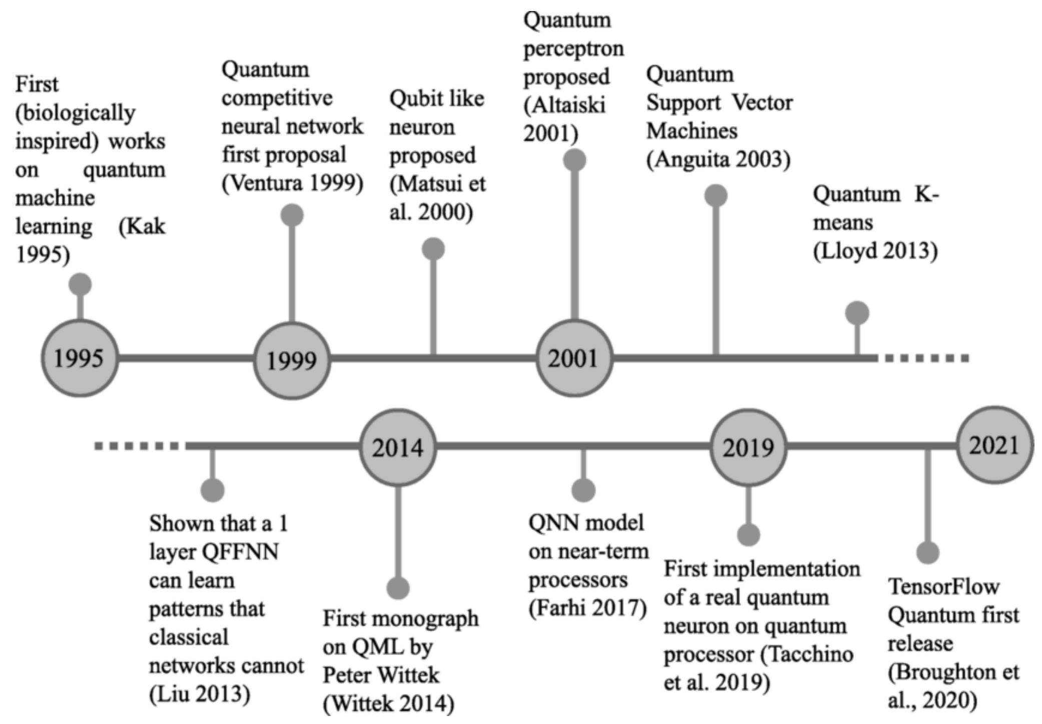
Quantum Machine Learning

Introduction:

Quantum Machine Learning (QML) combines principles from quantum computing and machine learning to enhance computational efficiency and tackle complex problems.


1995
First (biologically inspired) works on quantum machine learning (Kak 1995)
1999
Quantum competitive neural network first proposal (Ventura 1999)
Qubit like neuron proposed (Matsui et al. 2000)
2001
Quantum perceptron proposed (Altaiski 2001)
Quantum Support Vector Machines (Anguita 2003)
Quantum K-means (Lloyd 2013)
Shown that a 1 layer QFFNN can learn patterns that classical networks cannot (Liu 2013)
2014
First monograph on QML by Peter Wittek (Wittek 2014)
QNN model on near-term processors (Farhi 2017)
2019
First implementation of a real quantum neuron on quantum processor (Tacchino et al. 2019)
2021
TensorFlow Quantum first release (Broughton et al., 2020)

The field of QML traces its origins back to the early days of quantum computing, when researchers began exploring how quantum algorithms could solve certain types of problems faster than classical algorithms. Early breakthroughs, like Shor's algorithm for prime factorization and Grover's algorithm for searching unsorted databases, highlighted the potential of quantum speedup. As machine learning gained popularity in the early 2000s, researchers saw an opportunity to apply quantum computing's power to optimize and scale ML algorithms.

By the 2010s, the convergence of these fields took off, with advancements in quantum hardware sparking interest in developing quantum-enhanced versions of ML algorithms. Today, QML aims to leverage quantum systems to process data faster and more efficiently, with applications ranging from data classification and clustering to generative models and neural networks, all of which hold potential to outperform classical ML methods on specific, high-dimensional tasks.

Foundational Concepts in Quantum Computing for Machine Learning

1. Quantum Gates, Circuits, and Measurements

Quantum Gates:

Quantum gates are the building blocks of quantum circuits, analogous to classical logic gates. They manipulate qubits through unitary transformations. Key gates include:

Pauli Gates (X, Y, Z): Basic single-qubit operations.

Hadamard Gate (H): Creates superposition states.

CNOT Gate: A two-qubit gate introducing entanglement.

Rotation Gates (Rx, Ry, Rz): Rotate a qubit around axes of the Bloch sphere.

Quantum Circuits:

A quantum circuit is a sequence of quantum gates applied to qubits, designed to perform a specific computation.

Example: A circuit with entangled states used for quantum machine learning tasks like encoding data or training quantum models.

Measurement:

At the end of a quantum computation, qubits are measured to extract classical outcomes. Measurements collapse quantum states into classical bits, governed by probability amplitudes.

2. Quantum Algorithms and Their Relevance to ML

Quantum Algorithms:

Algorithms exploiting quantum mechanics to achieve speed-ups.

Examples include:

Quantum Fourier Transform (QFT): Basis for algorithms like Shor's for prime factorization.

Grover's Algorithm: Quadratic speed-up for unstructured search, useful for optimization tasks in ML.

Quantum Approximate Optimization Algorithm (QAOA): Addresses optimization problems, aiding ML in finding optimal parameters.

Relevance to ML:

Data Encoding: Quantum states encode classical data, enabling efficient representation of high-dimensional data.

Kernel Methods: Quantum kernels exploit quantum states for enhanced feature mapping in ML models.

Variational Quantum Algorithms: Hybrid algorithms, like the Variational Quantum Eigensolver (VQE) or Variational Quantum Classifiers (VQC), are used for optimization problems and learning tasks.

3. Key Quantum Computing Frameworks

IBM Qiskit:

Open-source framework for building quantum circuits, running simulations, and executing on IBM quantum hardware.

Features tools for quantum ML, such as Qiskit Machine Learning for implementing quantum classifiers and regression models.

Google Cirq:

Focuses on near-term quantum computers, with capabilities to design and optimize quantum circuits.

Supports hybrid approaches combining classical and quantum workflows.

Xanadu PennyLane:

Integrates quantum and classical machine learning seamlessly.

Specializes in variational quantum circuits and supports frameworks like TensorFlow and PyTorch.

Enables quantum differentiable programming for optimizing quantum models.

QML Demos using Pennylane:

1.Learning Shallow Quantum Circuits with Local Operations

Introduction

This paper explores the learning of shallow quantum circuits through local operations, focusing on the implications for quantum machine learning and computational efficiency. We present a framework for understanding how local operations can be utilised to simplify the learning process of quantum circuits, thereby enhancing their applicability in practical scenarios.

Quantum circuits are essentially the instructions or "programs" used to control a quantum computer. These circuits consist of quantum gates that manipulate qubits (quantum bits). Think of quantum gates like the operations (such as addition, subtraction, etc.) in classical computers, but they work with quantum properties like superposition and entanglement.

What are Shallow Quantum Circuits?

A shallow quantum circuit is a type of quantum circuit with fewer layers of quantum gates. The term shallow refers to the depth of the circuit, meaning that it requires fewer operations (layers of gates) to complete the computation.

Why are Shallow Quantum Circuits Important in quantum machine learning?

In quantum machine learning (QML), shallow quantum circuits play an important role for several reasons. These circuits, characterised by a limited depth and fewer quantum gates, are crucial when building practical quantum algorithms that can be run on near-term quantum devices, which are often noisy and have limited qubit counts.

1. Limited Coherence Time of Qubits:
    - Shallow circuits reduce the time qubits are exposed to operations, minimising the risk of decoherence and error accumulation.
2. Error Reduction:
    - Fewer gates mean fewer opportunities for errors during computation, ensuring more reliable results in quantum machine learning tasks.
3. Scalability with Current Quantum Hardware:
    - Shallow circuits are well-suited for today's Noisy Intermediate-Scale Quantum (NISQ) devices, which have a limited number of qubits and high noise levels.
4. Faster Execution for Real-World Problems:

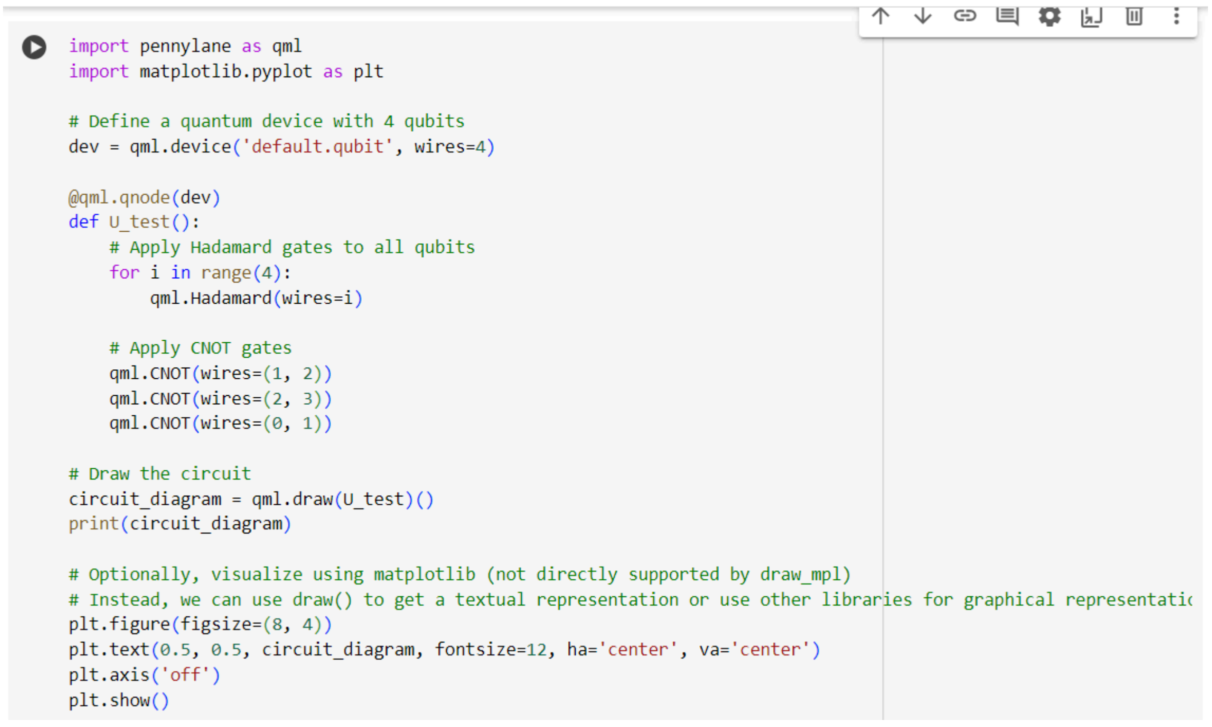
- Shorter quantum circuits result in faster execution times, enabling real-time applications like online learning and adaptive models.

5. Quantum Advantage in Machine Learning:
    - Shallow circuits provide the potential for quantum speedup in specific tasks, offering a quantum advantage in areas like feature mapping and pattern recognition.
6. Variational Quantum Algorithms (VQAs):
    - Shallow circuits are used in VQAs for tasks like quantum neural networks and optimization, making them efficient for training and inference on quantum devices.
7. Generalization and Flexibility:
    - Shallow quantum circuits are simpler and less likely to overfit, providing better generalisation in quantum machine learning models.

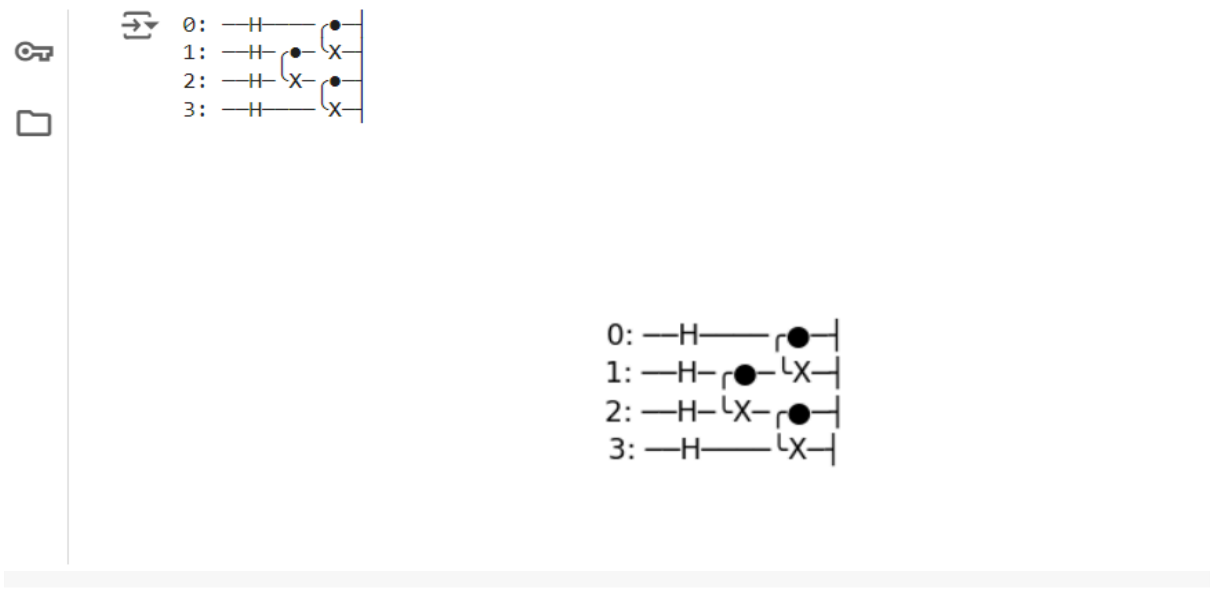
Simulation Experiments

1) Visualisation of Quantum Circuits:

This code demonstrates how to construct and visualize a quantum circuit using Hadamard and CNOT gates on a 4-qubit system with the PennyLane framework. The circuit is first defined and then drawn using both a textual representation and a graphical one using Matplotlib.



