with 64 and 16 nodes each using the relu activation function and a drop out rate of 0.1 and a final output layer with the sigmoid activation function. The models used for classical SVM and classical BDT are implemented by the sklearn and the xgboost libraries respectively. Some degree of hyperparameter tuning was performed on the classical SVM and classical BDT models only. Therefore, the results are expected to favor the classical SVM and the classical BDT models.

To quantify the performance of each classifier model, the Receriver Operating Characteric (ROC) curves in the plane of background rejection versus signal efficiency (i.e. true negative rate versus true negative rate) is used. Each ROC curve is characterized by its area-under-curve (auc) value which ranges between 0 and 1 with a higher value representing a better model. For each classifier model in each experiment, a total of 20 independent training runs are carried out. In each run, a total of $3k$ events (with signal to background ratio of 1:1) are randomly drawn from the dataset of a specific experiment with $k = 100, 200, 400, 800, 1600$. The $3k$ events are split equally into

the training, validation and test datasets. Only the training and validation datasets are visible to the model during training and the final model will be tested on the test dataset to obtain the predicted scores from which the true positive rates and false positive rates are calculated. The average true positive rates and false negative rates are then extrapolated from the 20 independent runs to obtain the average auc and its standard deviation.

Results from the two tests for the 5, 7 and 10 variable scenarios are shown in Figure 12, 13 and 14 respectively. It is observed that the performance of the VQC model is generally worse than the classical models especially when more qubits (variables) are used. This is partly due to the fact that the performance of the VQC model does not scale up well with the number of qubits used and can become unstable at times. However, it should be noted that the behaviour of a VQC model is highly dependent on the choice of the data encoding circuit and the variational circuit. Therefore, the above results should only be taken as a proof of concept and does not represent the full capability of the VQC.

The full results of the two tests are shown in Appendix Table  and Table .

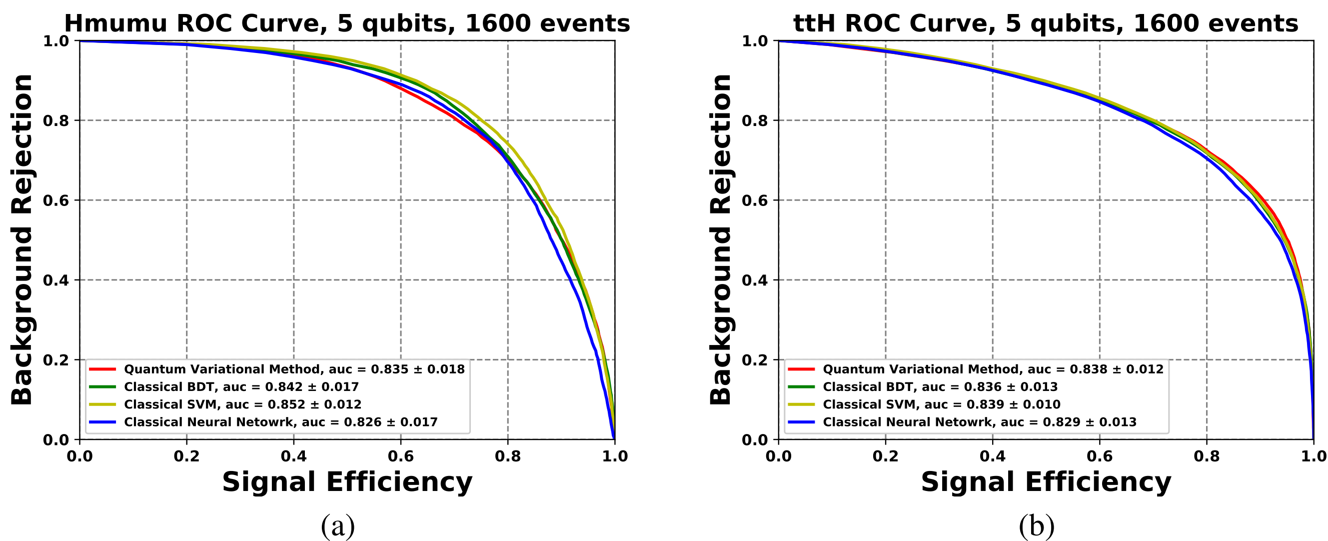
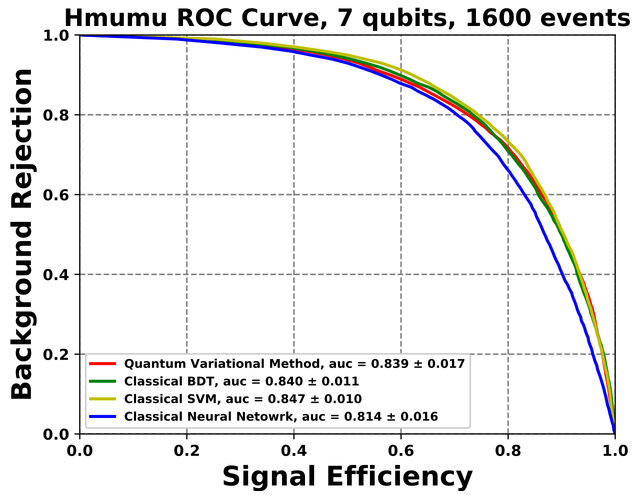
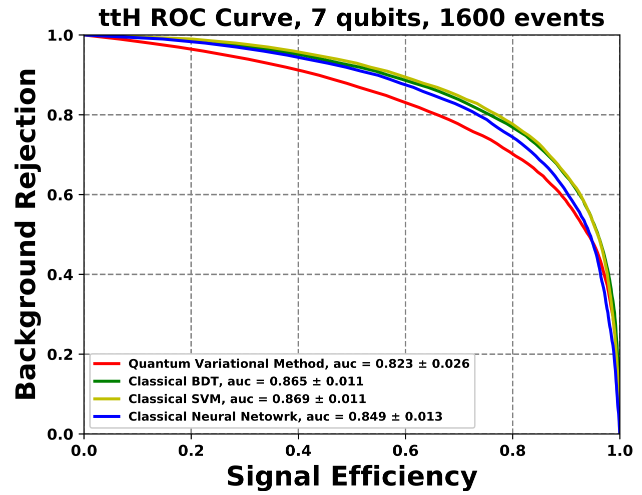



