QCNN

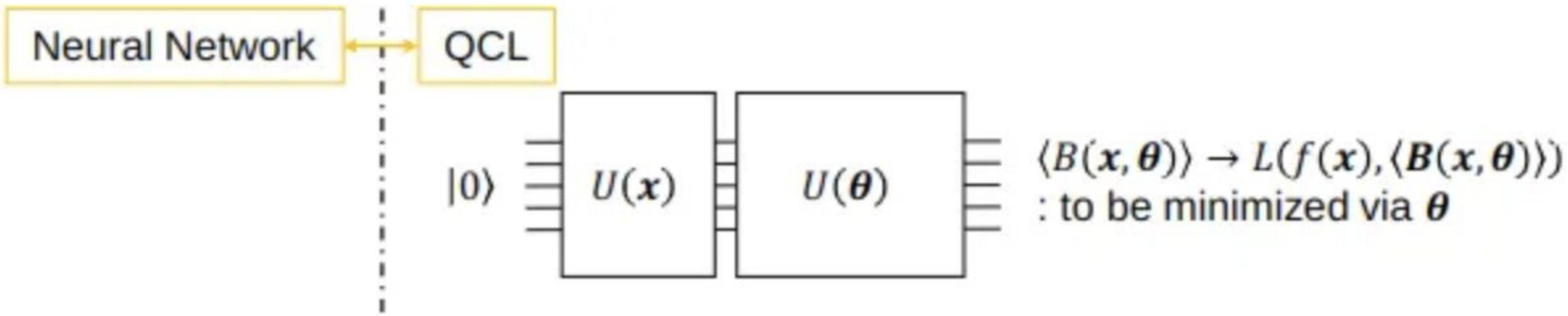
黃茂勛 鄭睿宏 周彥綸

大綱

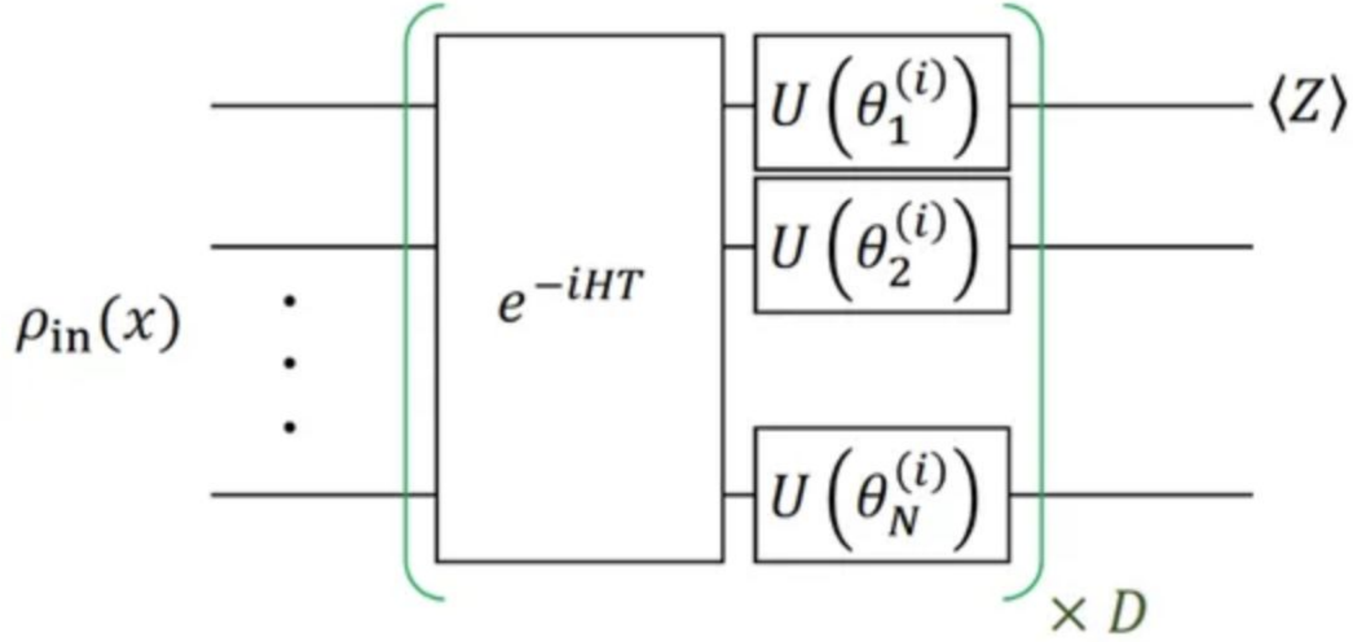
- VQC
- Classical CNN
  - train, evaluation
- Quantum CNN
  - train, evaluation
- Discussion