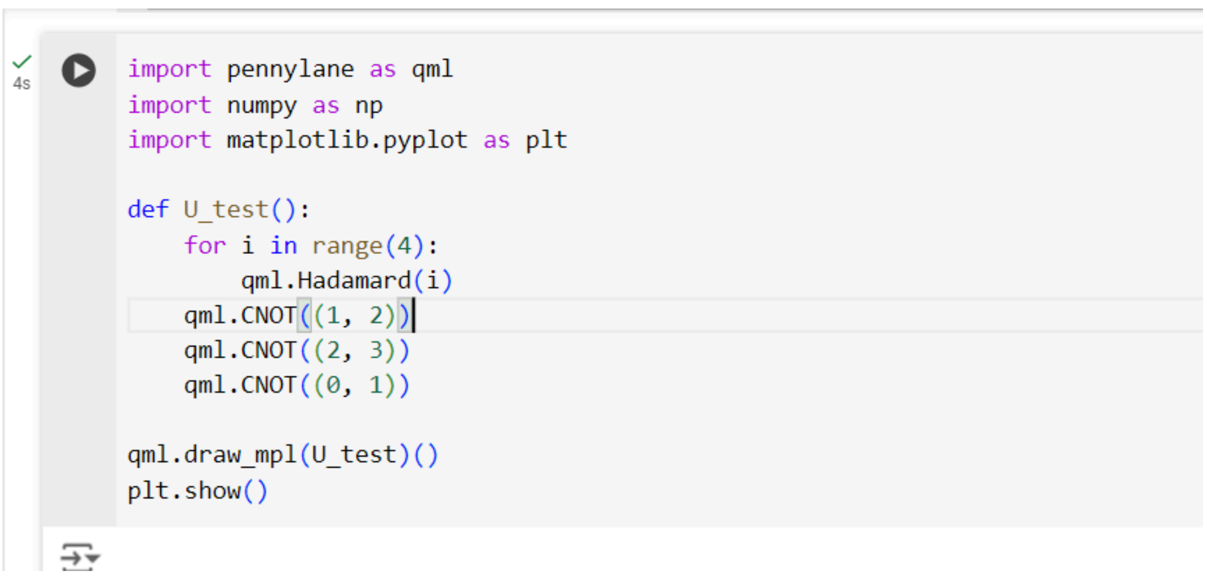
```
import pennylane as qml
import matplotlib.pyplot as plt

# Define a quantum device with 4 qubits
dev = qml.device('default.qubit', wires=4)

@qml.qnode(dev)
def U_test():
    # Apply Hadamard gates to all qubits
    for i in range(4):
        qml.Hadamard(wires=i)

    # Apply CNOT gates
    qml.CNOT(wires=(1, 2))
    qml.CNOT(wires=(2, 3))
    qml.CNOT(wires=(0, 1))

# Draw the circuit
circuit_diagram = qml.draw(U_test)()
print(circuit_diagram)

# Optionally, visualize using matplotlib (not directly supported by draw_mpl)
# Instead, we can use draw() to get a textual representation or use other libraries for graphical representatic
plt.figure(figsize=(8, 4))
plt.text(0.5, 0.5, circuit_diagram, fontsize=12, ha='center', va='center')
plt.axis('off')
plt.show()
```



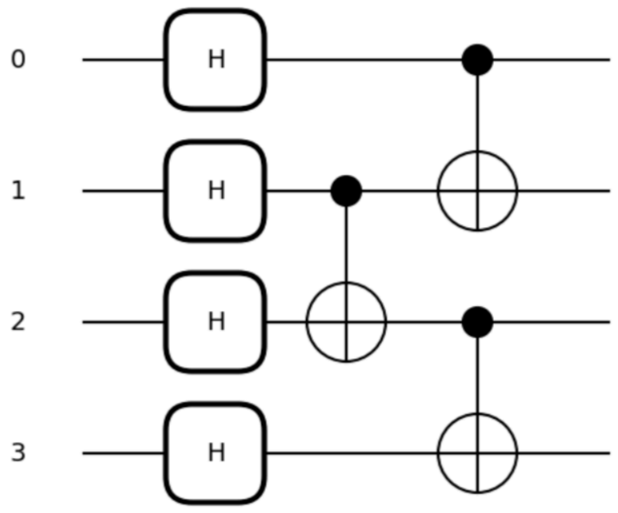
```
0: ──H────╭●─┤
1: ──H─╭●─╰X─┤
2: ──H─╰X─╭●─┤
3: ──H────╰X─┤
```

2) LOCAL INVERSIONS:

The code demonstrates how quantum circuits are created using the PennyLane library, which is commonly used in Quantum Machine Learning (QML). The circuit applies Hadamard gates to create superposition and CNOT gates to entangle qubits, which are fundamental concepts in quantum computing. In QML, these quantum operations can be used to enhance classical machine learning models by leveraging quantum properties like superposition and entanglement for more efficient processing and problem-solving, combining both quantum and classical methods for better performance.

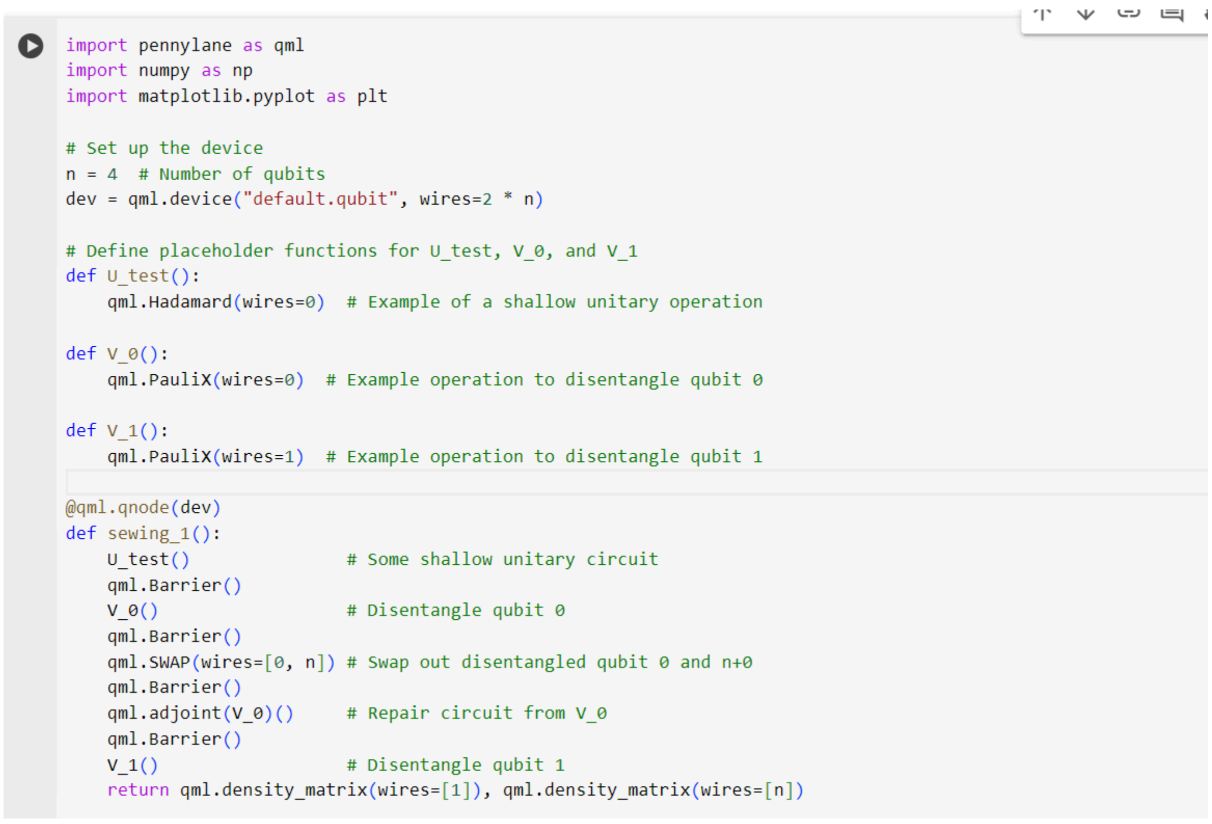


```python
import pennylane as qml
import numpy as np
import matplotlib.pyplot as plt

def U_test():
    for i in range(4):
        qml.Hadamard(i)
    qml.CNOT((1, 2))
    qml.CNOT((2, 3))
    qml.CNOT((0, 1))

qml.draw_mpl(U_test)()
plt.show()
```

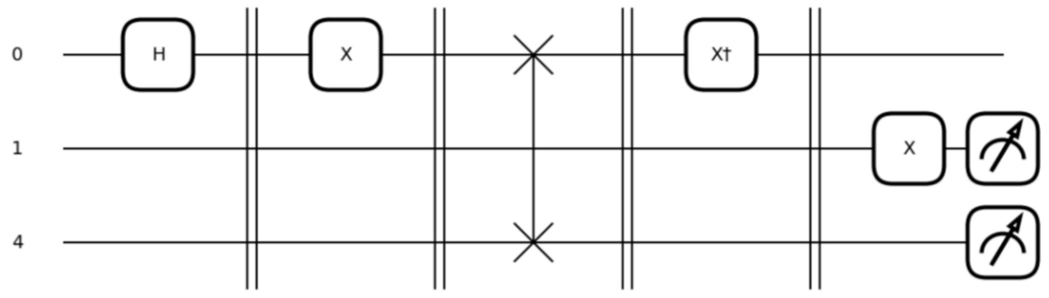


3) Circuit Sewing: Quantum Entanglement and Disentanglement

The following code implements a quantum circuit to demonstrate **circuit sewing**, a technique where entangled qubits are disentangled and then swapped to test their states. The circuit uses **Hadamard**, **Pauli-X**, and **SWAP** gates to perform these operations and visualise the evolution of the quantum states.


```python
import pennylane as qml
import numpy as np
import matplotlib.pyplot as plt

# Set up the device
n = 4  # Number of qubits
dev = qml.device("default.qubit", wires=2 * n)

# Define placeholder functions for U_test, V_0, and V_1
def U_test():
    qml.Hadamard(wires=0)  # Example of a shallow unitary operation

def V_0():
    qml.PauliX(wires=0)  # Example operation to disentangle qubit 0

def V_1():
    qml.PauliX(wires=1)  # Example operation to disentangle qubit 1

@qml.qnode(dev)
def sewing_1():
    U_test()               # Some shallow unitary circuit
    qml.Barrier()
    V_0()                  # Disentangle qubit 0
    qml.Barrier()
    qml.SWAP(wires=[0, n]) # Swap out disentangled qubit 0 and n+0
    qml.Barrier()
    qml.adjoint(V_0)()     # Repair circuit from V_0
    qml.Barrier()
    V_1()                  # Disentangle qubit 1
    return qml.density_matrix(wires=[1]), qml.density_matrix(wires=[n])
```

```
# Draw the circuit
qml.draw_mpl(sewing_1)()
plt.show()

# Run the circuit and print results
r1, rn = sewing_1()
print("Sewing qubit 1")
print(f"rho_1 = |0⟩⟨0| {np.allclose(r1, np.array([[1, 0], [0, 0]]))}")
print(f"rho_0+n = |0⟩⟨0| {np.allclose(rn, np.array([[1, 0], [0, 0]]))}")
```

