Quantum Neural Network with tensorflow and cirq

https://youtu.be/XEsoJ9zGcTY

HSU 한성대학교 HANSUNG UNIVERSITY

CryptoCraft LAB
https://crypto.modoo.at

Contents

CryptoCraft LAB

Tensorflow-Quantum (TFQ)

- **Hybrid Quantum-Classical ML model을 위한 라이브러리**
- Cirq에서 설계된 양자 컴퓨팅 알고리즘 및 로직 통합
- 고성능 양자 회로 시뮬레이터 및 **기존 TensorFlow API와 호환**되는 양자 컴퓨팅 기본 요소를 제공

Cirq

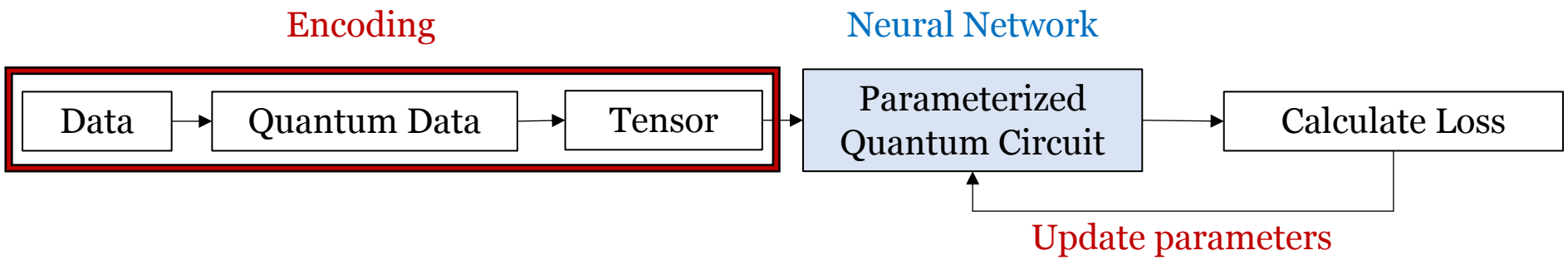
- **양자 컴퓨터 및 시뮬레이터에서 실행하기 위한 Python 라이브러리**
  - 양자 회로 구성, 조작, 최적화
  - 시뮬레이션, 회로 프린트 등
- Google꺼..

Quantum Neural Network

- **양자역학적 현상을 활용**하여 설계한 neural network
- **종류**

  그냥 Quantum NN, Classical – Quantum Hybrid NN 등
- **전체적인 과정**

  Encoding → Run parameterized quantum circuit → calculate loss → back propagation (parameter update)



Cirq – Qubit, Circuit and Quantum Hardware

- 다음과 같은 방법으로 **qubit 할당** 가능
  - 이름 붙여서 추상적인 알고리즘에 유용

```
q0 = cirq.NamedQubit('source')
q1 = cirq.NamedQubit('target')
```

  - 정확히 어딘진 모르겠는데 3이면 3개의 qubit 할당

```
q3 = cirq.LineQubit(3)
```

```
q0, q1, q2 = cirq.LineQubit.range(3)
```

