Day-10 Report

1. Introduction

Implementing a Quantum Neural Network (QNN) for binary image classification using both Cirq and Qiskit. The MNIST dataset was used, restricted to digits 0 and 1. Data was downsampled using PCA and modeled on quantum circuits using two different frameworks.

2. Dataset Preparation and Preprocessing

The MNIST dataset was loaded from TensorFlow. Only digits '0' and '1' were retained to simplify the binary classification task. Each image (28x28 pixels) was flattened and normalized. Then, PCA was applied to reduce the input dimensions to 4 to match the number of qubits available for simulation.

Load MNIST

```
[20] (x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Keep only digits 0 and 1

```
train_filter = np.where((y_train == 0) | (y_train == 1))
test_filter = np.where((y_test == 0) | (y_test == 1))

x_train, y_train = x_train[train_filter], y_train[train_filter]
x_test, y_test = x_test[test_filter], y_test[test_filter]
```

Downsample using PCA to 4 dimensions for 4 qubits

```
[23] x_train = x_train.reshape((-1, 28*28)) / 255.0
     x_test = x_test.reshape((-1, 28*28)) / 255.0
```

```
[24] pca = PCA(n_components=4)
     x_train = pca.fit_transform(x_train)
     x_test = pca.transform(x_test)
```

Build QNN in Cirq

Figure 0

3. QNN using Cirq

The Cirq-based implementation defined a quantum circuit using 4 qubits arranged linearly. Classical input data was encoded using Ry rotations. A variational layer with RZ and RY gates followed by entanglement using CNOT gates was applied. The model was trained using parameter shift gradients and binary cross-entropy loss.

Training was performed on a subset of 100 examples. The model was evaluated on 100 test examples, yielding an accuracy of 26%. Although Cirq was fast in simulation, the accuracy was suboptimal compared to Qiskit.

Parameterized layer

```
def variational_layer(params):
    ops = []
    for i in range(4):
        ops.append(cirq.rz(params[i])(qubits[i]))
        ops.append(cirq.ry(params[i + 4])(qubits[i]))
    # Add entanglement
    for i in range(3):
        ops.append(cirq.CNOT(qubits[i], qubits[i+1]))
    return ops
```

Full circuit

```
[29] def create_circuit(x, params):
    circuit = cirq.Circuit()
    circuit.append(encode_data(x))
    circuit.append(variational_layer(params))
    return circuit
```

Figure 1

Build QNN in Cirq

```
[25] import cirq
```

Define 4 qubits

```
[26] qubits = [cirq.GridQubit(0, i) for i in range(4)]
```

Encode classical data with Ry rotations

```
def encode_data(x):
    return [cirq.ry(x[i])(qubits[i]) for i in range(4)]
```

Figure 2

Define Expectation Measurement

```
[30] simulator = cirq.Simulator()
```

```
[31] def predict(x, params):
         circuit = create_circuit(x, params)
         circuit.append(cirq.measure(qubits[0], key='m'))
         result = simulator.run(circuit, repetitions=100)
         counts = result.histogram(key='m')
         prob_0 = counts.get(0, 0) / 100
         return prob_0  # closer to 1 = class 0
```

Train with Manual Parameter Shift

```
[32] def binary_cross_entropy(y_true, y_pred):
         epsilon = 1e-10
         return - (y_true * np.log(y_pred + epsilon) + (1 - y_true) * np.log(1 - y_pred + ep
```

Figure 3

```
[33] def parameter_shift_grad(x, y, params, shift=np.pi/2):
         grads = np.zeros_like(params)
         for i in range(len(params)):
             plus = params.copy()
             minus = params.copy()
             plus[i] += shift
             minus[i] -= shift
             y_pred_plus = predict(x, plus)
             y_pred_minus = predict(x, minus)
             loss_plus = binary_cross_entropy(y, y_pred_plus)
             loss_minus = binary_cross_entropy(y, y_pred_minus)
             grads[i] = 0.5 * (loss_plus - loss_minus)
         return grads
```

Training Loop (Small Subset)

```
x_train_small = x_train[:100]
y_train_small = y_train[:100]
```

+ Code + Text

```
[35] params = np.random.uniform(0, 2*np.pi, size=8)
     learning_rate = 0.1
```

Figure 4

```
for epoch in range(25):
    total_loss = 0
    for x, y in zip(x_train_small, y_train_small):
        y_pred = predict(x, params)
        loss = binary_cross_entropy(y, y_pred)
        grads = parameter_shift_grad(x, y, params)
        params -= learning_rate * grads
        total_loss += loss
    print(f"Epoch {epoch+1}: Loss = {total_loss/len(x_train_small):.4f}")
```