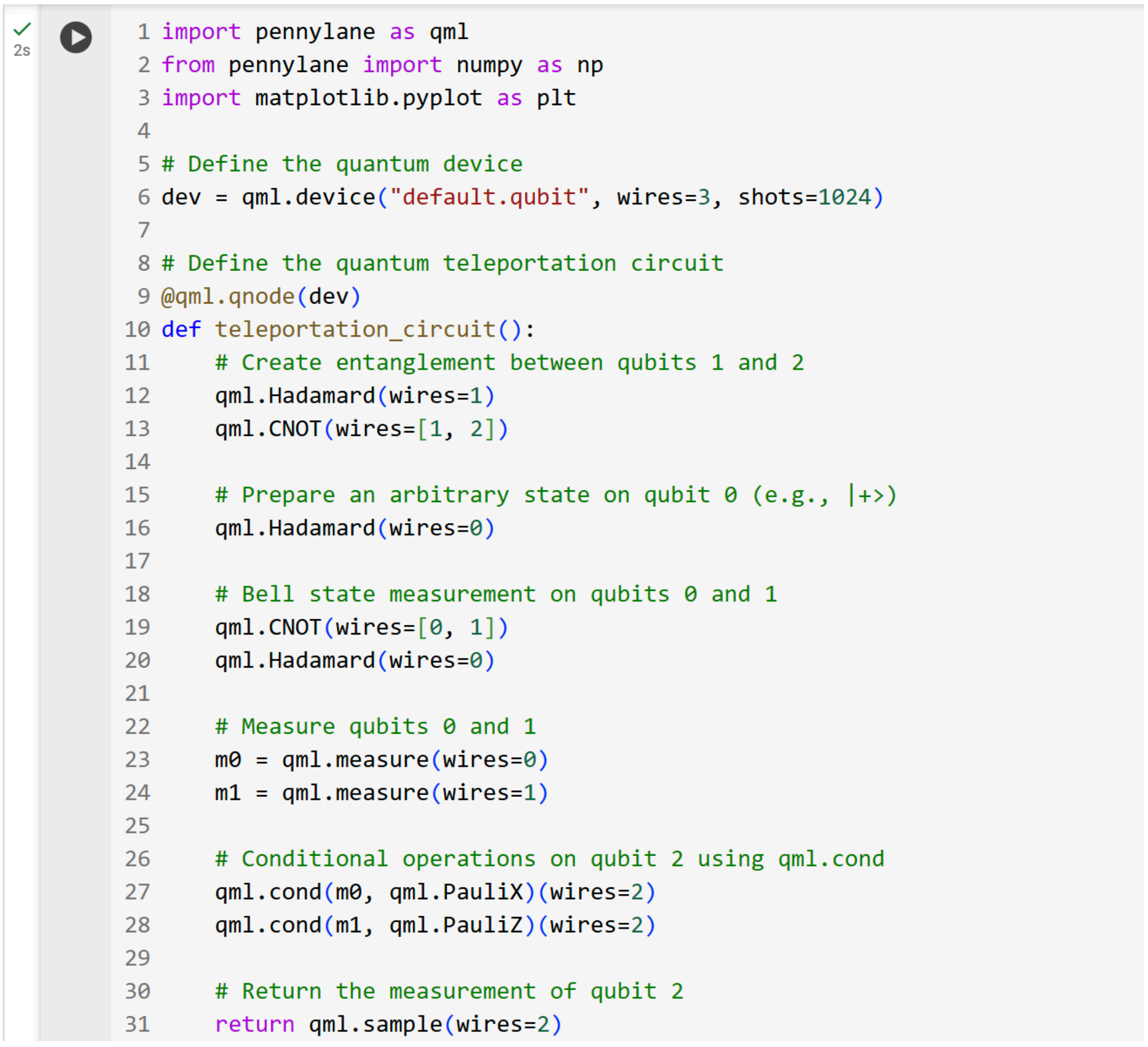


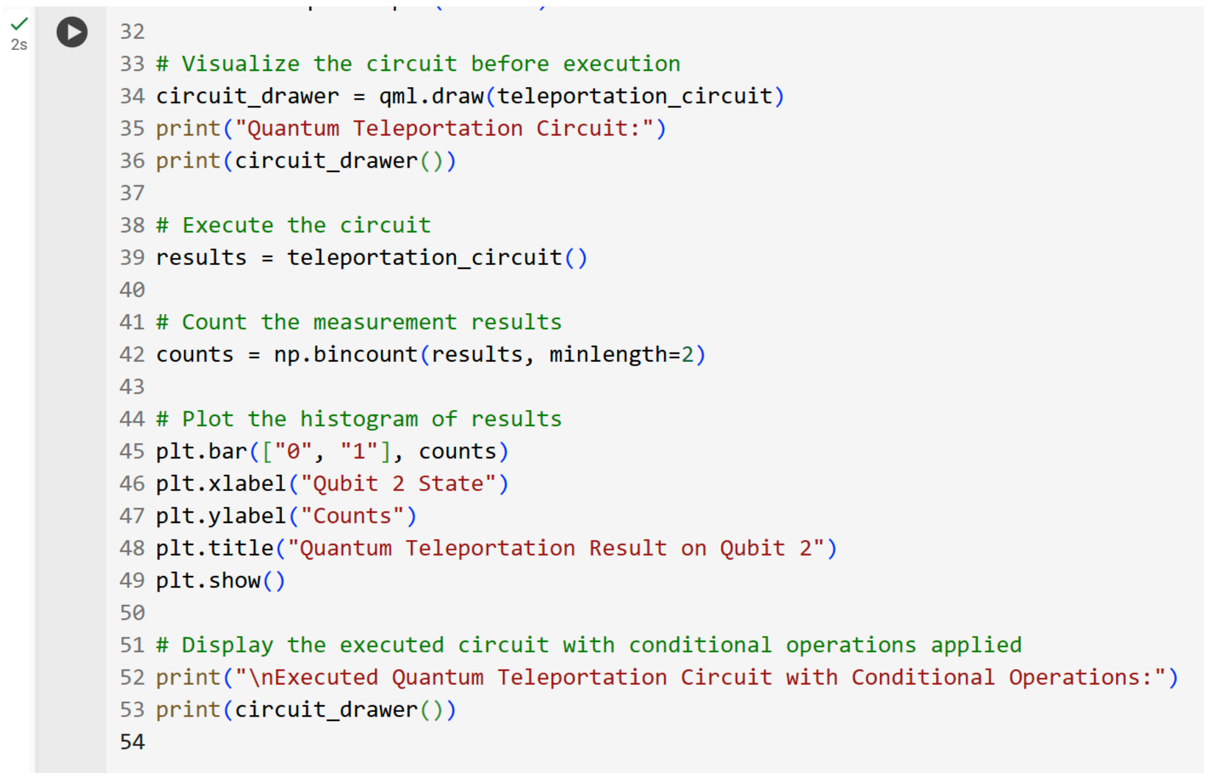
```
Sewing qubit 1
rho_1 = |0⟩⟨0| False
rho_0+n = |0⟩⟨0| False
```

4) Quantum Teleportation

This code implements a quantum teleportation protocol using PennyLane. It utilizes a shallow circuit with 3 qubits: one for the state to teleport, and two for entanglement and correction. The circuit starts by creating entanglement between two qubits, followed by preparing an arbitrary state to teleport. A Bell state measurement is performed, and the results are used to apply conditional corrections to the third qubit, ensuring it matches the original state. The use of shallow depth (6 gates) and minimal conditional operations makes it efficient for near-term quantum devices. The results are visualized in a histogram, confirming successful state teleportation. The design leverages minimal gate complexity and efficient classical communication, demonstrating the feasibility of quantum teleportation with shallow circuits

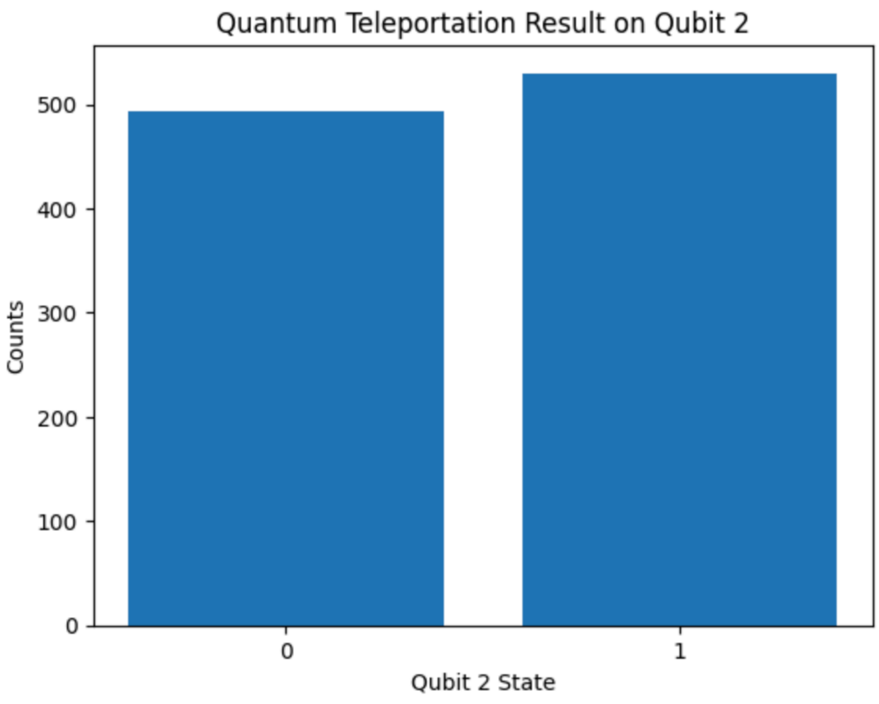
- The code demonstrates the **quantum teleportation protocol** using 3 qubits.
- Qubit 0's state is teleported to qubit 2, using entanglement between qubits 1 and 2, and conditional corrections based on measurements.
- The use of conditional gates (`qml.cond`) ensures that qubit 2 is corrected to match the original state of qubit 0.
- The plot shows the result, indicating the successful teleportation of the state.


```python
import pennylane as qml
from pennylane import numpy as np
import matplotlib.pyplot as plt

# Define the quantum device
dev = qml.device("default.qubit", wires=3, shots=1024)

# Define the quantum teleportation circuit
@qml.qnode(dev)
def teleportation_circuit():
    # Create entanglement between qubits 1 and 2
    qml.Hadamard(wires=1)
    qml.CNOT(wires=[1, 2])

    # Prepare an arbitrary state on qubit 0 (e.g., |+>)
    qml.Hadamard(wires=0)

    # Bell state measurement on qubits 0 and 1
    qml.CNOT(wires=[0, 1])
    qml.Hadamard(wires=0)

    # Measure qubits 0 and 1
    m0 = qml.measure(wires=0)
    m1 = qml.measure(wires=1)

    # Conditional operations on qubit 2 using qml.cond
    qml.cond(m0, qml.PauliX)(wires=2)
    qml.cond(m1, qml.PauliZ)(wires=2)

    # Return the measurement of qubit 2
    return qml.sample(wires=2)
```



```python

# Visualize the circuit before execution
circuit_drawer = qml.draw(teleportation_circuit)
print("Quantum Teleportation Circuit:")
print(circuit_drawer())

# Execute the circuit
results = teleportation_circuit()

# Count the measurement results
counts = np.bincount(results, minlength=2)

# Plot the histogram of results
plt.bar(["0", "1"], counts)
plt.xlabel("Qubit 2 State")
plt.ylabel("Counts")
plt.title("Quantum Teleportation Result on Qubit 2")
plt.show()

# Display the executed circuit with conditional operations applied
print("\nExecuted Quantum Teleportation Circuit with Conditional Operations:")
print(circuit_drawer())

```

```
Quantum Teleportation Circuit:
0: ──H────╭●──H─┤↗├───────────┤       
1: ──H─╭●─╰X────║──┤↗├────────┤       
2: ────╰X───────║───║──X──Z──┤  Sample
                ╚═══║══╝  ║
                    ╚═════╝
```



```
Executed Quantum Teleportation Circuit with Conditional Operations:
0: ──H────╭●──H─┤↗├───────────┤       
1: ──H─╭●─╰X────║──┤↗├────────┤       
2: ────╰X───────║───║──X──Z──┤  Sample
                ╚═══║══╝  ║
                    ╚═════╝
```

2.Generalization in QML from few training data

Introduction

In this tutorial, we learn about the generalization capabilities of quantum machine learning models. For the example of a Quantum Convolutional Neural Network(QCNN), we show how its generalization error behaves as a function of the number of training samples.

What is generalisation in QML?

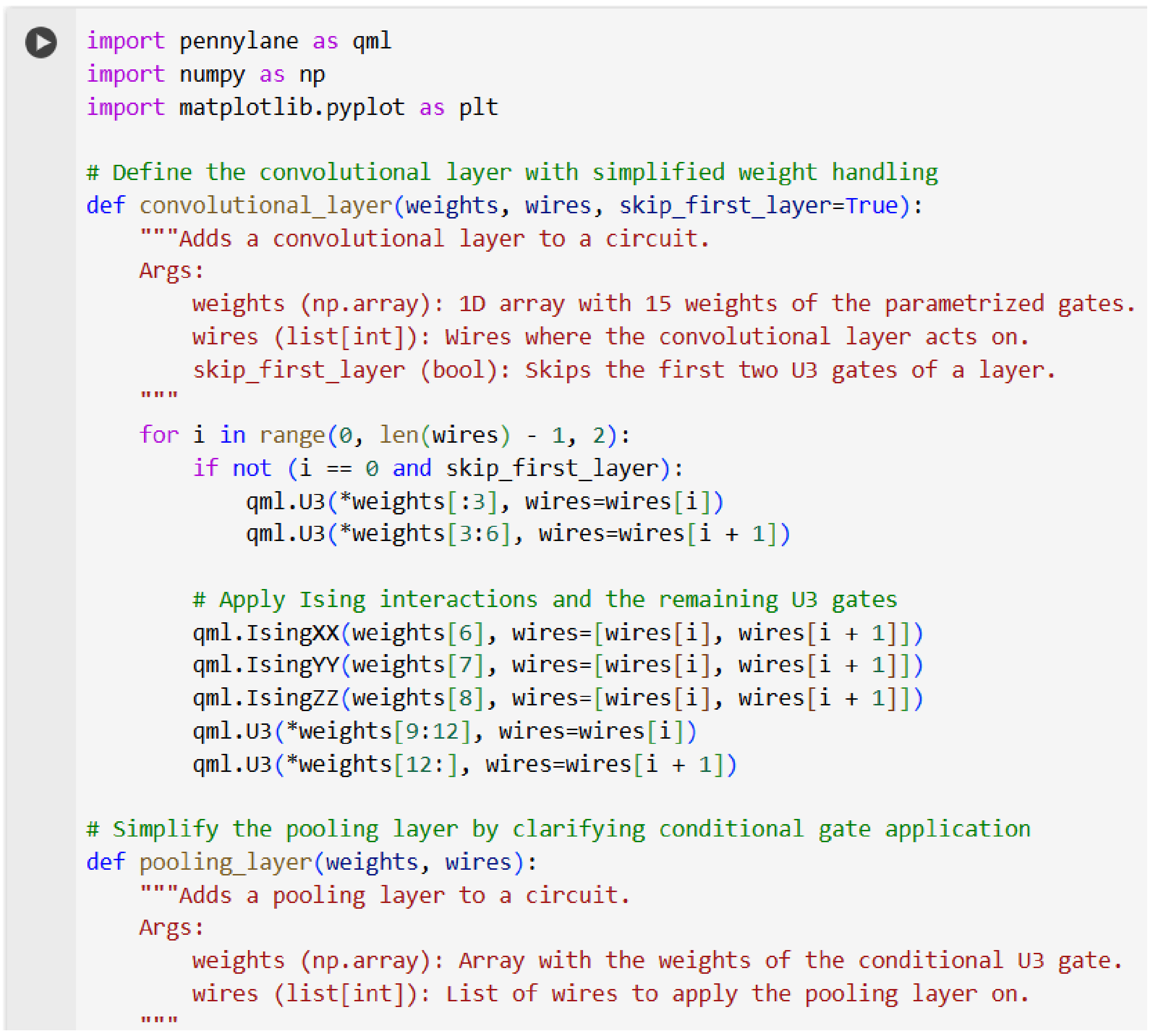
In machine learning, generalization refers to a model's ability to apply learned patterns to new, unseen data. Limited training data often results in a trade-off: a low-bias model that fits training data perfectly may overfit, leading to high variance and poor generalization. Conversely, a model with higher bias may generalize better but might not capture all patterns. Techniques like regularization help optimize this trade-off, improving performance across different data distributions by reducing the generalization error.

In quantum machine learning, generalization assesses a model's performance on new data, measured by the generalization error, or the gap between true expected loss and training loss. A QML model encodes classical data into quantum states, processes it via parameterized gates, and evaluates predictions through measurement. More parameters can lower training error but may increase generalization error. However, quantum convolutional neural networks (QCNNs) reduce this error by parameter-sharing, improving scaling with larger data, making them efficient for tasks with limited training data.