  - 그리드에서 (4,5)에 위치하는 qubit

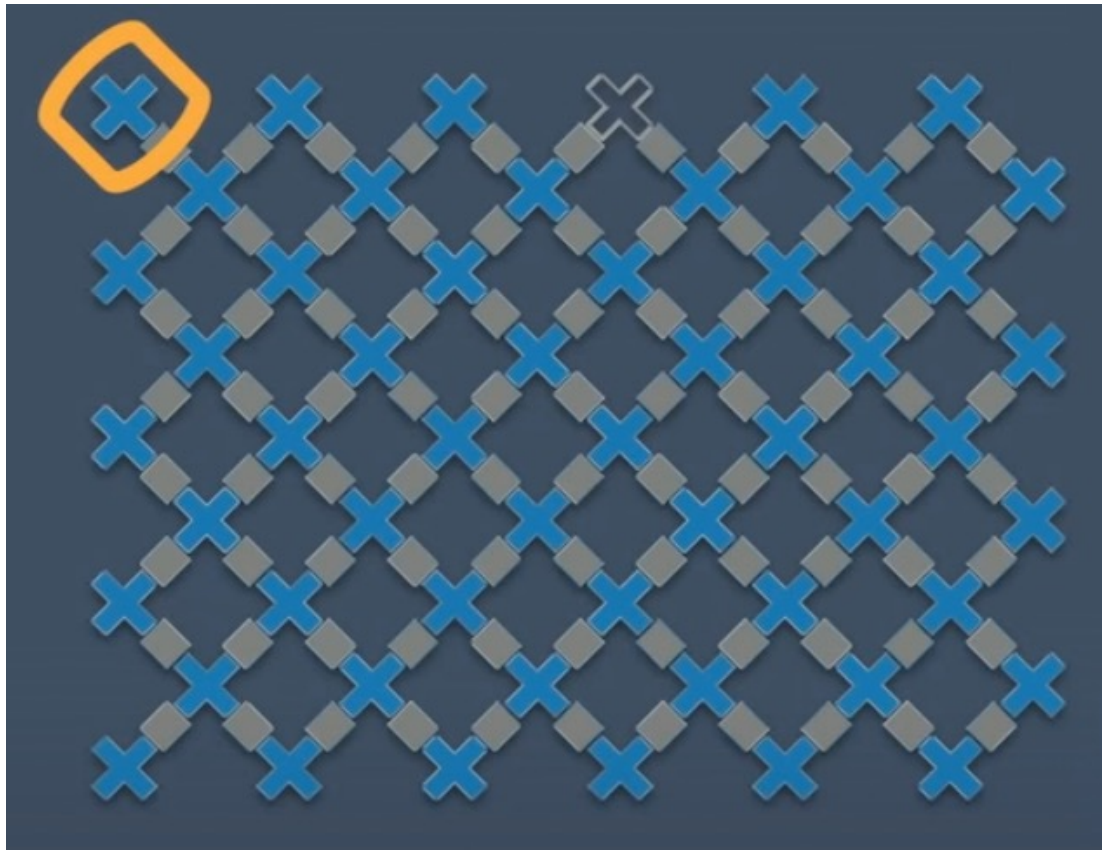
```
q4_5 = cirq.GridQubit(4,5)
```

  - 그리드에서 (0,0)~(3,3) 까지 총 16개의 qubit 할당

```
qubits = cirq.GridQubit.square(4)
```

- **Circuit 생성**

```
cir = cirq.Circuit()
```



*GridQubit은 2 차원 정사각형 격자 형태의 qubit

Cirq – Qubit and Quantum Hardware

- **cirq에서 제공하는 quantum hardware**

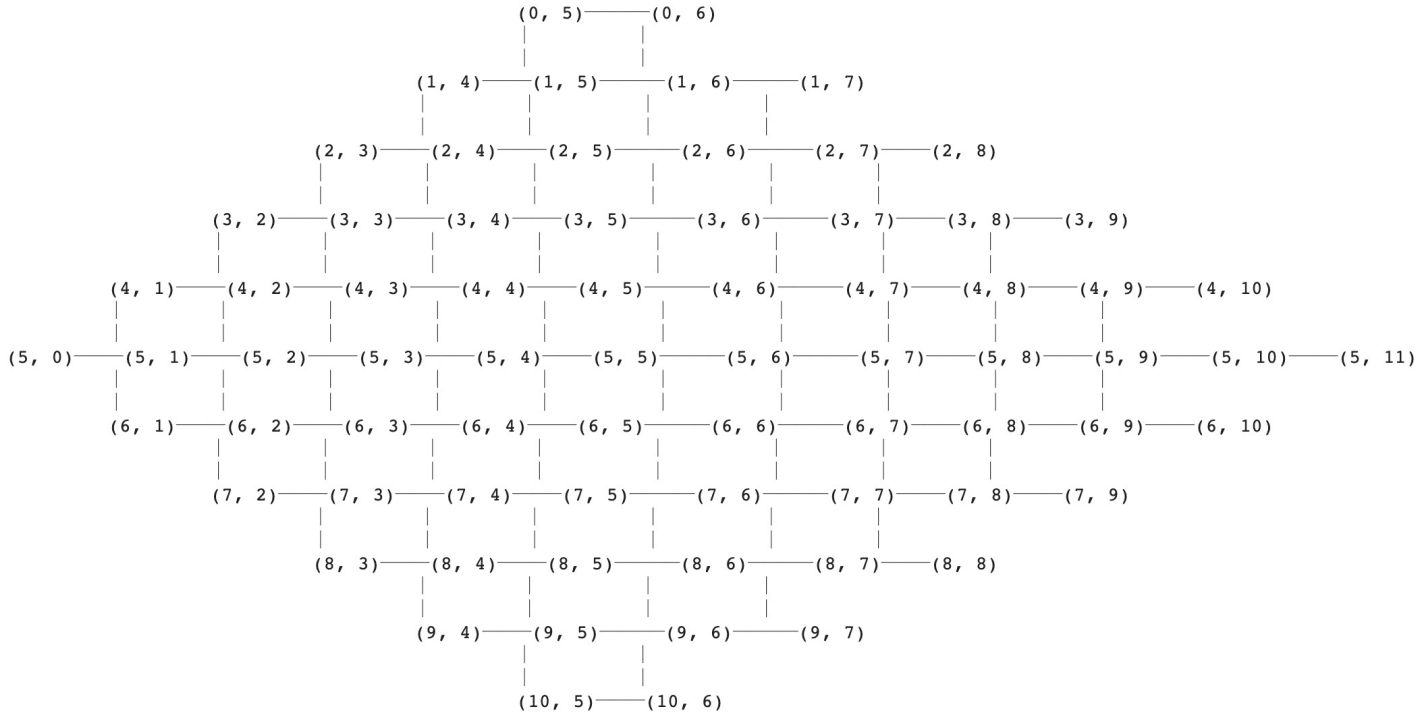
```
print(cirq_google.Foxtail)
```

```
(0, 0)———(0, 1)———(0, 2)———(0, 3)———(0, 4)———(0, 5)———(0, 6)———(0, 7)———(0, 8)———(0, 9)———(0, 10)
|        |        |        |        |        |        |        |        |        |
|        |        |        |        |        |        |        |        |        |
(1, 0)———(1, 1)———(1, 2)———(1, 3)———(1, 4)———(1, 5)———(1, 6)———(1, 7)———(1, 8)———(1, 9)———(1, 10)
```