Figure 12: The ROC curves of the variational quantum classifier (red curves), the classical neural network (blue curves), the classical SVM (yellow curves), and the classical BDT (green curves) using 5 variables for (a) $H \to \mu^+\mu^-$ analysis with a dataset of 1600 events and (b) $ttH$ analysis with a dataset of 1600 events.

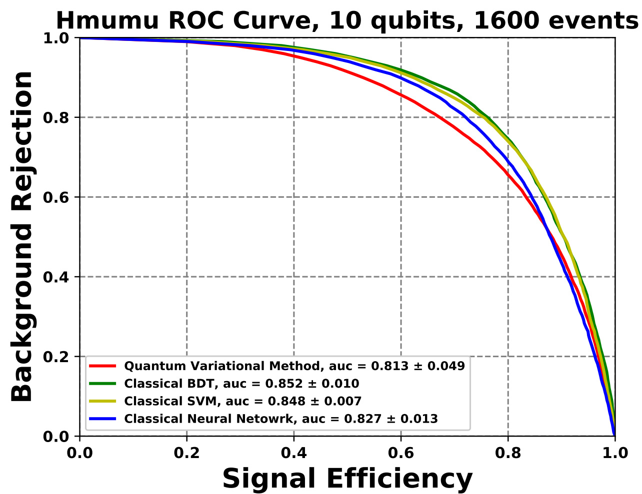



(a)

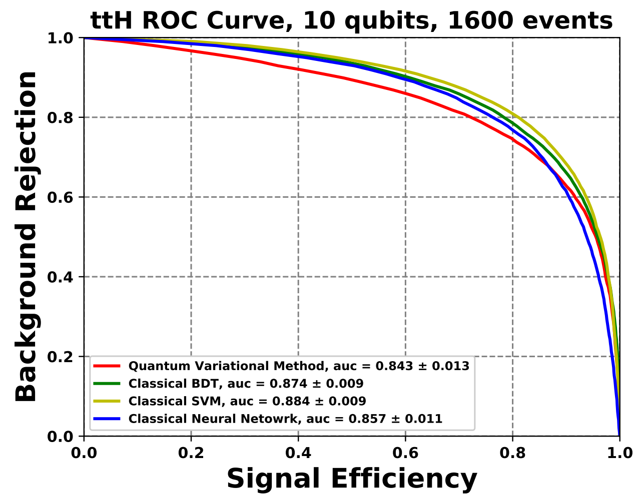



(b)

Figure 13: The ROC curves of the variational quantum classifier (red curves), the classical neural network (blue curves), the classical SVM (yellow curves), and the classical BDT (green curves) using 7 variables for (a) $H \to \mu^+\mu^-$ analysis with a dataset of 1600 events and (b) $ttH$ analysis with a dataset of 1600 events.