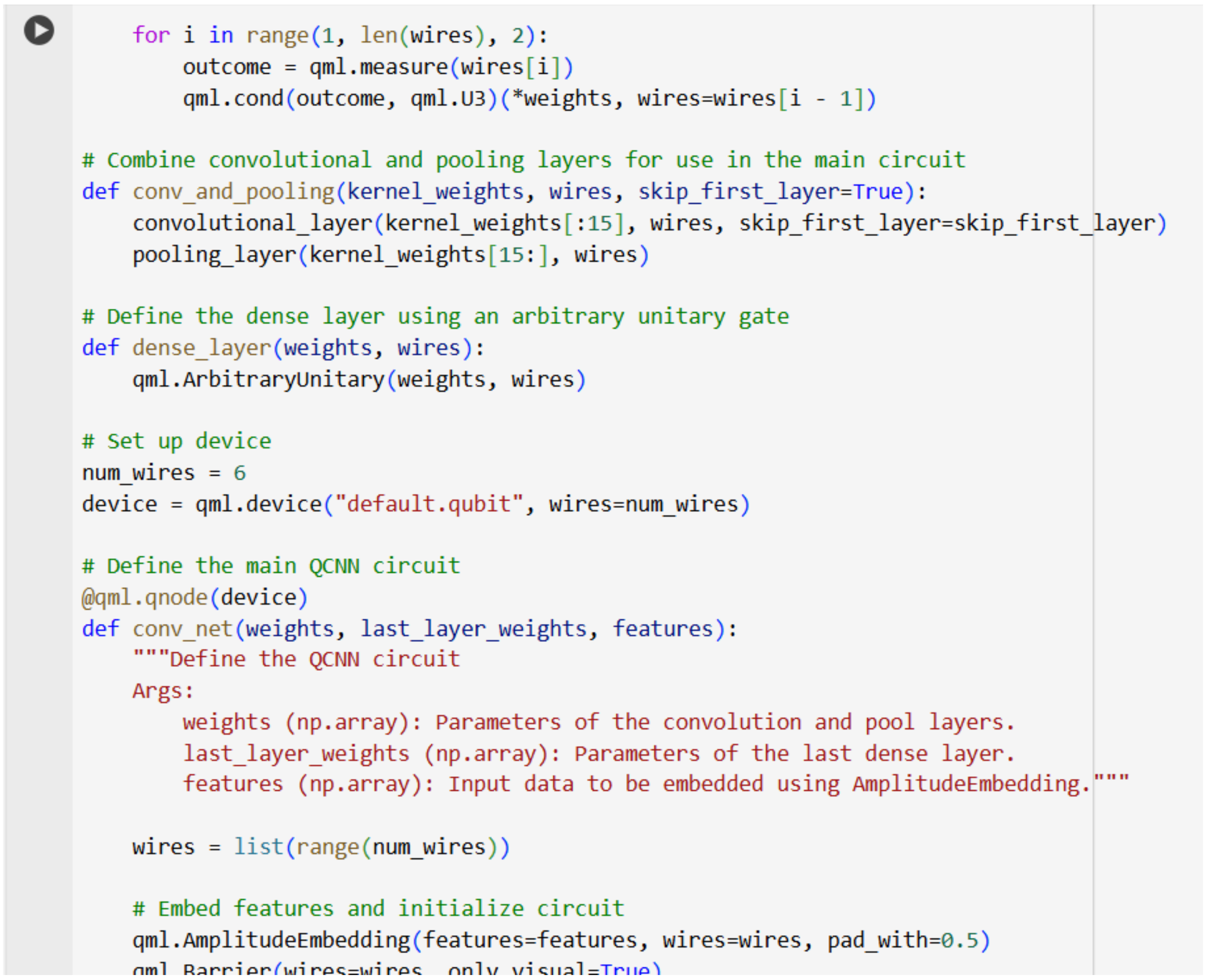
Quantum Convolutional Neural Network

Quantum Convolutional Neural Networks (QCNNs) aim to replicate the structure and benefits of classical CNNs within quantum circuits. In classical CNNs, convolutional layers use small filters to extract local features from data, while pooling layers reduce dimensionality and make the model invariant to certain transformations. In QCNNs, a similar approach is taken where convolutional layers correlate qubits locally with two-qubit unitaries, and pooling layers reduce feature dimensions using conditioned single-qubit unitaries. This structure helps QCNNs extract quantum-relevant features efficiently.

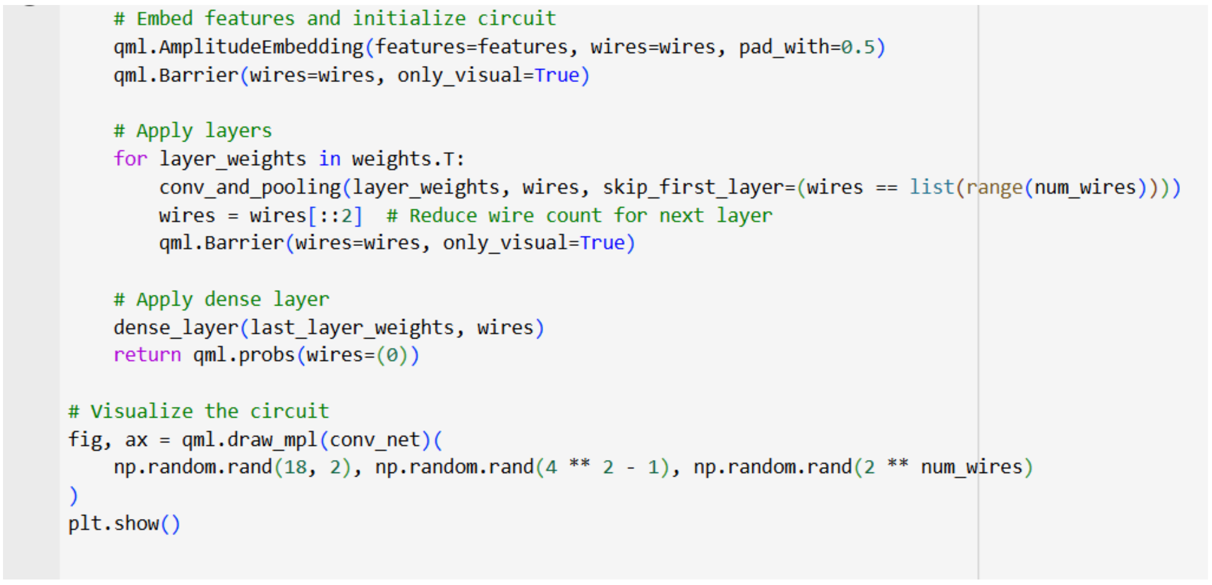
Quantum Convolutional Neural Network (QCNN) implementation using PennyLane



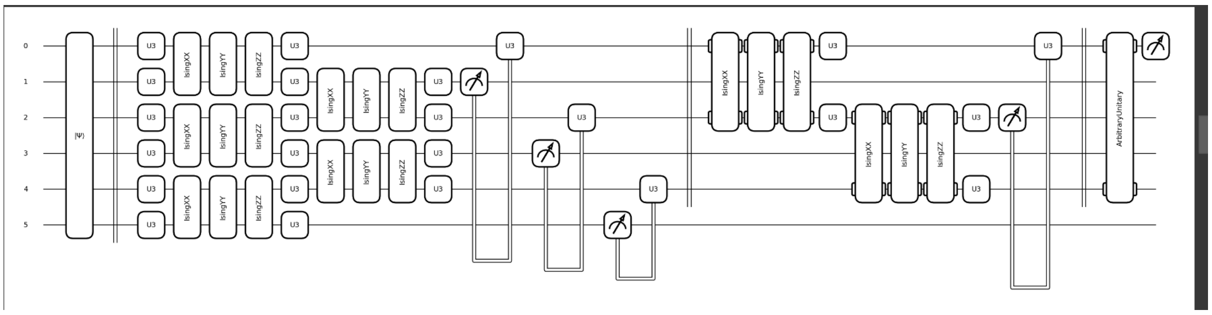
```python
import pennylane as qml
import numpy as np
import matplotlib.pyplot as plt

# Define the convolutional layer with simplified weight handling
def convolutional_layer(weights, wires, skip_first_layer=True):
    """Adds a convolutional layer to a circuit.
    Args:
        weights (np.array): 1D array with 15 weights of the parametrized gates.
        wires (list[int]): Wires where the convolutional layer acts on.
        skip_first_layer (bool): Skips the first two U3 gates of a layer.
    """
    for i in range(0, len(wires) - 1, 2):
        if not (i == 0 and skip_first_layer):
            qml.U3(*weights[:3], wires=wires[i])
            qml.U3(*weights[3:6], wires=wires[i + 1])

        # Apply Ising interactions and the remaining U3 gates
        qml.IsingXX(weights[6], wires=[wires[i], wires[i + 1]])
        qml.IsingYY(weights[7], wires=[wires[i], wires[i + 1]])
        qml.IsingZZ(weights[8], wires=[wires[i], wires[i + 1]])
        qml.U3(*weights[9:12], wires=wires[i])
        qml.U3(*weights[12:], wires=wires[i + 1])

# Simplify the pooling layer by clarifying conditional gate application
def pooling_layer(weights, wires):
    """Adds a pooling layer to a circuit.
    Args:
        weights (np.array): Array with the weights of the conditional U3 gate.
        wires (list[int]): List of wires to apply the pooling layer on.
    """
```



```python
    for i in range(1, len(wires), 2):
        outcome = qml.measure(wires[i])
        qml.cond(outcome, qml.U3)(*weights, wires=wires[i - 1])

# Combine convolutional and pooling layers for use in the main circuit
def conv_and_pooling(kernel_weights, wires, skip_first_layer=True):
    convolutional_layer(kernel_weights[:15], wires, skip_first_layer=skip_first_layer)
    pooling_layer(kernel_weights[15:], wires)

# Define the dense layer using an arbitrary unitary gate
def dense_layer(weights, wires):
    qml.ArbitraryUnitary(weights, wires)

# Set up device
num_wires = 6
device = qml.device("default.qubit", wires=num_wires)

# Define the main QCNN circuit
@qml.qnode(device)
def conv_net(weights, last_layer_weights, features):
    """Define the QCNN circuit
    Args:
        weights (np.array): Parameters of the convolution and pool layers.
        last_layer_weights (np.array): Parameters of the last dense layer.
        features (np.array): Input data to be embedded using AmplitudeEmbedding."""

    wires = list(range(num_wires))

    # Embed features and initialize circuit
    qml.AmplitudeEmbedding(features=features, wires=wires, pad_with=0.5)
    qml.Barrier(wires=wires, only_visual=True)
```



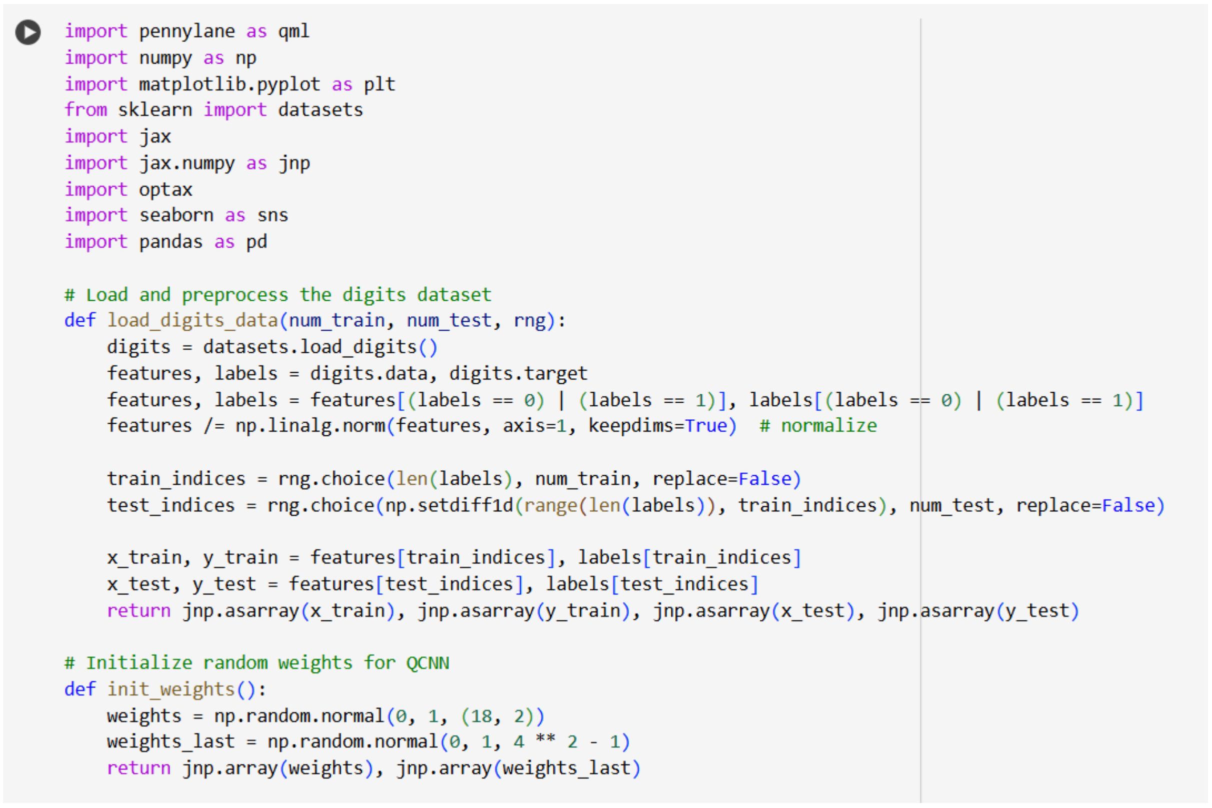
```python
    # Embed features and initialize circuit
    qml.AmplitudeEmbedding(features=features, wires=wires, pad_with=0.5)
    qml.Barrier(wires=wires, only_visual=True)

    # Apply layers
    for layer_weights in weights.T:
        conv_and_pooling(layer_weights, wires, skip_first_layer=(wires == list(range(num_wires))))
        wires = wires[::2]  # Reduce wire count for next layer
        qml.Barrier(wires=wires, only_visual=True)

    # Apply dense layer
    dense_layer(last_layer_weights, wires)
    return qml.probs(wires=(0))

# Visualize the circuit
fig, ax = qml.draw_mpl(conv_net)(
    np.random.rand(18, 2), np.random.rand(4 ** 2 - 1), np.random.rand(2 ** num_wires)
)
plt.show()
```