VQC

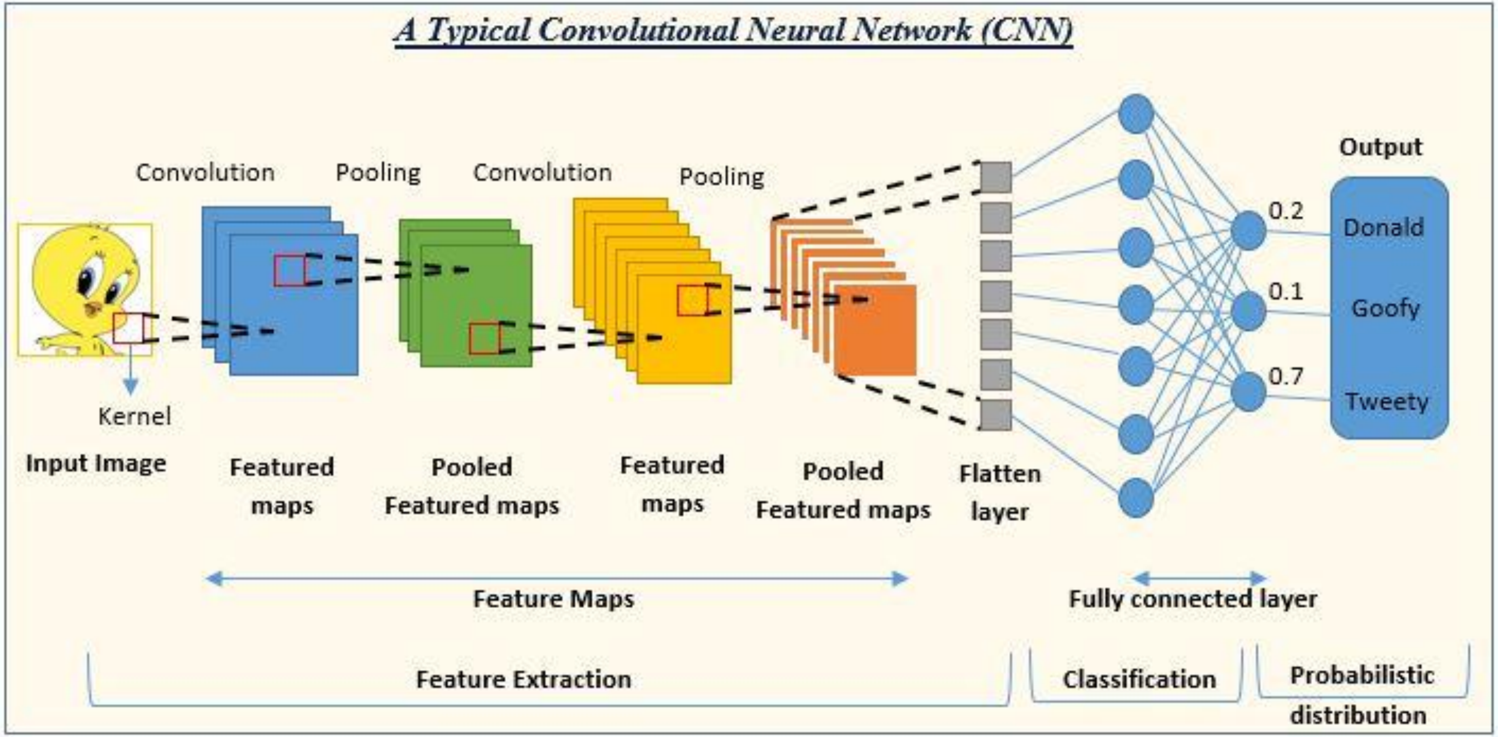
- Quantum Circuit Learning (經典論文)
- VQC 介紹文章