```
Epoch 1: Loss = 1.1197
Epoch 2: Loss = 0.4978
Epoch 3: Loss = 0.9285
Epoch 4: Loss = 0.8762
Epoch 5: Loss = 0.5205
Epoch 6: Loss = 0.9918
Epoch 7: Loss = 1.1096
Epoch 8: Loss = 0.5099
Epoch 9: Loss = 0.5428
Epoch 10: Loss = 0.9807
Epoch 11: Loss = 0.7319
Epoch 12: Loss = 0.4775
Epoch 13: Loss = 0.9218
Epoch 14: Loss = 1.4663
Epoch 15: Loss = 0.7978
Epoch 16: Loss = 0.8941
Epoch 17: Loss = 0.7142
Epoch 18: Loss = 0.4850
Epoch 19: Loss = 0.5682
Epoch 20: Loss = 0.7886
Epoch 21: Loss = 0.7501
Epoch 22: Loss = 1.0056
Epoch 23: Loss = 0.7504
Epoch 24: Loss = 1.1064
```

Figure 5

Evaluate

```
[39] correct = 0
     for x, y in zip(x_test[:100], y_test[:100]):
         y_pred = predict(x, params)
         predicted_label = 1 if y_pred < 0.5 else 0
         correct += int(predicted_label == y)
     print(f"Accuracy: {correct} / 100 = {correct}%")
```

```
Accuracy: 26 / 100 = 26%
```

Figure 6

4. QNN using Qiskit

In Qiskit, a similar architecture was built using 4 qubits. Input was encoded using Ry rotations, and parameterized layers used RZ and RY gates. Entanglement was achieved using CNOT gates. Qiskit Aer's Sampler primitive was used for inference. Parameter shift rule was applied to compute gradients and perform manual training.

Training on 100 samples with 5 epochs led to an accuracy of 75% on the test set. Though the performance was significantly better than Cirq, training time was considerably longer due to overheads in Qiskit simulation.

Step 2: Build variational quantum circuit with data and parameters

```
def build_circuit(data_point, weights):
    qc = QuantumCircuit(4)
    # Encode data
    for i in range(4):
        qc.ry(data_point[i], i)
    # Parameterized layer
    for i in range(4):
        qc.rz(weights[i], i)
        qc.ry(weights[i+4], i)
    for i in range(3):
        qc.cx(i, i+1)
    qc.measure_all()
    return qc
```

Figure 7

Binary cross-entropy loss

```
[75] def binary_cross_entropy(y_true, y_pred):
         epsilon = 1e-10
         return - (y_true * np.log(y_pred + epsilon) + (1 - y_true) * np.log(1 - y_pred + ep
```

Parameter shift gradient

```
shift = np.pi / 2
def parameter_shift_grad(x, y, params):
    grads = np.zeros_like(params)
    for i in range(len(params)):
        plus = params.copy()
        minus = params.copy()
        plus[i] += shift
        minus[i] -= shift
        loss_plus = binary_cross_entropy(y, predict(x, plus))
        loss_minus = binary_cross_entropy(y, predict(x, minus))
        grads[i] = 0.5 * (loss_plus - loss_minus)
    return grads
```

Figure 8

Step 3: Inference using Qiskit's Sampler

```
[73] sampler = Sampler()
```

```
[74] # Function to get prediction probability (Z measurement on qubit 0)
     def predict(x, params):
         circuit = build_circuit(x, params)
         job = sampler.run(circuit).result()
         counts = job.quasi_dists[0]
         # Get qubit-0 marginal (bit 0 in string)
         prob_0 = sum(p for key, p in counts.items() if (format(key, '04b'))[-1] == '0')
         return prob_0
```

Figure 9

Step 4: Train QNN

```python
params = np.random.uniform(0, 2 * np.pi, size=8)
lr = 0.2
for epoch in range(5):
    total_loss = 0
    for x, y in zip(x_train_small, y_train_small):
        y_pred = predict(x, params)
        loss = binary_cross_entropy(y, y_pred)
        grads = parameter_shift_grad(x, y, params)
        params -= lr * grads
        total_loss += loss
    print(f"Epoch {epoch+1}: Loss = {total_loss / len(x_train_small):.4f}")
```

```
Epoch 1: Loss = 0.5295
Epoch 2: Loss = 0.5824
Epoch 3: Loss = 1.1697
Epoch 4: Loss = 0.6951
Epoch 5: Loss = 0.6322
```

Figure 10

Step 5: Evaluate

```python
[78] preds = [1 if predict(x, params) > 0.5 else 0 for x in x_test_small]
     acc = accuracy_score(y_test_small, preds)
     print(f"\nTest Accuracy: {acc * 100:.2f}%")
```

```

Test Accuracy: 75.00%
```

Figure 11

5. Performance Comparison

- Cirq:

    Accuracy: 26%
    Speed: Fast
    Implementation: Manual training with Cirq simulator

- Qiskit:

    Accuracy: 75%
    Speed: Slow
    Implementation: Manual training using Qiskit Aer Sampler

6. Conclusion

Cirq provides faster iteration time but less accuracy in this case, likely due to limited entanglement and gradient expressivity. Qiskit, while slower, offers more accurate simulation and better performance with modern primitives. For deeper quantum ML experiments, Qiskit might be more suitable, provided runtime performance is acceptable.