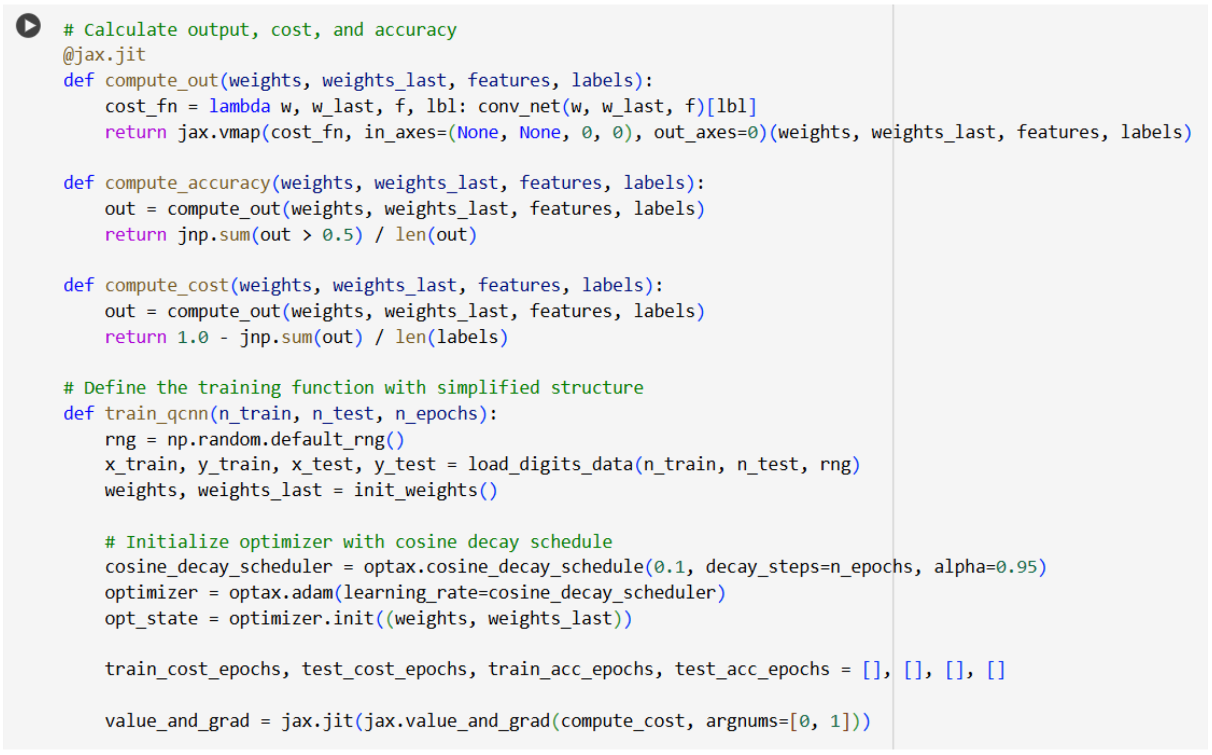
Training the QCNN on the digits dataset

The Quantum Convolutional Neural Network (QCNN) is trained to classify handwritten digits, specifically digits 0 and 1, from the classical digits dataset. The dataset is preprocessed by normalising the images and selecting only the relevant classes. The QCNN model, initialised with random weights, is optimised using JAX and PennyLane with the Adam optimizer. Training results are evaluated by tracking the cost and accuracy, with the model's performance plotted across various training set sizes to observe generalisation behaviour and compare training vs. test accuracies.

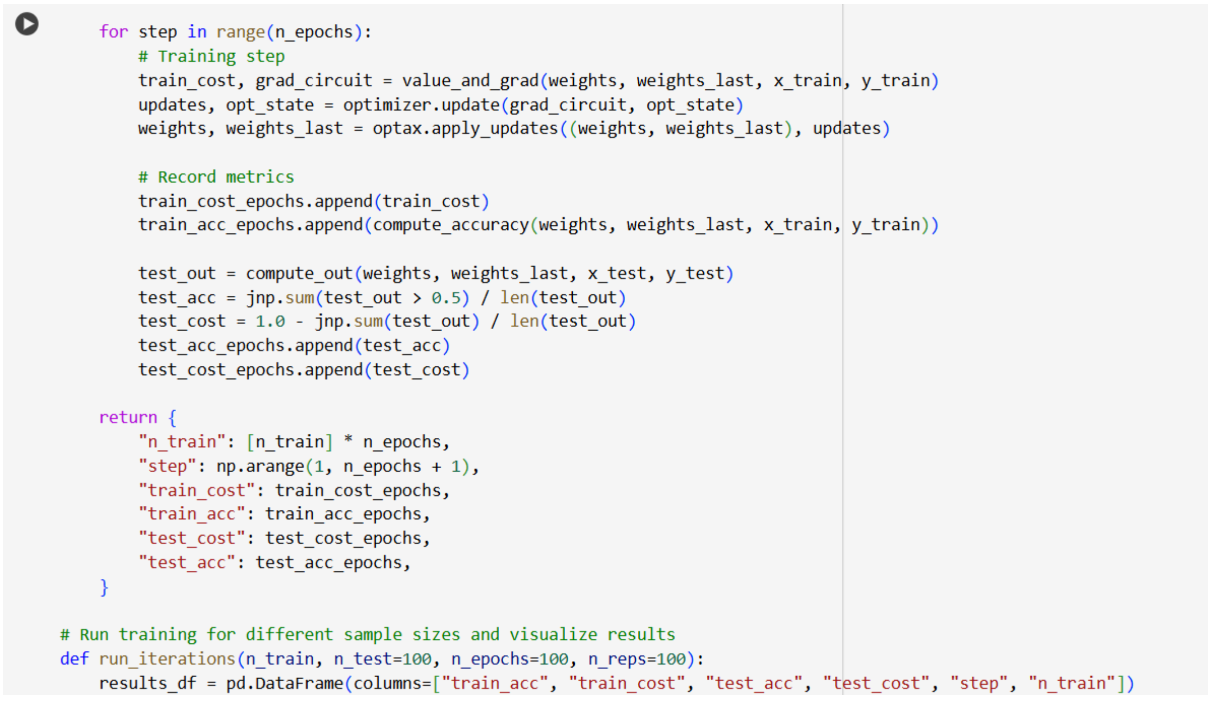
Dataset :



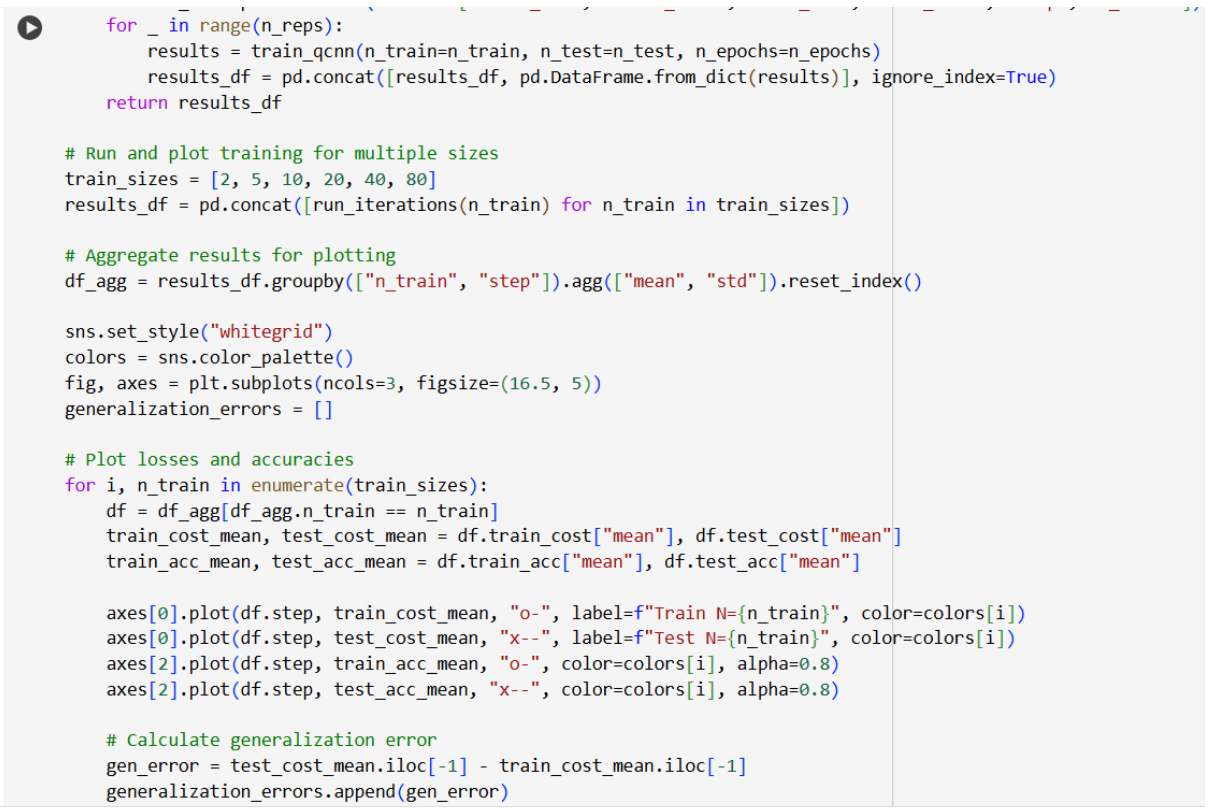
```python
import pennylane as qml
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
import jax
import jax.numpy as jnp
import optax
import seaborn as sns
import pandas as pd

# Load and preprocess the digits dataset
def load_digits_data(num_train, num_test, rng):
    digits = datasets.load_digits()
    features, labels = digits.data, digits.target
    features, labels = features[(labels == 0) | (labels == 1)], labels[(labels == 0) | (labels == 1)]
    features /= np.linalg.norm(features, axis=1, keepdims=True)  # normalize

    train_indices = rng.choice(len(labels), num_train, replace=False)
    test_indices = rng.choice(np.setdiff1d(range(len(labels)), train_indices), num_test, replace=False)

    x_train, y_train = features[train_indices], labels[train_indices]
    x_test, y_test = features[test_indices], labels[test_indices]
    return jnp.asarray(x_train), jnp.asarray(y_train), jnp.asarray(x_test), jnp.asarray(y_test)

# Initialize random weights for QCNN
def init_weights():
    weights = np.random.normal(0, 1, (18, 2))
    weights_last = np.random.normal(0, 1, 4 ** 2 - 1)
    return jnp.array(weights), jnp.array(weights_last)
```



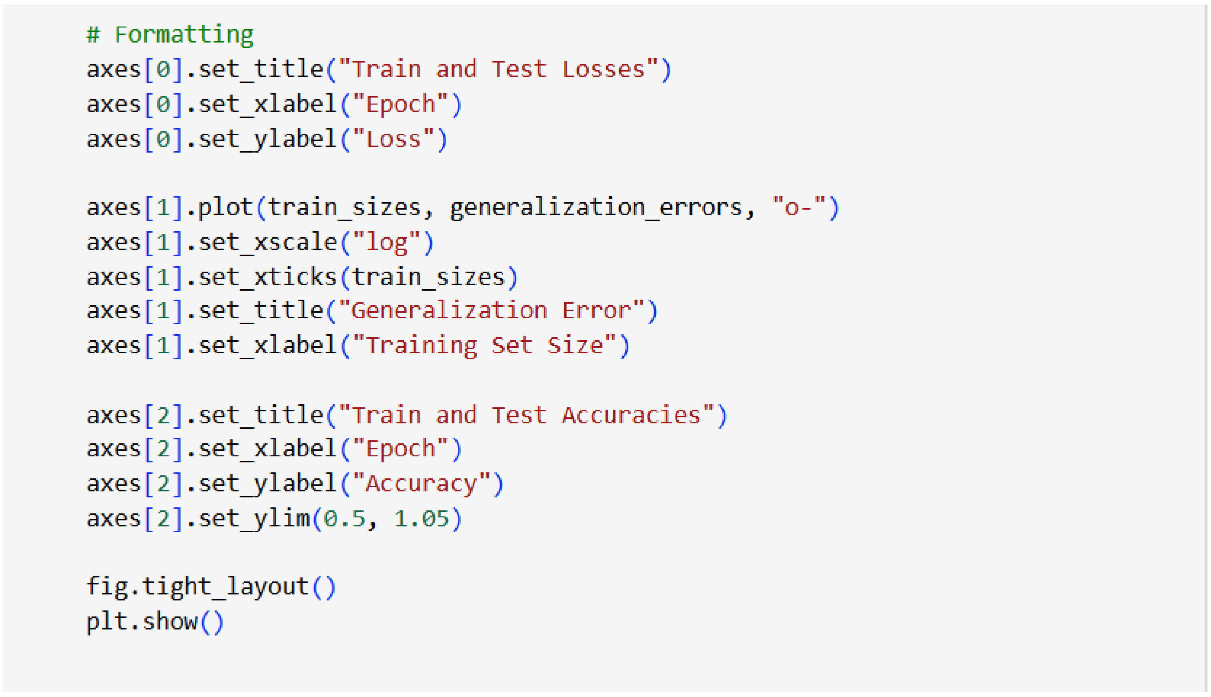
```python
# Calculate output, cost, and accuracy
@jax.jit
def compute_out(weights, weights_last, features, labels):
    cost_fn = lambda w, w_last, f, lbl: conv_net(w, w_last, f)[lbl]
    return jax.vmap(cost_fn, in_axes=(None, None, 0, 0), out_axes=0)(weights, weights_last, features, labels)

def compute_accuracy(weights, weights_last, features, labels):
    out = compute_out(weights, weights_last, features, labels)
    return jnp.sum(out > 0.5) / len(out)

def compute_cost(weights, weights_last, features, labels):
    out = compute_out(weights, weights_last, features, labels)
    return 1.0 - jnp.sum(out) / len(labels)

# Define the training function with simplified structure
def train_qcnn(n_train, n_test, n_epochs):
    rng = np.random.default_rng()
    x_train, y_train, x_test, y_test = load_digits_data(n_train, n_test, rng)
    weights, weights_last = init_weights()

    # Initialize optimizer with cosine decay schedule
    cosine_decay_scheduler = optax.cosine_decay_schedule(0.1, decay_steps=n_epochs, alpha=0.95)
    optimizer = optax.adam(learning_rate=cosine_decay_scheduler)
    opt_state = optimizer.init((weights, weights_last))

    train_cost_epochs, test_cost_epochs, train_acc_epochs, test_acc_epochs = [], [], [], []

    value_and_grad = jax.jit(jax.value_and_grad(compute_cost, argnums=[0, 1]))
```