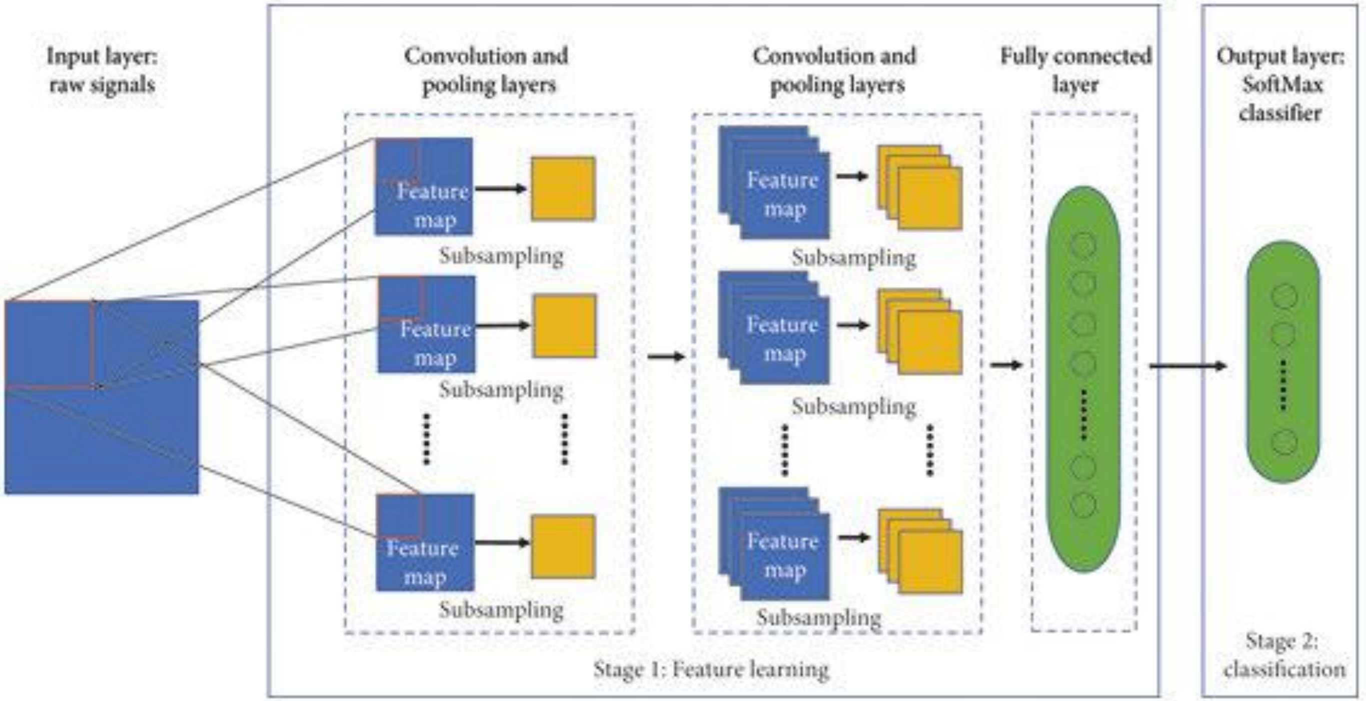
使用U(θ)的轉換得到output state。



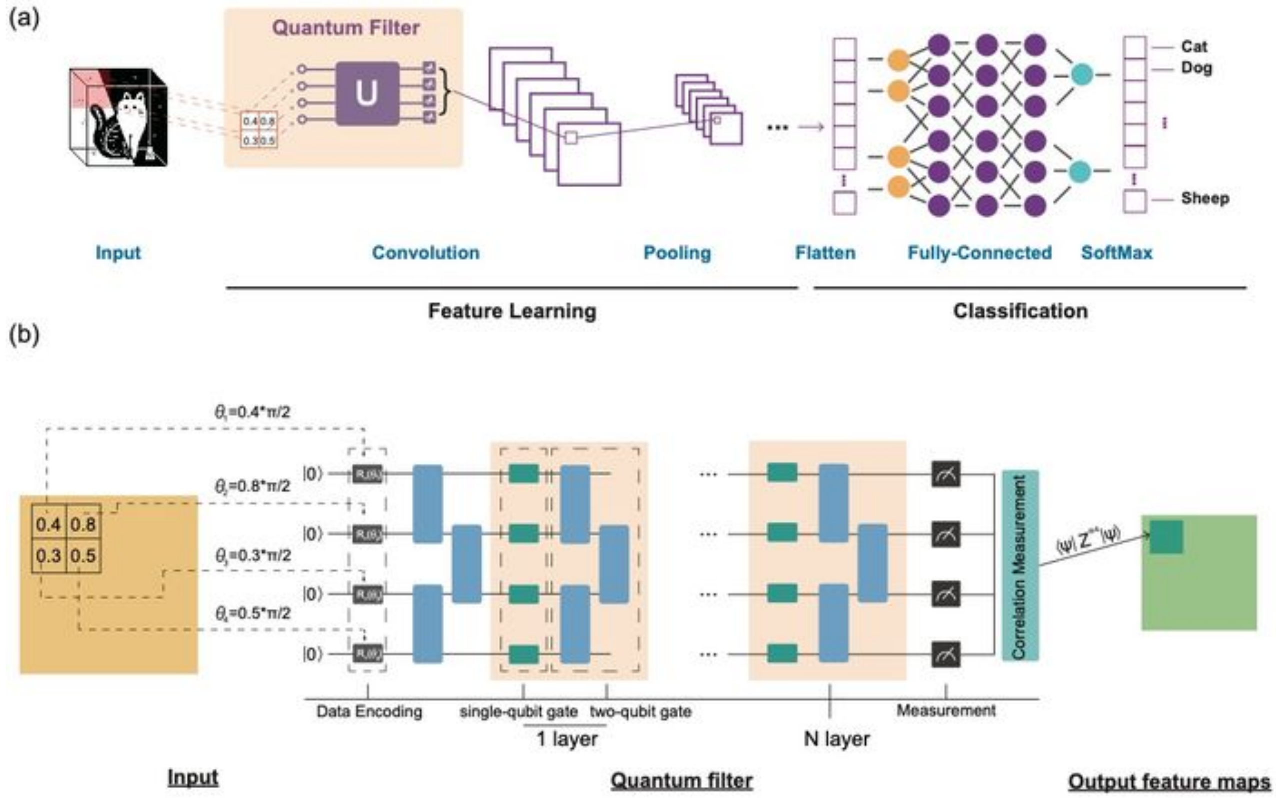
Classical CNN & Quantum CNN



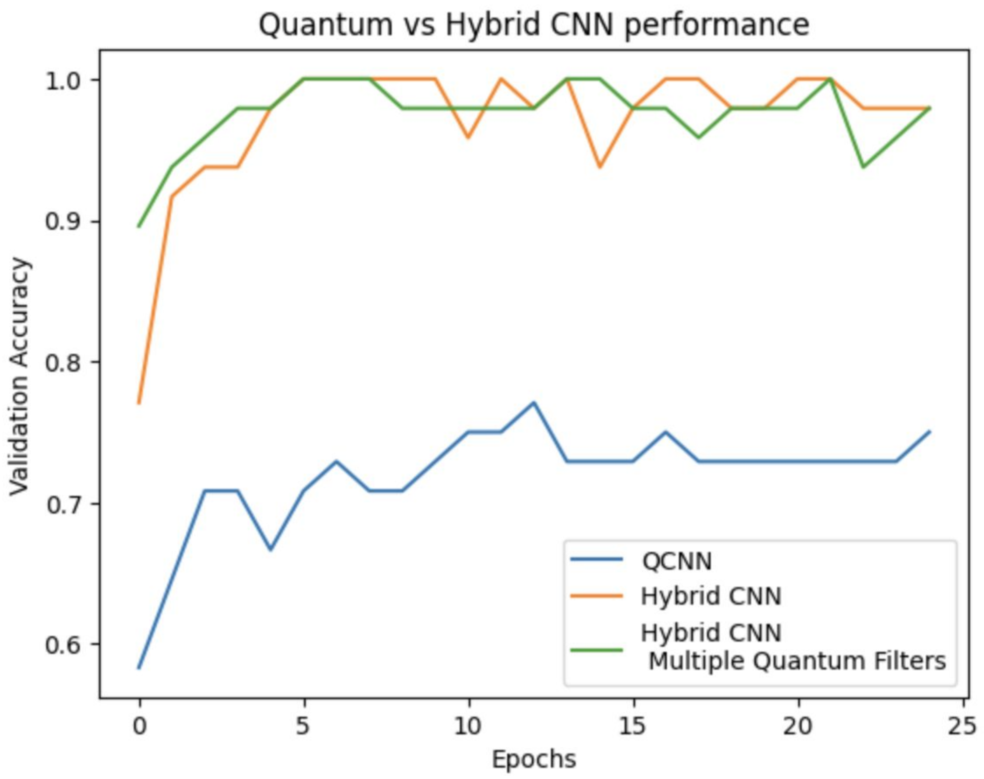
Classical CNN & Quantum CNN



Classical CNN & Quantum CNN



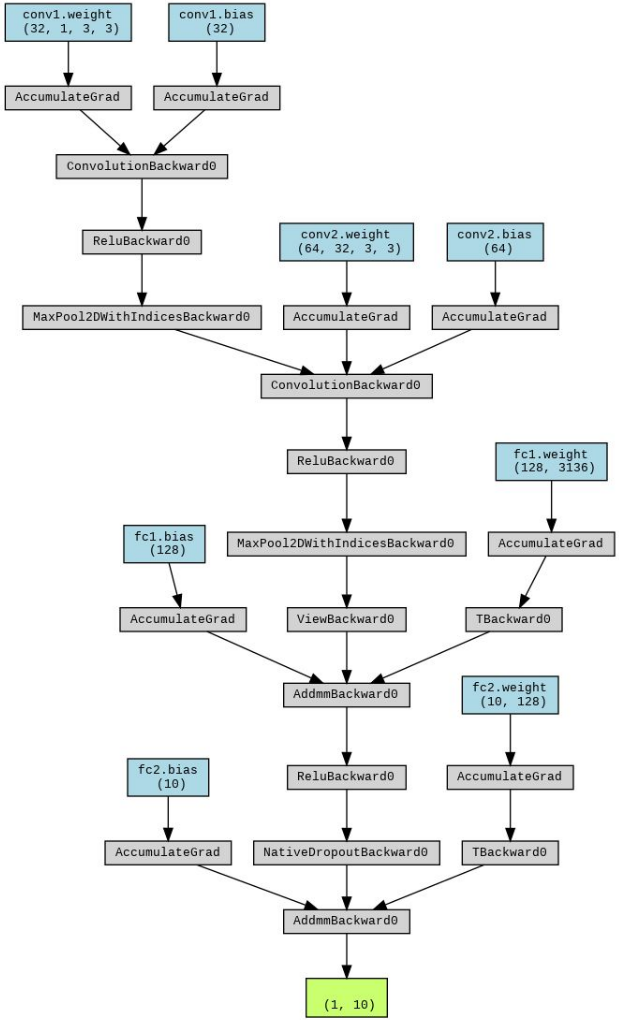
Hybrid vs. Pure



Train Classical CNN & QCNN on MNIST

Classical CNN

```
class ClassicCNN(nn.Module):
    def __init__(self):
        super(ClassicCNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, stride=1, padding=1) # 32 filters
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)
        self.dropout = nn.Dropout(p=0.25)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x))) # convolutional: [batch_size, channels, height, width]
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7) # Flatten the tensor for the fully connected layer
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```


conv1.weight
(32, 1, 3, 3)
conv1.bias
(32)
AccumulateGrad
AccumulateGrad
ConvolutionBackward0
ReluBackward0
conv2.weight
(64, 32, 3, 3)
conv2.bias
(64)
MaxPool2DWithIndicesBackward0
AccumulateGrad
AccumulateGrad
ConvolutionBackward0
ReluBackward0
fc1.weight
(128, 3136)
fc1.bias
(128)
MaxPool2DWithIndicesBackward0
AccumulateGrad
AccumulateGrad
ViewBackward0
TBackward0
AddmmBackward0
fc2.weight
(10, 128)
fc2.bias
(10)
ReluBackward0
AccumulateGrad
AccumulateGrad
NativeDropoutBackward0
TBackward0
AddmmBackward0
(1, 10)

Classical CNN (trained on MNIST)

```
num_epochs = 10

for epoch in range(num_epochs):
    running_loss = 0.0
    for i, (inputs, labels) in enumerate(trainloader):
        inputs, labels = inputs.to(device), labels.to(device)

        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs, labels)

        # Backward pass and optimization
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if i % 100 == 99:    # Print every 100 mini-batches
            print(f"[{epoch + 1}, {i + 1}] loss: {running_loss / 100:.3f}")
            running_loss = 0.0