(a)




(b)

Figure 14: The ROC curves of the variational quantum classifier (red curves), the classical neural network (blue curves), the classical SVM (yellow curves), and the classical BDT (green curves) using 10 variables for (a) $H \to \mu^+\mu^-$ analysis with a dataset of 1600 events and (b) $ttH$ analysis with a dataset of 1600 events.

# Conclusion

The Quple package has implemented various components for building quantum machine learning algorithms based on the google Cirq and tensorflow quantum libraries. Some of the key components provided by Quple include the `QuantumCircuit` class for the construction of a general quantum circuit, `ParameterizedCircuit` class for the contruction of parameterized quantum circuit based on the circuit-centric design, various interaction maps for specifying the qubit connectivity of a given gate operation, the `EncodingCircuit` class for the construction of data encoding circuits, various data encoding functions (or feature maps) for mapping data vectors into the parameters of a gate operation, the `VQC` class for the construction of a variational quantum classifier model and various circuit descriptors for quantifying the properties of a parameterized circuit. Users can easily switch between different choices of individual components to customize and build their own quantum machine learning model. With Quple's modular and extensible architecture, users are also welcomed to contribute to the framework by providing new algorithms and components.

The variational quantum classifier implemented by Quple has been applied to two of the Large Hadron Collider (LHC) high energy physics experiments, namely the $H \to \mu^+\mu^-$ and $ttH$ Higgs boson decay channels. The results show that the classifier model can successfully learn and perform classification task. More detailed analysis is needed to explore the capability of the classifier.

Future plans for the Quple package include

1. the design of new variational circuits and data encoding circuits,
2. an interface for other popular quantum computing libraries such Qiskit and QuTip,
3. incorporation of noisy quantum simulation,
4. implementation of other quantum machine learning algorithms such as the quantum kernel method and quantum neural netowrk.

# References


[1] Cirq. `https://github.com/quantumlib/Cirq`, 2019.

[2] Y. B.-H. S. B. N. B. L. C. A. C. V. J. C. R. C. A. F. J. G. S. G. L. G. S. D. L. P. G. F. H. T. I. D. M. A. M. Z. M. R. M. G. N. P. N. A. P. M. P. A. R. J. S. J. S. J. S. K. T. M. T. S. W. J. W. Abraham Asfaw, Luciano Bello. Learn quantum computation using qiskit, 2020. URL `http://community.qiskit.org/textbook`.

[3] M. Benedetti, E. Lloyd, S. Sack, and M. Fiorentini. Parameterized quantum circuits as machine learning models. *Quantum Science and Technology*, 4, 10 2019. doi: 10.1088/2058-9565/ab4eb5.

[4] M. Broughton, G. Verdon, T. McCourt, A. Martinez, J. Yoo, S. Isakov, P. Massey, Y. Niu, R. Halavati, E. Peters, M. Leib, A. Skolik, M. Streif, D. Von Dollen, J. Mcclean, S. Boixo, D. Bacon, A. Ho, H. Neven, and M. Mohseni. Tensorflow quantum: A software framework for quantum machine learning. 03 2020.

[5] B. Foxen, C. Neill, A. Dunsworth, P. Roushan, B. Chiaro, A. Megrant, J. Kelly, Z. Chen, K. Satzinger, R. Barends, F. Arute, R. Babbush, D. Bacon, J. Bardin, S. Boixo, D. Buell, B. Burkett, Y. Chen, and J. Martinis. Demonstrating a continuous set of two-qubit gates for near-term quantum algorithms, 01 2020.

[6] A. Kandala, K. Temme, A. Corcoles, A. Mezzacapo, J. Chow, and J. Gambetta. Extending the computational reach of a noisy superconducting quantum processor, 05 2018.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[8] M. Schuld, A. Bocharov, K. Svore, and N. Wiebe. Circuit-centric quantum classifiers. *Physical Review A*, 101, 04 2018. doi: 10.1103/PhysRevA.101.032308.