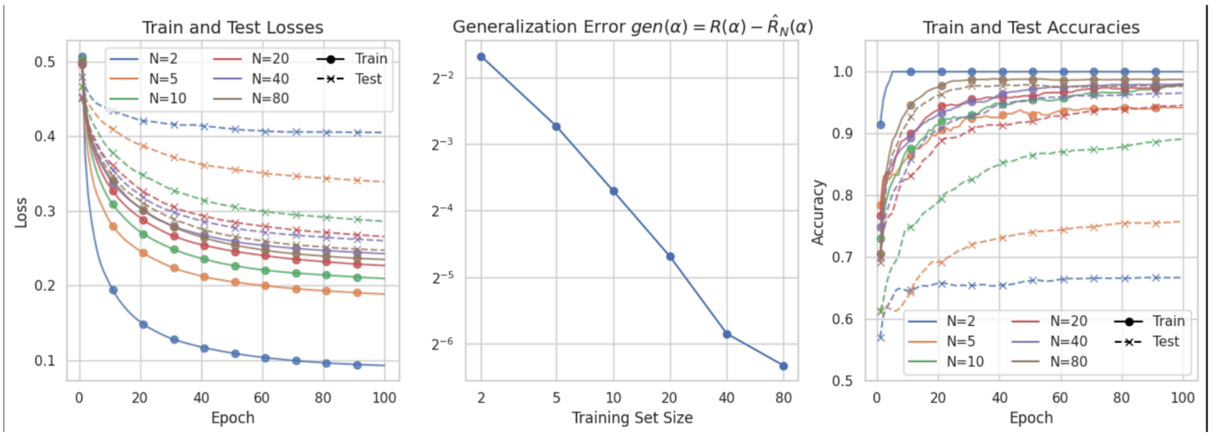
```python
    for step in range(n_epochs):
        # Training step
        train_cost, grad_circuit = value_and_grad(weights, weights_last, x_train, y_train)
        updates, opt_state = optimizer.update(grad_circuit, opt_state)
        weights, weights_last = optax.apply_updates((weights, weights_last), updates)

        # Record metrics
        train_cost_epochs.append(train_cost)
        train_acc_epochs.append(compute_accuracy(weights, weights_last, x_train, y_train))

        test_out = compute_out(weights, weights_last, x_test, y_test)
        test_acc = jnp.sum(test_out > 0.5) / len(test_out)
        test_cost = 1.0 - jnp.sum(test_out) / len(test_out)
        test_acc_epochs.append(test_acc)
        test_cost_epochs.append(test_cost)

    return {
        "n_train": [n_train] * n_epochs,
        "step": np.arange(1, n_epochs + 1),
        "train_cost": train_cost_epochs,
        "train_acc": train_acc_epochs,
        "test_cost": test_cost_epochs,
        "test_acc": test_acc_epochs,
    }

# Run training for different sample sizes and visualize results
def run_iterations(n_train, n_test=100, n_epochs=100, n_reps=100):
    results_df = pd.DataFrame(columns=["train_acc", "train_cost", "test_acc", "test_cost", "step", "n_train"])
```



```
    for _ in range(n_reps):
        results = train_qcnn(n_train=n_train, n_test=n_test, n_epochs=n_epochs)
        results_df = pd.concat([results_df, pd.DataFrame.from_dict(results)], ignore_index=True)
    return results_df

# Run and plot training for multiple sizes
train_sizes = [2, 5, 10, 20, 40, 80]
results_df = pd.concat([run_iterations(n_train) for n_train in train_sizes])

# Aggregate results for plotting
df_agg = results_df.groupby(["n_train", "step"]).agg(["mean", "std"]).reset_index()

sns.set_style("whitegrid")
colors = sns.color_palette()
fig, axes = plt.subplots(ncols=3, figsize=(16.5, 5))
generalization_errors = []

# Plot losses and accuracies
for i, n_train in enumerate(train_sizes):
    df = df_agg[df_agg.n_train == n_train]
    train_cost_mean, test_cost_mean = df.train_cost["mean"], df.test_cost["mean"]
    train_acc_mean, test_acc_mean = df.train_acc["mean"], df.test_acc["mean"]

    axes[0].plot(df.step, train_cost_mean, "o-", label=f"Train N={n_train}", color=colors[i])
    axes[0].plot(df.step, test_cost_mean, "x--", label=f"Test N={n_train}", color=colors[i])
    axes[2].plot(df.step, train_acc_mean, "o-", color=colors[i], alpha=0.8)
    axes[2].plot(df.step, test_acc_mean, "x--", color=colors[i], alpha=0.8)

    # Calculate generalization error
    gen_error = test_cost_mean.iloc[-1] - train_cost_mean.iloc[-1]
    generalization_errors.append(gen_error)
```



```
# Formatting
axes[0].set_title("Train and Test Losses")
axes[0].set_xlabel("Epoch")
axes[0].set_ylabel("Loss")

axes[1].plot(train_sizes, generalization_errors, "o-")
axes[1].set_xscale("log")
axes[1].set_xticks(train_sizes)
axes[1].set_title("Generalization Error")
axes[1].set_xlabel("Training Set Size")

axes[2].set_title("Train and Test Accuracies")
axes[2].set_xlabel("Epoch")
axes[2].set_ylabel("Accuracy")
axes[2].set_ylim(0.5, 1.05)

fig.tight_layout()
plt.show()
```



Results and Analysis:

The QCNN achieves high accuracy, especially with larger training sets.Using six qubits, we have trained the QCNN to distinguish between handwritten digits of 0s and 1s. With 80 samples, we have achieved a model with accuracy greater than 97% in 100 training epochs. This demonstrates the potential of quantum machine learning for classification tasks. The generalization error, the difference between training and testing accuracy, is analysed to understand the model's ability to generalize to unseen data. Furthermore, we have compared the test and train accuracy of this model for a different number of training samples and found the scaling of the generalization error agrees with the theoretical bounds.

Case Study: QCNN vs Classical CNN

Classification is particularly relevant to Information Retrieval, as it is used in various subtasks of the search pipeline. In this work, we propose a quantum convolutional neural network (QCNN) for multi-class classification of classical data. The model is implemented using PennyLane. The optimization process is conducted by minimizing the cross-entropy loss through parameterized quantum circuit optimization. The QCNN is tested on the MNIST dataset with 4, 6, 8 and 10 classes. The results show that with 4 classes, the performance is slightly lower compared to the classical CNN, while with a higher number of classes, the QCNN out-performs the classical neural network

In recent years, there has been a significant advancement in quantum machine learning, particularly with the development and implementation of quantum convolutional neural networks (QCNNs). This growing interest in leveraging quantum computing for machine learning tasks stems from the potential of quantum systems to outperform classical systems in various applications, thanks to their unique properties such as superposition and entanglement.

The QCNN is designed to tackle multi-class classification problems, which are prevalent in machine learning. While classical neural networks have demonstrated considerable success in these tasks, they often require a large number of parameters and extensive training data to achieve high accuracy. In contrast, our QCNN aims to achieve competitive performance with a significantly lower parameter count, presenting a promising alternative.

In this study, we focus on specific scenarios involving classification tasks with 4, 6, 8, and 10 classes. The architecture of the QCNN incorporates convolutional layers and pooling layers, which are crucial for capturing spatial hierarchies in data. However, the current design leaves some qubit states unused during the measurement process, potentially introducing noise and impacting classification accuracy.

Looking ahead, we recognize the need to optimize the QCNN architecture, particularly by modifying the pooling layers and enhancing the measurement strategies. Our research aims to contribute to a deeper understanding of the capabilities of quantum neural networks and their potential advantages over classical counterparts in machine learning applications. Through these efforts, we aspire to unlock new possibilities in the field of quantum machine learning.

Code Implementation:

1.We implement a QCNN trained to classify the digits of the MNIST dataset:

```
import tensorflow as tf
import pickle
import inspect

from sklearn.metrics import multilabel_confusion_matrix
from sklearn.decomposition import PCA

import pennylane as qml
from pennylane.templates.embeddings import AmplitudeEmbedding,
AngleEmbedding
from pennylane import numpy as np
import autograd.numpy as anp

n_qubit = 8 #number of qubit in the circuit
encoding = 'amplitude' #choose the quantum encoding: 'amplitude' or
'angle'
num_classes = 10 # choose how many classes: 4, 6, 8, 10
all_samples = True #True if you want all the samples, False, if you
want only 250 samples for each class
seed = 43 #set to None to generate the seed randomly
U_params = 15 #number of parameters of F_2 circuit
num_layer = 1 #number of convolutional layer repetitions
load_params = False #if True load parameters from a file
opt = 'Adam' #choose the optimizer: Adam, QNGO, or GDO
lr = 0.01 #learning rate
epochs = 2 #number of epochs
batch_size = 64 #size of batch
```

2. Load the MNIST dataset: split it in train and test, take only 250 samples if all_samples == False.

Take only 256 features if the amplitude encoding is applied, otherwise only 8 if the angle encoding is used

```
"""
It loads the MNIST dataset and then it processes the dataset based on
the encoding method, number of classes and if we want all the samples.
param encoding: indicate the quantum encoding used: 'amplitude' or
'angle'
```

```
param num_classes: number of classes to be predicted, which samples
take from the dataset
param all_samples: True if we want all the samples, False to take only
250 samples for each class
param seed: random_state seed
return X_train, X_test, Y_train, Y_test: the dataset divided in
training and test set
"""
def data_load_and_process(encoding, num_classes, all_samples, seed):
  if seed != None:
    tf.random.set_seed(seed)
    np.random.seed(seed)

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


  x_train, x_test = x_train[..., np.newaxis] / 255.0, x_test[...,
np.newaxis] / 255.0  # normalize the data

  #check is the user want all the samples
  if not all_samples:
    num_examples_per_class = 250
    selected_indices = []

    # Iterate through each class to select 1000 examples
    for class_label in range(10):
      indices = np.where(y_train ==
class_label)[0][:num_examples_per_class]
      selected_indices.extend(indices)

    # Filter the training data to contain only the selected examples
    x_train_subset = x_train[selected_indices]
    y_train_subset = y_train[selected_indices]

    # Shuffle the data
    shuffle_indices = np.random.permutation(len(x_train_subset))
    x_train = x_train_subset[shuffle_indices]
    y_train = y_train_subset[shuffle_indices]

  print("Shape of subset training data:", x_train.shape)
  print("Shape of subset training labels:", y_train.shape)
  #take only the number of classes selected
  mask_train = np.isin(y_train, range(0, num_classes))
  mask_test = np.isin(y_test, range(0, num_classes))

  X_train = x_train[mask_train]
  X_test = x_test[mask_test]
  Y_train = y_train[mask_train]
  Y_test = y_test[mask_test]

  print("Shape of subset training data:", X_train.shape)
  print("Shape of subset training labels:", Y_train.shape)

  #check which encoding is used
  #if amplitude encoding is used, then the 256 most important features
are taken using PCA
  if encoding == 'amplitude':
    X_train_flat = X_train.reshape(X_train.shape[0], -1)
    X_test_flat = X_test.reshape(X_test.shape[0], -1)
    pca = PCA(n_components = 256)
    X_train = pca.fit_transform(X_train_flat)
    X_test = pca.transform(X_test_flat)
    return X_train, X_test, Y_train, Y_test
  #if amplitude encoding is used, then the 8 most important features
are taken using PCA
  elif encoding == 'angle':
    X_train = tf.image.resize(X_train[:], (784, 1)).numpy()
    X_test = tf.image.resize(X_test[:], (784, 1)).numpy()
    X_train, X_test = tf.squeeze(X_train), tf.squeeze(X_test)

    pca = PCA(8)

    X_train = pca.fit_transform(X_train)
    X_test = pca.transform(X_test)

    # Rescale for angle embedding

    X_train, X_test = (X_train - X_train.min()) * (np.pi /
(X_train.max() - X_train.min())),\
              (X_test - X_test.min()) * (np.pi / (X_test.max() -
X_test.min()))
    return X_train, X_test, Y_train, Y_test
```

```
X_train, X_test, Y_train, Y_test = data_load_and_process(encoding,
num_classes, all_samples, seed)
```

3. Define the QCNN:

1) Create the two circuits which implement the convolutional layer and the circuit which implements the pooling operation

```
"""
F_1 circuit of the paper
param params: theta angle of the rotations. parameters to be trained
param wires: qubits to apply the gates
"""
def CC14(params, wires):
  #U_CC14 r = 1
  for i in range(0, len(wires)):
    qml.RY(params[i], wires=wires[i])
  for i in range(0, len(wires)):
    qml.CRX(params[i + len(wires)], wires=[wires[(i - 1) % len(wires)],
wires[i]])



  #U_CC14 r = -1 or 3
  for i in range(0, len(wires)):
    qml.RY(params[i + 2 * len(wires)], wires=wires[i])

  if len(wires) % 3 == 0 or len(wires) == 2:
    for i in range(len(wires) - 1, -1, -1):
      qml.CRX(params[i + 3 * len(wires)], wires=[wires[i], wires[(i-1)
% len(wires)]])

  else:
    control = len(wires) - 1
    target = (control + 3) % len(wires)
    for i in range(len(wires) - 1, -1, -1):
      qml.CRX(params[i + 3 * len(wires)], wires=[wires[control],
wires[target]])

      control = target
      target = (control + 3) % len(wires)

"""
F_2 circuit of the paper
```

```
param params: theta angle of the rotations. parameters to be trained
param wires: qubits to apply the gates
"""
def U_SU4(params, wires): # 15 params
  qml.U3(params[0], params[1], params[2], wires=wires[0])
  qml.U3(params[3], params[4], params[5], wires=wires[1])
  qml.CNOT(wires=[wires[0], wires[1]])
  qml.RY(params[6], wires=wires[0])
  qml.RZ(params[7], wires=wires[1])
  qml.CNOT(wires=[wires[1], wires[0]])
  qml.RY(params[8], wires=wires[0])
  qml.CNOT(wires=[wires[0], wires[1]])
  qml.U3(params[9], params[10], params[11], wires=wires[0])
  qml.U3(params[12], params[13], params[14], wires=wires[1])

"""
It implements the pooling circuit
param params: theta angle of the rotations. parameters to be trained
param wires: qubits to apply the gates
"""
def Pooling_ansatz(params, wires): #2 params
  qml.CRZ(params[0], wires=[wires[0], wires[1]])
  qml.PauliX(wires=wires[0])
  qml.CRX(params[1], wires=[wires[0], wires[1]])
```

2) create the structure of the convolutional layers

```
"""
Quantum Circuits for Convolutional layers
param U: unitary that implements the convolution
param params: theta angle of the rotations. parameters to be trained
param U_params: number of parameters which implement a single block of
the F_2 circuit
param num_layer: number of repetition of the convolutional layer
param qubits: array that indicate to which qubit apply the
convolutional layer
"""
def conv_layer(U, params, U_params, num_layer, qubits):
    param0 = 0
    param1 = len(qubits) * 2

    #add f_1 circuit
    for l in range(num_layer):
      if len(qubits) == 8: #if it is the first layer, the F_1 circuit
is "divided"
        for i in range(0, len(qubits), len(qubits)//2):
          U(params[param0: param1], wires = qubits[i: i +
len(qubits)//2])
      else:
        param1 += len(qubits) * 2
        U(params[param0: param1], wires = qubits[0: len(qubits)])

      #now add the two-qubit circuit (F_2)
      param0 = param1
      param1 += U_params
      for i in range(0, len(qubits), 2):
        U_SU4(params[param0: param1], wires = [qubits[i % len(qubits)],
qubits[(i + 1) % len(qubits)]])

      for i in range(1, len(qubits), 2):
        U_SU4(params[param0: param1], wires = [qubits[i % len(qubits)],
qubits[(i + 1) % len(qubits)]])

      param0 = param1
      param1 += len(qubits) * 2
```