print("Finished Training")
```

epoch=10
train_loss = 0.022

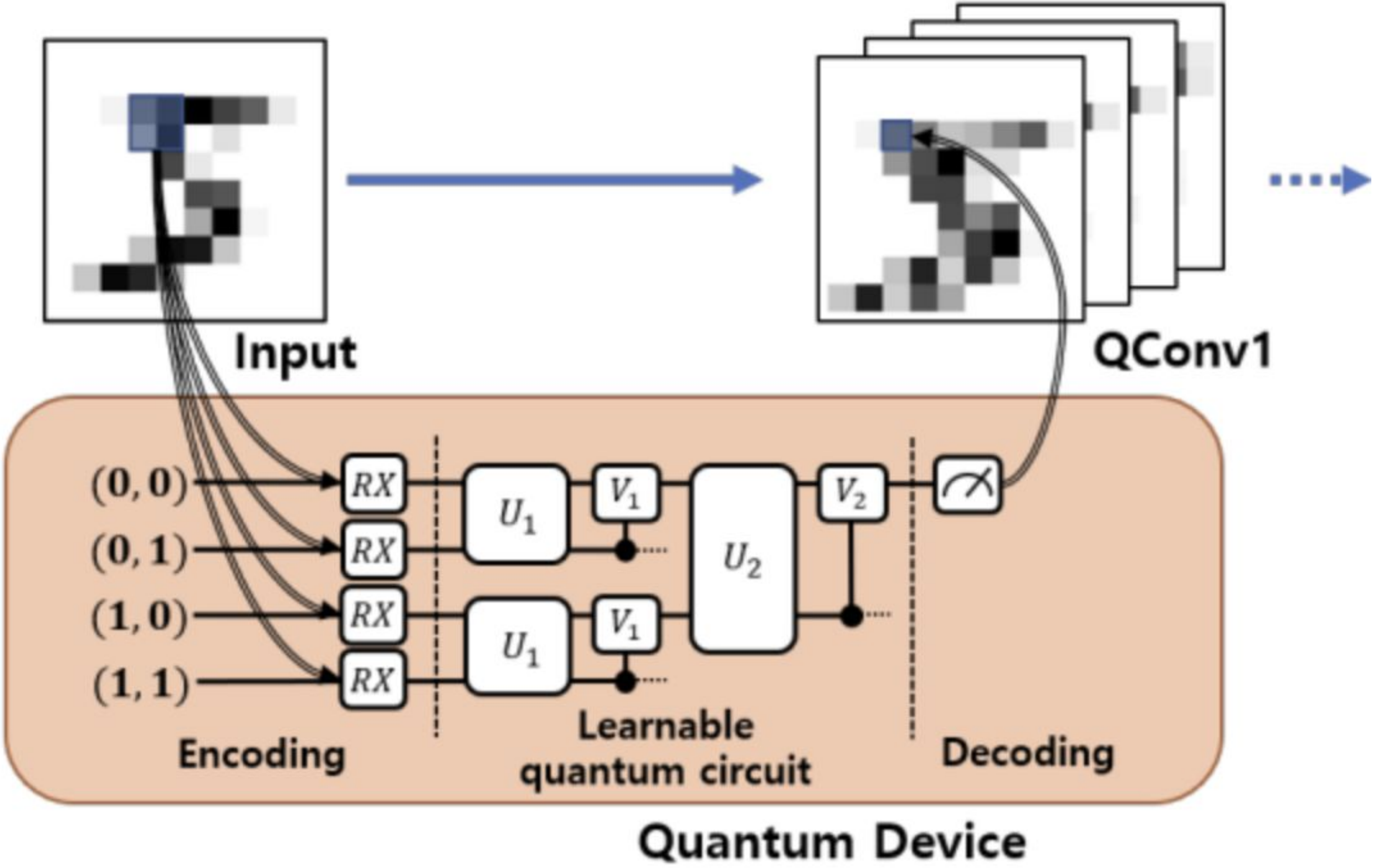
Classical CNN (trained on MNIST)