# Appendix

| N Variable | N Event | VQC auc | SVM auc | BDT auc | NN auc |
|---|---|---|---|---|---|
| 5 | 100 | 0.823±0.078 | 0.780±0.062 | 0.801±0.054 | 0.775±0.053 |
| 5 | 200 | 0.827±0.029 | 0.825±0.031 | 0.813±0.032 | 0.810±0.045 |
| 5 | 400 | 0.830±0.025 | 0.844±0.022 | 0.832±0.020 | 0.821±0.025 |
| 5 | 800 | 0.835±0.020 | 0.848±0.016 | 0.839±0.009 | 0.827±0.022 |
| 5 | 1600 | 0.835±0.018 | 0.852±0.012 | 0.842±0.017 | 0.826±0.017 |
| 7 | 100 | 0.830±0.042 | 0.769±0.059 | 0.757±0.059 | 0.741±0.058 |
| 7 | 200 | 0.827±0.036 | 0.809±0.035 | 0.807±0.036 | 0.791±0.038 |
| 7 | 400 | 0.825±0.037 | 0.827±0.028 | 0.826±0.022 | 0.794±0.029 |
| 7 | 800 | 0.846±0.012 | 0.836±0.014 | 0.830±0.016 | 0.810±0.019 |
| 7 | 1600 | 0.839±0.017 | 0.847±0.010 | 0.840±0.011 | 0.814±0.016 |
| 10 | 100 | 0.807±0.103 | 0.795±0.047 | 0.794±0.051 | 0.764±0.060 |
| 10 | 200 | 0.809±0.060 | 0.833±0.025 | 0.805±0.028 | 0.783±0.037 |
| 10 | 400 | 0.817±0.066 | 0.835±0.020 | 0.828±0.028 | 0.803±0.019 |
| 10 | 800 | 0.805±0.060 | 0.846±0.014 | 0.842±0.014 | 0.821±0.016 |
| 10 | 1600 | 0.813±0.049 | 0.848±0.007 | 0.852±0.010 | 0.827±0.013 |

Table 2: Summary of the VQC result on the $H \to \mu^+\mu^-$ Higgs boson decay channels of the LHC high energy physics experiment.

| N Variable | N Event | VQC auc | SVM auc | BDT auc | NN auc |
|---|---|---|---|---|---|
| 5 | 100 | 0.779±0.075 | 0.795±0.043 | 0.768±0.053 | 0.776±0.069 |
| 5 | 200 | 0.780±0.047 | 0.808±0.036 | 0.794±0.031 | 0.795±0.046 |
| 5 | 400 | 0.820±0.025 | 0.818±0.020 | 0.806±0.020 | 0.810±0.023 |
| 5 | 800 | 0.812±0.026 | 0.835±0.016 | 0.820±0.019 | 0.817±0.019 |
| 5 | 1600 | 0.838±0.012 | 0.839±0.010 | 0.836±0.013 | 0.829±0.013 |
| 7 | 100 | 0.774±0.063 | 0.782±0.048 | 0.797±0.049 | 0.761±0.052 |
| 7 | 200 | 0.775±0.039 | 0.818±0.034 | 0.812±0.038 | 0.812±0.032 |
| 7 | 400 | 0.806±0.031 | 0.844±0.019 | 0.835±0.014 | 0.819±0.025 |
| 7 | 800 | 0.825±0.022 | 0.857±0.015 | 0.854±0.012 | 0.835±0.016 |
| 7 | 1600 | 0.823±0.026 | 0.869±0.011 | 0.865±0.011 | 0.849±0.013 |
| 10 | 100 | 0.756±0.082 | 0.806±0.037 | 0.798±0.055 | 0.780±0.053 |
| 10 | 200 | 0.807±0.041 | 0.834±0.028 | 0.828±0.029 | 0.800±0.034 |
| 10 | 400 | 0.818±0.033 | 0.851±0.021 | 0.848±0.018 | 0.835±0.023 |
| 10 | 800 | 0.829±0.020 | 0.868±0.013 | 0.863±0.015 | 0.842±0.013 |
| 10 | 1600 | 0.843±0.013 | 0.884±0.009 | 0.874±0.009 | 0.857±0.011 |

Table 3: Summary of the VQC result on the $ttH$ Higgs boson decay channels of the LHC high energy physics experiment.