3) create the structure of the pooling layers

```
"""
Quantum Circuits for Pooling layers
param V: unitary which implements the pooling operation
param params: theta angle of the rotations. parameters to be trained
"""
def pooling_layer1(V, params):
  V(params, wires=[7, 6])
  V(params, wires=[1, 0])

def pooling_layer2(V, params):
  V(params, wires=[3, 2])
  V(params, wires=[5, 4])

def pooling_layer3(V, params, num_classes):
  if num_classes == 4: #if we need only 4 classes. we trace out another
qubit
    V(params, wires=[2,0])
  V(params, wires=[6,4])
```

4) create the structure of the QCNN

```
"""
It implements the structure of the QCNN
param U: unitary F_1
param params: theta angle of the rotations. parameters to be trained
param U_params: number of parameters which implement a single block of
the F_2 circuit
param num_classes: how many classes the QNN has to predict
param num_layer: number of repetition of the convolutional layer
"""
def QCNN_structure(U, params, U_params, num_classes, num_layer):
  #divide the number of parameters for each layer: conv layer1, pooling
layer 1, conv layer 2, ...
  #U_params indicates the number of parameters of the F_2 circuit (the
circuit applied to couple of adjacent qubit)
  #n_qubit * 2: is the number of parameters for the circuit F_1
  param1CL = params[0: (U_params + n_qubit * 2) * num_layer]
  param1PL = params[(U_params + n_qubit * 2) * num_layer: ((U_params +
n_qubit * 2) * num_layer) + 2]

  param2CL = params[((U_params + n_qubit * 2) * num_layer) + 2:
((U_params + n_qubit * 2) * num_layer) + 2 + ((U_params + (n_qubit - 2)
* 4) * num_layer)]
  param2PL = params[((U_params + n_qubit * 2) * num_layer) + 2 +
((U_params + (n_qubit - 2) * 4) * num_layer):
            ((U_params + n_qubit * 2) * num_layer) + 2 + ((U_params +
(n_qubit - 2) * 4) * num_layer) + 2]

  param3CL = params[((U_params + n_qubit * 2) * num_layer) + 2 +
((U_params + (n_qubit - 2) * 4) * num_layer) + 2:
            ((U_params + n_qubit * 2) * num_layer) + 2 + ((U_params +
(n_qubit - 2) * 4) * num_layer) + 2 + ((U_params + n_qubit * 2) *
num_layer)]

  #apply the circuits
  conv_layer(U, param1CL, U_params, num_layer, range(n_qubit))
  pooling_layer1(Pooling_ansatz, param1PL)

  conv_layer(U, param2CL, U_params, num_layer, [0, 2, 3, 4, 5, 6])
  pooling_layer2(Pooling_ansatz, param2PL)

  conv_layer(U, param3CL, U_params, num_layer, [0, 2, 4, 6])
  #if we have only 4, 6 or 8 classes, then we need another pooling
layer and we need to trace out:
  #another qubit if we have 6 or 8 classes, because we need only 3
qubits to represent 6 or 8 classes
  #2 qubits if we have 4 classes, because we need only 2 qubits to
represent 4 classes/states
  #if we have 10 classes, then we don't apply another pooling layer,
because we need 4 qubits
  if num_classes == 4 or num_classes == 6 or num_classes == 8:
    param3PL = params[((U_params + n_qubit * 2) * num_layer) + 2 +
((U_params + (n_qubit - 2) * 4) * num_layer) + 2 + ((U_params + n_qubit
* 2) * num_layer):
              ((U_params + n_qubit * 2) * num_layer) + 2 + ((U_params +
(n_qubit - 2) * 4) * num_layer) + 2 + ((U_params + n_qubit * 2) *
num_layer) + 2]

    pooling_layer3(Pooling_ansatz, param3PL, num_classes)
```

5)Create the QCNN

```
"""
define the simulator and the QCNN: encoding + VQC + measurement
param X: sample in input
param params: theta angle of the rotations. parameters to be trained
param U_params: number of parameters which implement a single block of
the F_2 circuit
param embedding_type: which encoding is chosen
param num_classes: how many classes the QNN has to predict
param num_layer: number of repetition of the convolutional layer
return result: the probabilities of the states, which are associated to
the MNIST classes
"""
dev = qml.device('default.qubit', wires = n_qubit)
@qml.qnode(dev)
def QCNN(X, params, U_params, embedding_type='amplitude',
num_classes=10, num_layer = 1):
  # Data Embedding
  if embedding_type == 'amplitude':
    AmplitudeEmbedding(X, wires=range(8), normalize=True)
  elif embedding_type == 'angle':
    AngleEmbedding(X, wires=range(8), rotation='Y')
```

```
# Create the VQC
QCNN_structure(CC14, params, U_params, num_classes, num_layer)

#Measures the necessary qubits
if num_classes == 4:
  result = qml.probs(wires=[0, 4])
elif num_classes == 6:
  result = qml.probs(wires=[0, 2, 4])
elif num_classes == 8:
  result = qml.probs(wires=[0, 2, 4])
else:
  result = qml.probs(wires=[0, 2, 4, 6])

return result
```

Training Step:

1) define the loss function

```
"""
It computes the cross-entropy loss
param labels: correct classes of the Training set
param predictions: classes predicted by the QCNN
param num_classes: number of classes
return loss: average loss
"""
def cross_entropy(labels, predictions, num_classes):
  epsilon = 1e-15
  num_samples = len(labels)

  num_classes = len(predictions[0])
  Y_true_one_hot = anp.eye(num_classes)[labels]

  loss = 0.0
  for i in range(num_samples):
    predictions[i] = anp.clip(predictions[i], epsilon, 1 - epsilon)
    loss -= anp.sum(Y_true_one_hot[i] * anp.log(predictions[i]))


  return loss / num_samples
```

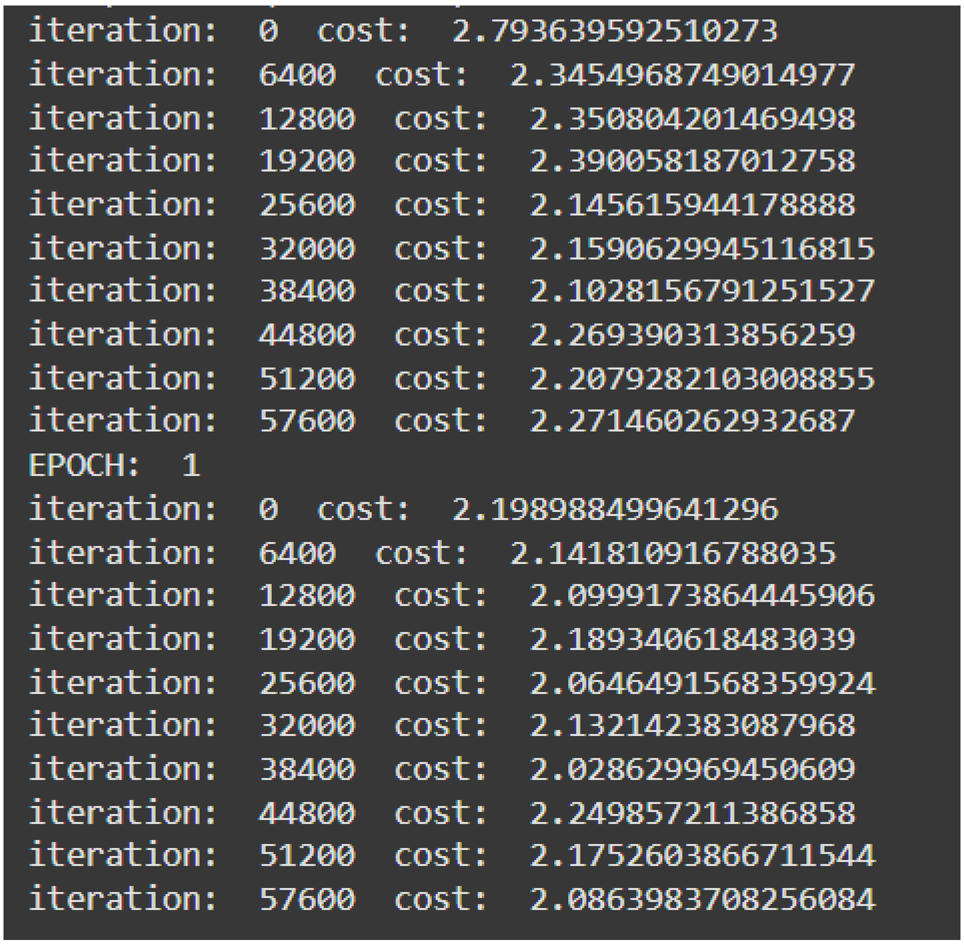
```
"""
It executes the circuit for each image of the dataset (divided in
batches)
param calculate the loss function
param params: the angle to be trained
param X: batches of the training set
param Y: batches of the training set
param U_params: number of parameters which implement a single block of
the F_2 circuit
param embedding_type: indicate the chosen encoding )
param circ_layer: number of repetitions of the convolutional layer
return loss: average loss
"""
def cost(params, X, Y, U_params, embedding_type, num_classes,
circ_layer):
  predictions = [QCNN(x, params, U_params, embedding_type,
num_classes=num_classes, layer = circ_layer) for x in X]


  loss = cross_entropy(Y, predictions, num_classes)

  return loss
```

2) Execute the training:

```
"""
It executes the training of the QNN
param X_train: X training set
param Y_train: Y training set
param U_params: number of parameters which implement a single block of
the F_2 circuit
param embedding_type: the encoding method used
param num_classes: number of classes
param num_layer: number of repetitions of conv layer
param loadParams: if True the parameters are loaded from a file (used
to continue a stopped training)
param optimizer: the optimizer used
param learning_rate: learning rate of the optimizer
param epochs: number of epochs
param all_samples: it all the samples are used
param batch_size: size of the batches
param seed: if None a random seed is used, otherwise the value in the
variable
return params: the trained parameters
"""
def circuit_training(X_train, Y_train, U_params, embedding_type,
num_classes, num_layer, loadParams, optimizer, learning_rate, epochs,
all_samples, batch_size, seed):
  if seed != None:
    np.random.seed(seed)
    anp.random.seed(seed)

  #calculate the number of parameters
  if num_classes == 10:
    total_params =  ((U_params + n_qubit * 2) * num_layer) + 2 +
((U_params + (n_qubit - 2) * 4) * num_layer) + 2 + ((U_params + n_qubit
* 2) * num_layer)
  else: #we have to add another pooling layer at the end, so we need
two parameters
    total_params =  ((U_params + n_qubit * 2) * num_layer) + 2 +
((U_params + (n_qubit - 2) * 4) * num_layer) + 2 + ((U_params + n_qubit
* 2) * num_layer) + 2

  #laod the parameters
  if not loadParams:
    params = np.random.randn(total_params, requires_grad=True)
  else:
    fileParams = open('params' + 'L' + str(num_layer) + 'LR' +
str(learning_rate) + optimizer + 'C' + str(num_classes) +
str(all_samples) + '.obj', 'rb')

    params = pickle.load(fileParams)
    fileParams.close()
    print(params)

  #choose the optimizer
  if optimizer == 'Adam':
    opt = qml.AdamOptimizer(stepsize=learning_rate)
  elif optimizer == 'GDO':
    opt = qml.GradientDescentOptimizer(stepsize=learning_rate)
  else:
    opt = qml.QNGOptimizer(stepsize=learning_rate)

  #start the training
  loss_history = []
  grad_vals = []
  for e in range(0, epochs):
    print("EPOCH: ", e)
    for b in range(0, len(X_train), batch_size):
      if (b + batch_size) <= len(X_train):
        X_batch = [X_train[i] for i in range(b, b + batch_size)]
        Y_batch = [Y_train[i] for i in range(b, b + batch_size)]
      else:
        X_batch = [X_train[i] for i in range(b, len(X_train))]
        Y_batch = [Y_train[i] for i in range(b, len(X_train))]

      if optimizer == 'QNGO':
        metric_fn = lambda p: qml.metric_tensor(QCNN,
approx="block-diag")(X_batch, p, U_params, embedding_type, num_classes,
num_layer)
        params, cost_new = opt.step_and_cost(lambda v: cost(v, X_batch,
Y_batch, U_params, embedding_type, num_classes, num_layer),
                              params, metric_tensor_fn=metric_fn)
      else:
        params, cost_new = opt.step_and_cost(lambda v: cost(v, X_batch,
Y_batch, U_params, embedding_type, num_classes, num_layer),
                              params)


      if b % (batch_size * 100) == 0:
        print("iteration: ", b, " cost: ", cost_new)
        """
        loss_history.append(cost_new)
        gradient_fn = qml.grad(cost)
        gradients = gradient_fn(params, X_batch, Y_batch, U, U_params,
embedding_type, cost_fn, num_classes, num_layer)
        grad_vals.append(gradients[-1])
        print(gradients)
        print("var ", np.var(grad_vals))
        print("mean grad: ", np.mean(grad_vals))
        """


    #save the novel parameters at the end of each epoch
    fileParams = open('params' + 'L' + str(num_layer) + 'LR' +
str(learning_rate) + optimizer + 'C' + str(num_classes) +
str(all_samples) + '.obj', 'wb')
```

```
    pickle.dump(params, fileParams)
    fileParams.close()
  return params

print("Loss History for circuit with " + encoding)
trained_params = circuit_training(X_train, Y_train, U_params, encoding,
num_classes, num_layer, load_params,
                opt, lr, epochs, all_samples, batch_size, seed)
```

Output:



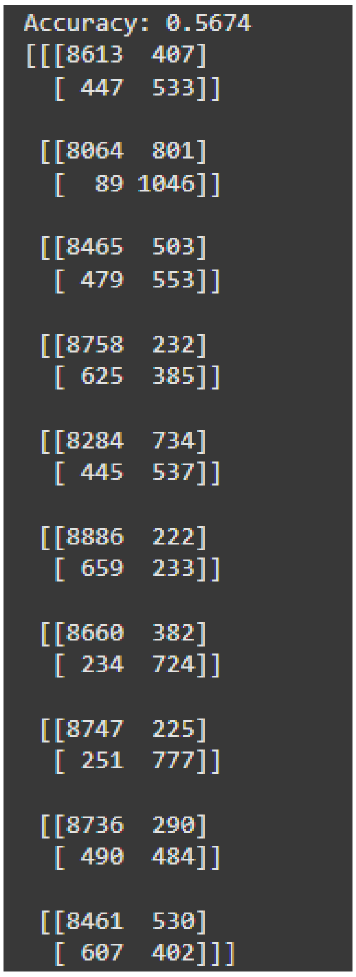
```
iteration:  0  cost:  2.793639592510273
iteration:  6400  cost:  2.3454968749014977
iteration:  12800  cost:  2.350804201469498
iteration:  19200  cost:  2.390058187012758
iteration:  25600  cost:  2.145615944178888
iteration:  32000  cost:  2.1590629945116815
iteration:  38400  cost:  2.1028156791251527
iteration:  44800  cost:  2.269390313856259
iteration:  51200  cost:  2.2079282103008855
iteration:  57600  cost:  2.271460262932687
EPOCH:  1
iteration:  0  cost:  2.198988499641296
iteration:  6400  cost:  2.141810916788035
iteration:  12800  cost:  2.0991738644445906
iteration:  19200  cost:  2.189340618483039
iteration:  25600  cost:  2.0646491568359924
iteration:  32000  cost:  2.132142383087968
iteration:  38400  cost:  2.028629969450609
iteration:  44800  cost:  2.249857211386858
iteration:  51200  cost:  2.1752603866711544
iteration:  57600  cost:  2.0863983708256084
```

Infere on the test set

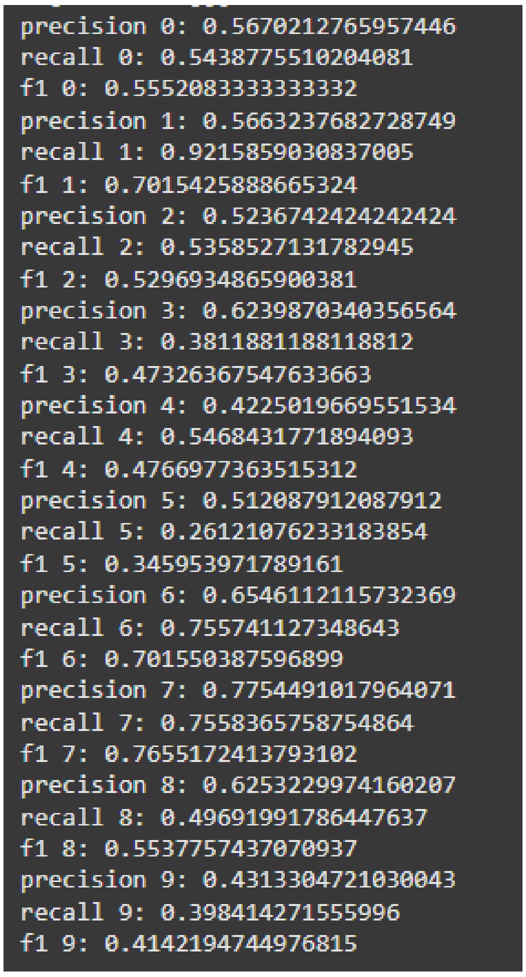
```
"""
It computes the accuracy on the test set
param predictions: classes predicted
param labels: true classes
param num_classes: number of classes
return accuracy: accuracy
"""
def accuracy_multi(predictions, labels, num_classes):
  correct_predictions = 0


  for l, p in zip(labels, predictions):
    p2 = []
    for i in range(0, num_classes):
      p2.append(p[i])
    predicted_class = np.argmax(p2) # Find the index of the predicted
class with highest probability
    if predicted_class == l:
      correct_predictions += 1

  accuracy = correct_predictions / len(labels)
  return accuracy

"""
It computes the precision, recall, F1 score and Confusion Matrix on the
test set
param predictions: classes predicted
param labels: true classes
param num_classes: number of classes
return accuracy: accuracy
"""
def accuracy_test_multiclass(predictions, label, num_classes):
  #confusion matrix

  preds_np = np.array(predictions)
  preds = np.argmax(preds_np[:, :num_classes], axis = 1)

  conf_mat = multilabel_confusion_matrix(label, preds, labels =
list(range(num_classes)))
  print(conf_mat)
  precision = []
  recall = []
  f1 = []
  i = 0
  for c in conf_mat:
    precision.append(c[1][1] / (c[1][1] + c[0][1]))
    recall.append(c[1][1] / (c[1][1] + c[1][0]))
    f1.append(2 * (precision[i] * recall[i]) / (precision[i] +
recall[i] + np.finfo(float).eps))

    print("precision " + str(i) + ": " + str(precision[i]))
    print("recall " + str(i) + ": " + str(recall[i]))
    print("f1 " + str(i) + ": " + str(f1[i]))
    i += 1

predictions = []

for x in X_test:
  predictions.append(QCNN(x, trained_params, U_params, encoding,
num_classes, num_layer))

accuracy = accuracy_multi(predictions, Y_test, num_classes)
print("Accuracy: " + str(accuracy))
accuracy_test_multiclass(predictions, Y_test, num_classes)
```

Output:

This way, it is continued for 4,6,8 classes as well



```
Accuracy: 0.5674
[[[8613  407]
  [ 447  533]]

 [[8064  801]
  [  89 1046]]

 [[8465  503]
  [ 479  553]]

 [[8758  232]
  [ 625  385]]

 [[8284  734]
  [ 445  537]]

 [[8886  222]
  [ 659  233]]

 [[8660  382]
  [ 234  724]]

 [[8747  225]
  [ 251  777]]

 [[8736  290]
  [ 490  484]]

 [[8461  530]
  [ 607  402]]]
```



```
precision 0: 0.5670212765957446
recall 0: 0.5438775510204081
f1 0: 0.5552083333333332
precision 1: 0.5663237682728749
recall 1: 0.9215859030837005
f1 1: 0.7015425888665324
precision 2: 0.5236742424242424
recall 2: 0.5358527131782945
f1 2: 0.5296934865900381
precision 3: 0.6239870340356564
recall 3: 0.3811881188118812
f1 3: 0.47326367547633663
precision 4: 0.4225019669551534
recall 4: 0.5468431771894093
f1 4: 0.4766977363515312
precision 5: 0.512087912087912
recall 5: 0.26121076233183854
f1 5: 0.345953971789161
precision 6: 0.6546112115732369
recall 6: 0.755741127348643
f1 6: 0.701550387596899
precision 7: 0.7754491017964071
recall 7: 0.7558365758754864
f1 7: 0.7655172413793102
precision 8: 0.6253229974160207
recall 8: 0.49691991786447637
f1 8: 0.5537757437070937
precision 9: 0.4313304721030043
recall 9: 0.398414271555996
f1 9: 0.4142194744976815
```

Conclusion:

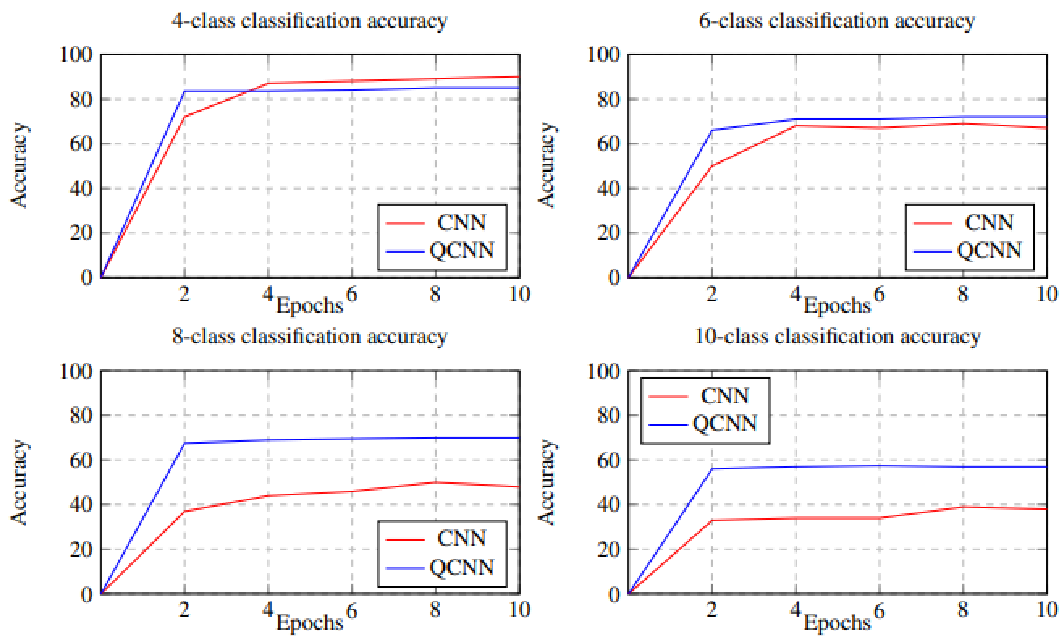


Figure 5: Comparison of the classification accuracy of CNN and QCNN. The evaluation involves increasing the number of epochs and varying the number of classes, with learning rate equal to 0.01. In the 4-class case, classes 0 to 3 are considered, while for 6 classes, classes from 0 to 5 are taken into account, and for 8 classes, classes 0 to 7 are included.

In Fig. 5, a detailed depiction of how the performance of CNN and QCNN varies over epochs is provided. The plots demonstrate that while the CNN achieves better results when it sees the same data multiple times, the QCNN does not show significantly better results. By incrementing the number of epochs for the classical CNN, while a better result is obtained with 4 classes, with 6, 8 and 10 classes the CNN cannot reach the same result of the QCNN.

The QCNN has better performance for 6, 8 and 10 classes. It achieves 72% of accuracy for 6 classes, 70% for 8 classes and 57% for 10 classes, while classical CNN attains 69%, 50% and 38% respectively. Even if the CNN is trained for an additional 90 epochs, it still fails to match the performance of the QCNN.

Challenges and Limitations in Quantum Machine Learning (QML)

1. Noise and Decoherence in Quantum Computers

- Noise:
  Quantum systems are highly sensitive to environmental interactions, leading to errors in quantum states. Noise affects:
  - Gate Operations: Errors during application of quantum gates.
  - Measurement: Inaccurate outcomes during the readout of quantum states.
- Decoherence:
  Quantum states lose coherence (superposition and entanglement) due to interactions with the environment, limiting the time available for computations. This restricts the depth of quantum circuits that can be executed reliably.
- Impact on QML:
  Noise and decoherence introduce inaccuracies in data encoding, model training, and optimization processes, reducing the effectiveness of QML algorithms.

2. Limited Qubit Counts and Gate Fidelity Issues in Current Quantum Hardware

- Limited Qubit Counts:
  Current quantum computers have a small number of qubits (e.g., tens to low hundreds), restricting the size and complexity of problems they can handle. Large-scale datasets and models cannot be fully implemented yet.

- Gate Fidelity:
  Quantum gates are not perfect; their operations introduce errors due to imperfections in hardware and control mechanisms.
  - Cumulative Errors: Errors increase with the number of gates in a circuit.
  - Low Scalability: High error rates hinder the execution of deep quantum circuits needed for QML applications.
- Impact on QML:
  The limited scale of quantum hardware prevents the deployment of robust, high-performance QML models for real-world tasks.

3. High Computational Costs for Simulation on Classical Systems

- Simulating Quantum Systems:
  Classical simulation of quantum systems requires exponential resources as the number of qubits grows. For example, simulating 30+ qubits can demand significant memory and processing power.
- Bottlenecks in Development:
  - Testing QML algorithms on classical hardware is computationally expensive, slowing research and development.
  - Simulations are often limited to small systems, which may not capture the behavior of larger, real-world quantum models.
- Impact on QML:
  The reliance on classical simulations to design and validate QML algorithms delays progress and innovation in the field.

<u>Conclusion</u>

Quantum Machine Learning (QML) stands at the crossroads of quantum computing and classical machine learning, promising transformative impacts on both fields.

1. Impact on Machine Learning:
    - Enhanced Computational Power: Quantum algorithms, such as quantum support vector machines and quantum neural networks, can process large datasets and complex models faster than classical counterparts.
    - Improved Accuracy: Quantum states enable the exploration of vast solution spaces, potentially leading to more precise predictions and pattern recognition.
    - New Paradigms: Concepts like quantum kernels and hybrid algorithms expand the scope of machine learning to problems previously considered intractable.
2. Impact on Quantum Computing:
    - Demonstrating Practical Use Cases: QML offers a clear application that showcases the practical utility of quantum hardware in solving real-world problems.
    - **Accelerated Hardware Development**: The demands of QML drive innovation in quantum processors, error correction, and quantum software ecosystems.

Discussion of Expected Advancements and How They Might Reshape the Landscape

1. Algorithmic Breakthroughs:
    - Development of more robust hybrid quantum-classical algorithms, which combine the best aspects of quantum and classical approaches.
    - Novel quantum-native models tailored to exploit entanglement and superposition for specific tasks.

2. Scalability:
    - Improved quantum hardware, with more qubits and lower error rates, will make QML models scalable and applicable to industries like drug discovery, finance, and optimization.
3. Reshaping Industries:
    - Healthcare: Faster drug discovery by analysing molecular interactions in quantum states.
    - Finance: Advanced risk modelling and portfolio optimisation through faster data processing.
    - Artificial Intelligence: Revolutionizing AI by enabling faster training of deep learning models and exploring novel architectures.
4. Ethical and Societal Considerations:
    - We are addressing challenges in data security and the ethical use of AI, given the immense power of quantum-enhanced systems.
    - They are ensuring that advancements in QML are inclusive and benefit a wide spectrum of society.

Team Members:

1.Jacob Antony Jeejo	-	2023BCY0026
2.Manasa Manoj	-	2023BCY0041
3.Praneetha Reddy	-	2023BCY0047
4.Nahil Rasheed K.E	-	2023BCY0057
5.Adil Omar	-	2023BCY0005
6.Ben Savio	-	2023BCY0009

References:

1. Quantum Programming Software — PennyLane
2. Multi-Class Quantum Convolutional Neural Networks by Marco Mordacci, Davide Ferrari, Michele Amoretti