```
correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in testloader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Accuracy of the network on the 10000 test images: {100 * correct / total}%")
```

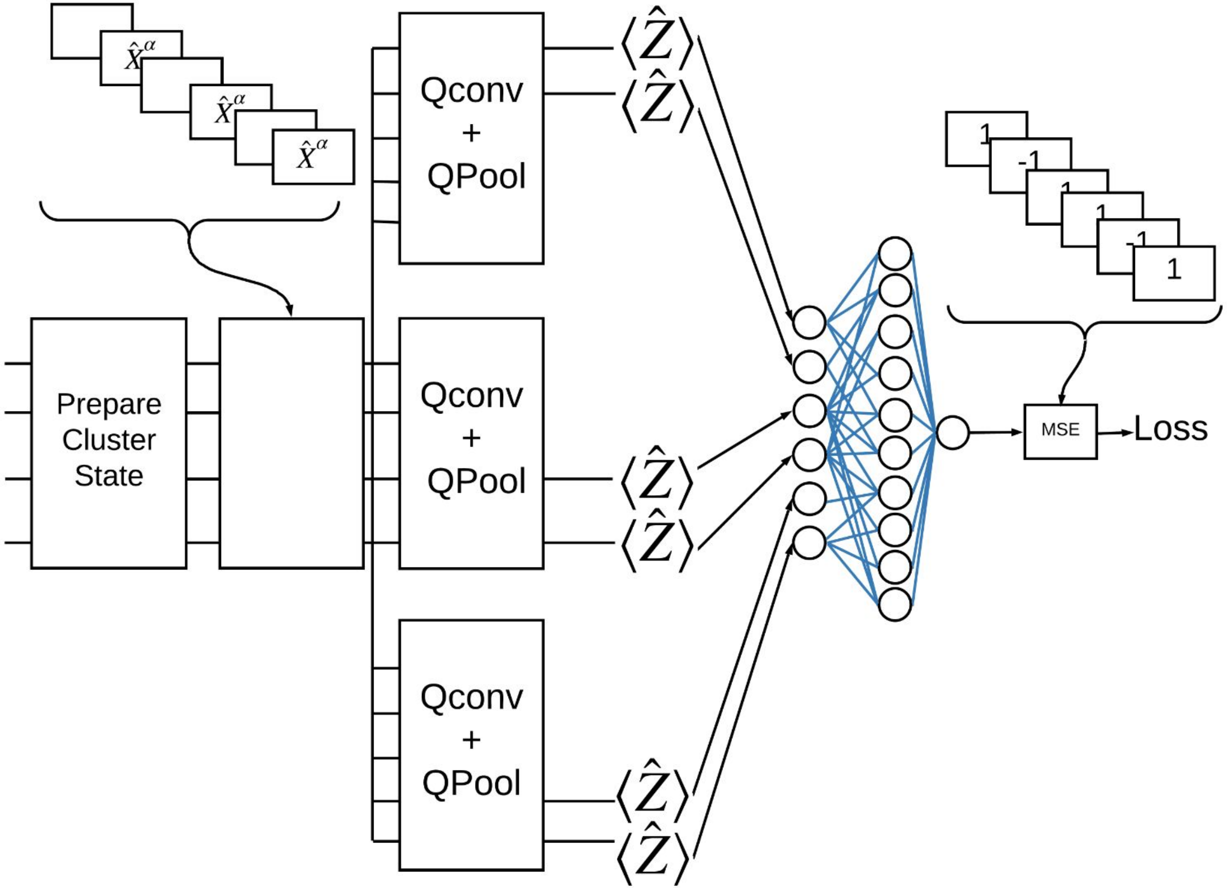
```
Accuracy of the network on the 10000 test images: 98.88%
```

不太意外！現在的CNN都可以做得很好了！

How about QCNN?



Input
QConv1
(0, 0)
(0, 1)
(1, 0)
(1, 1)
RX
RX
RX
RX
U_1
U_1
V_1
V_1
U_2
V_2
Encoding
Learnable
quantum circuit
Decoding
Quantum Device



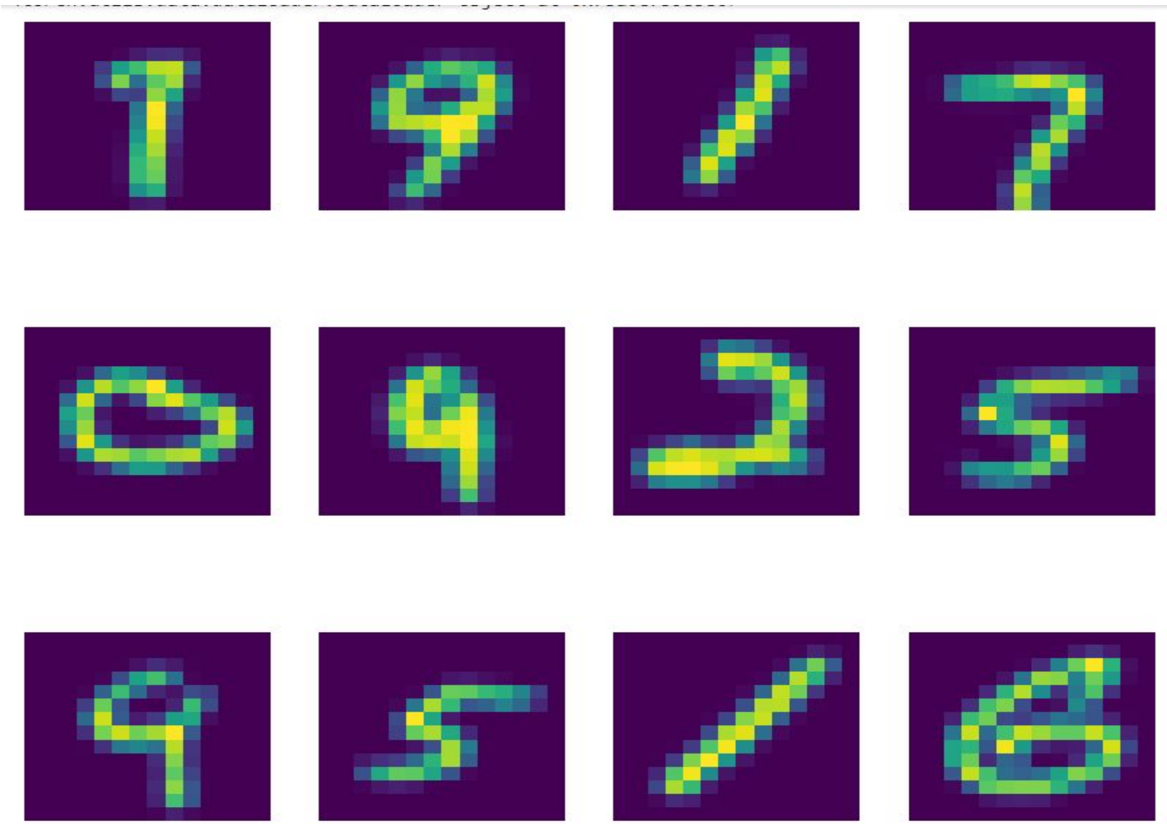
$\hat{X}^{\alpha}$
$\hat{X}^{\alpha}$
$\hat{X}^{\alpha}$
Prepare Cluster State
Qconv + QPool
Qconv + QPool
Qconv + QPool
$\langle\hat{Z}\rangle$
$\langle\hat{Z}\rangle$
$\langle\hat{Z}\rangle$
$\langle\hat{Z}\rangle$
$\langle\hat{Z}\rangle$
$\langle\hat{Z}\rangle$
1
-1
1
1
-1
1
MSE
Loss

Dataset fore-processing

```
# 下載 MNIST 資料集
trainset = torchvision.datasets.MNIST(root='./data', train=True,download=True, transform=transform)

filtered_classes = [ # 定義想要分類的類別
        '0 - zero',
        '1 - one',
        '2 - two',
        '3 - three',
        '4 - four',
        '5 - five',
        '6 - six',
        '7 - seven',
        '8 - eight',
        '9 - nine'
]

batch_size = 20 # 每次訓練的選的圖形數
epochs = 10     # 測試次數
```

```
# 挑選部分資料集
datasetNum = 4000 # 選取的資料量
trainset = trainset[:datasetNum]

# 對每個資料進行標籤
filtered_trainset = []
for data, label in trainset:
    if label in filtered_class_indices:
        new_label = 0
        for i in range(0, len(filtered_class_indices)):
            if label == filtered_class_indices[i]:
                new_label = i

        filtered_trainset.append((data, new_label))
```

Mnist dataset



Quantum circuit

```
def circuit(inputs, weights):
    var_per_qubit = int(len(inputs) / n_qubits) + 1
    encoding_gates = ['RZ', 'RY'] * ceil(var_per_qubit / 2)
    for qub in range(n_qubits):
        qml.Hadamard(wires=qub)
        for i in range(var_per_qubit):
            if (qub * var_per_qubit + i) < len(inputs):
                exec('qml.{}({}, wires = {})'.format(encoding_gates[i], inputs[qub * var_per_qubit + i], qub))
            else:  # load nothing
                pass

    for l in range(n_layers):
        for i in range(n_qubits):
            qml.CRZ(weights[l, i], wires=[i, (i + 1) % n_qubits])
            # qml.CNOT(wires = [i, (i + 1) % n_qubits])
        for j in range(n_qubits, 2 * n_qubits):
            qml.RY(weights[l, j], wires=j % n_qubits)

    _expectations = [qml.expval(qml.PauliZ(i)) for i in range(n_qubits)]
    return _expectations
    # return qml.expval(qml.PauliZ(0))
```

using draw() to visualize circuit in Pennylane

```
@qml.qnode(dev)
def circuit(params):
    qml.RY(params[0], wires=0)
    qml.RY(params[1], wires=1)
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.PauliZ(1))

params = [0.1, 0.3]
```

In order to visualize your circuit, you can define a **drawer** that comes from your corresponding circuit by using `qml.draw()`, and visualize your circuit by passing it the circuit parameters.

```
>>> drawer = qml.draw(circuit)
>>> print(drawer(params))

0: ──RY(0.1)──╭C──┤
1: ──RY(0.3)──╰X──┤ ⟨Z⟩
```

Or you can also visualize it directly by providing your circuit and parameters:

draw() is different between qnode and qnn

```
qnode = qml.QNode(circuit, dev, interface='torch', diff_method='best')
weight_shapes = {"weights": (n_layers, 2 * n_qubits)}
print(weight_shapes)

ql1 = qml.qnn.TorchLayer(qnode, weight_shapes)
print(ql1.weights); print('\n')

drawer = qml.draw(circuit)
print(drawer(params, ql1.weights) + '\n')
```

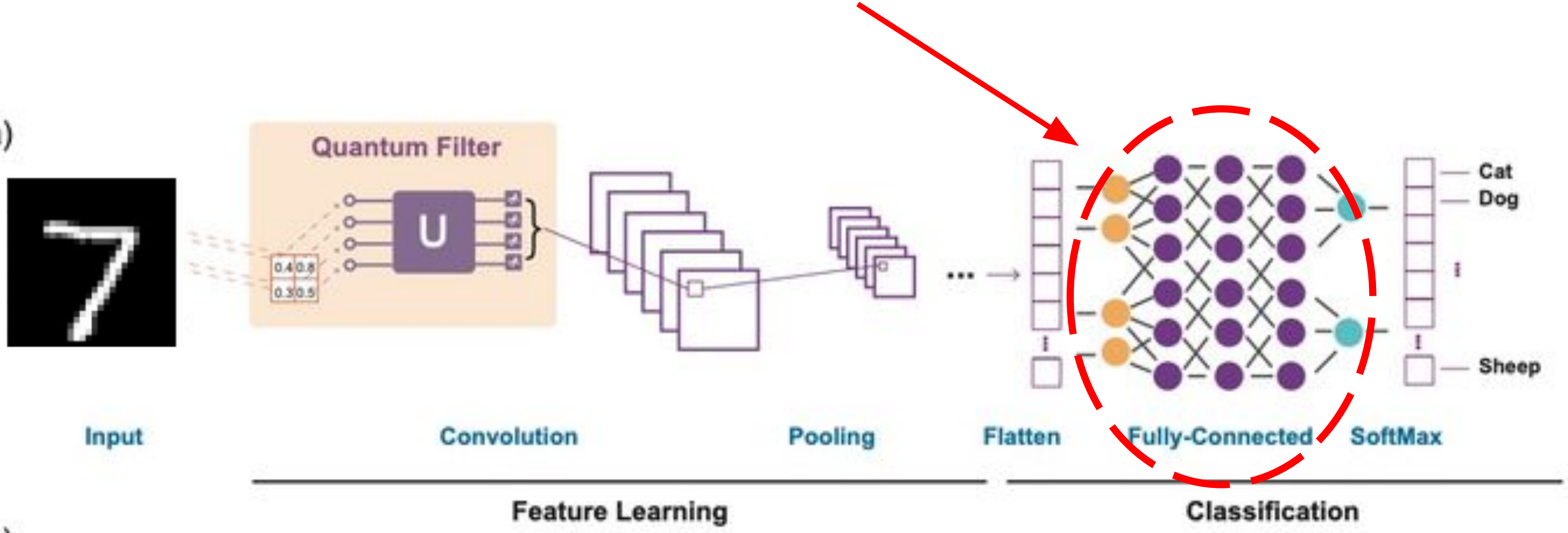
```
{'weights': (1, 8)}
Parameter containing:
tensor([[5.5948, 6.0205, 0.9626, 1.4835, 0.1876, 4.8642, 6.2060, 1.2725]],
       requires_grad=True)


0: ──H─ ╭C──────────────────────────────────────╭RZ(1.48)──RY(0.19)─┤ ⟨Z⟩
1: ──H─ ╰RZ(5.59)─ ╭C───────────────────────────│──────────RY(4.86)─┤ ⟨Z⟩
2: ──H───────────── ╰RZ(6.02)─ ╭C───────────────│──────────RY(6.21)─┤ ⟨Z⟩
3: ──H─────────────────────────╰RZ(0.96)─ ╰C──────────────RY(1.27)─┤ ⟨Z⟩
```

Neural Network

```
class Net(nn.Module):
        # define nn
        def __init__(self):
                super(Net, self).__init__()
                self.ql1 = Quanv2d(kernel_size=kernel_size, stride=stride)
                self.conv1 = nn.Conv2d(4,16,3,stride=1)
                self.fc1 = nn.Linear(16*4*4, n_class * 2)
                self.lr1 = nn.LeakyReLU(0.1)
                self.fc2 = nn.Linear(n_class * 2, n_class)

        def forward(self, X):
                bs = X.shape[0]
                X = X.view(bs, 1, image_x_y_dim, image_x_y_dim)
                X = self.ql1(X)
                X = self.lr1(self.conv1(X))
                X = X.view(bs,-1)
                X = self.fc1(X)
                X = self.lr1(X)
                X = self.fc2(X)
                return X
```

Neural Network in Quantum CNN

(a)



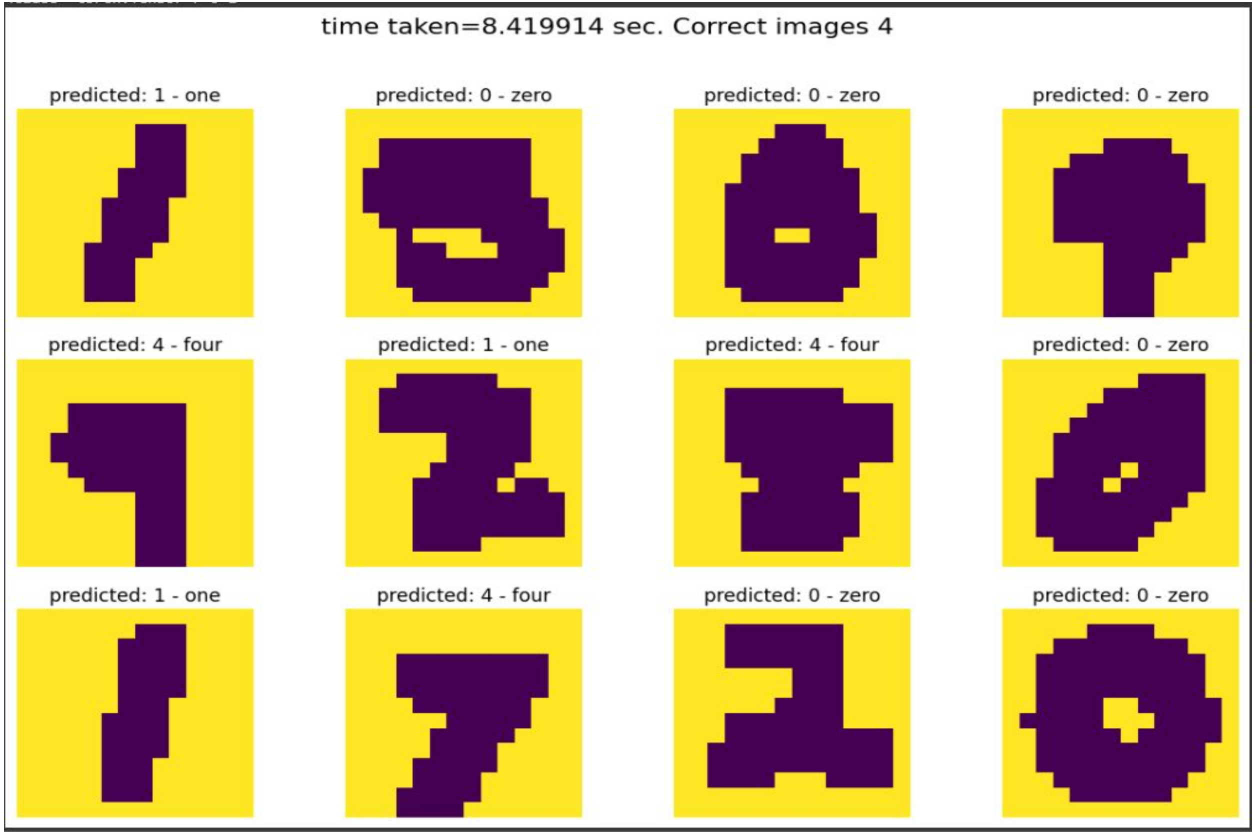
(b)

Result: train for 4hr / 4 epoch(GPU)、13hr / 8 epoch(CPU)

```
tensor([1, 0, 8, 2, 9, 0, 6, 0, 3, 5, 3, 7, 2, 6, 1, 9])
train epoch[4/10] loss:1.624: 100%|██████████|
tensor([1, 4, 3, 8, 0, 9, 1, 6, 8, 7, 2, 8, 9, 6, 3, 2])
train epoch[4/10] loss:1.645: 100%|██████████|
100%|██████████| 50/50 [06:40<00:00,  8.00s/it]
[epoch 4] train_loss: 1.750  val_accuracy: 0.549
```

```
tensor([1, 6, 4, 9, 7, 0, 3, 9, 6, 8, 3, 5, 3, 7, 5, 5])
train epoch[8/10] loss:1.170: 100%|██████████| 199/200 [38:17<00:11, 11.50s/it]
tensor([7, 0, 1, 9, 2, 7, 6, 9, 6, 4, 3, 8, 6, 9, 0, 9])
train epoch[8/10] loss:1.231: 100%|██████████| 200/200 [38:29<00:00, 11.55s/it]
100%|██████████| 50/50 [06:51<00:00,  8.22s/it]
[epoch 8] train_loss: 1.194  val_accuracy: 0.661
```

Qualitative Analysis



Quantitive Analysis

	Classical CNN (epoch=10)	QCNN (epoch=4)	QCNN (epoch=8)
Time	about 2mins	4hr (GPU)	13hr (CPU)
Loss	0.0022	1.750	1.194
Accuracy	0.988	0.549	0.661

Discussion

- Simulated QCNN can have a **good performance** in Picture Classifier
- However, it **takes so long** for training. We even only use 2 circuit to simulate the kernels here. (It may be relateed to the packages)
- Because of the limitation of quantum circuit (time, error, …), we can combine classical and quantum components to get better results. (**Hybrid Approach**)
- We can use **different circuit structure** to simulate the components in the classical NN structures.

程式碼與模型

- Github 連結：https://github.com/RyanCheng98153/QC-CNN-mnist

分工

- 資管三甲 110306019 黃茂勛
  - Classical CNN
  - 簡報製作
  - 報告
- 資科三 110703007 鄭睿宏
  - Quantum CNN
  - 簡報製作

參考資料

- https://medium.com/@raywu0/%E7%B0%A1%E4%BB%8B-quantum-variational-circuits-7f867ca02811
- https://medium.com/@chs.li.work/paper-review-quantum-circuit-learning-and-quantum-variational-circuits-introduction-b147aefbe62f
- https://www.eurekalert.org/multimedia/829458
- https://www.tensorflow.org/quantum/tutorials/qcnn