Bristlecone : **72-qubit** 제공

해보진 않았지만 지금까지의 생각으로는
72-bit 평문까지는 커버 가능한 것 같음

```
print(cirq.google.Bristlecone)
```



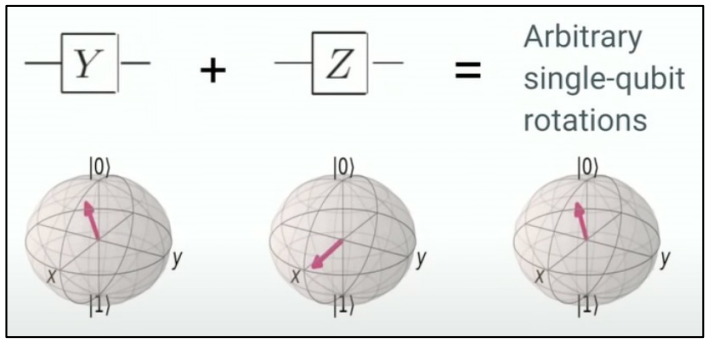
Quantum data

- 생성된 **qubit에는 quantum data**가 들어가야함

*양자 상태에 상태 변화를 가할 수 있는 것은 **unitary matrix** 곱셈뿐
→ 양자 게이트는 unitary matrix
ex) X, Y, Z, H, Rx, Ry, CX, CCX …

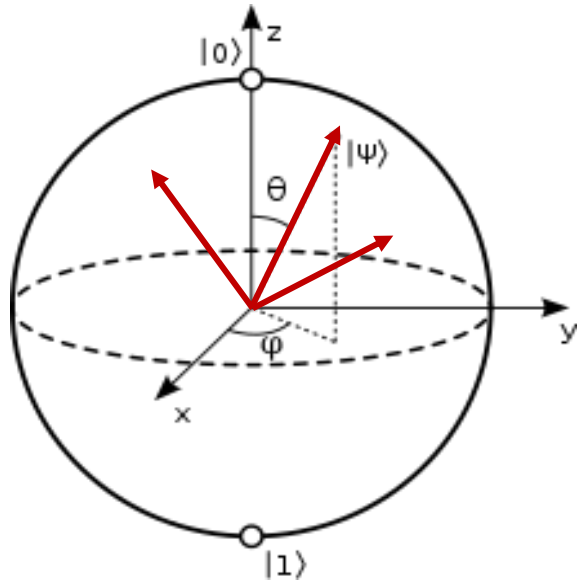
- quantum data 생성을 위해 원래 상태에 **rotation 연산**
  → 가능한 경우의 수를 한번에 연산하기 위함 → rotation 통해 불확실성을 주어 quantum data 생성

- ***Rx, Ry, Rz* gate 사용 가능**
  → Rx gate : x축 기준 rotation (y, z의 rotation ; 좌표 변환)
  → **하나의 qubit에 대한 random rotation → 두 축에 대한 rotation 필요**




```
def convert_data(data, qubits, test=False):
    cs = []
    for i in data:
        cir = cirq.Circuit()
        for j in qubits:
            cir += cirq.rx(i[0] * np.pi).on(j)
            cir += cirq.ry(i[1] * np.pi).on(j)
        cs.append(cir)
    if test:
        return tfq.convert_to_tensor([cs])
    return tfq.convert_to_tensor(cs)
```

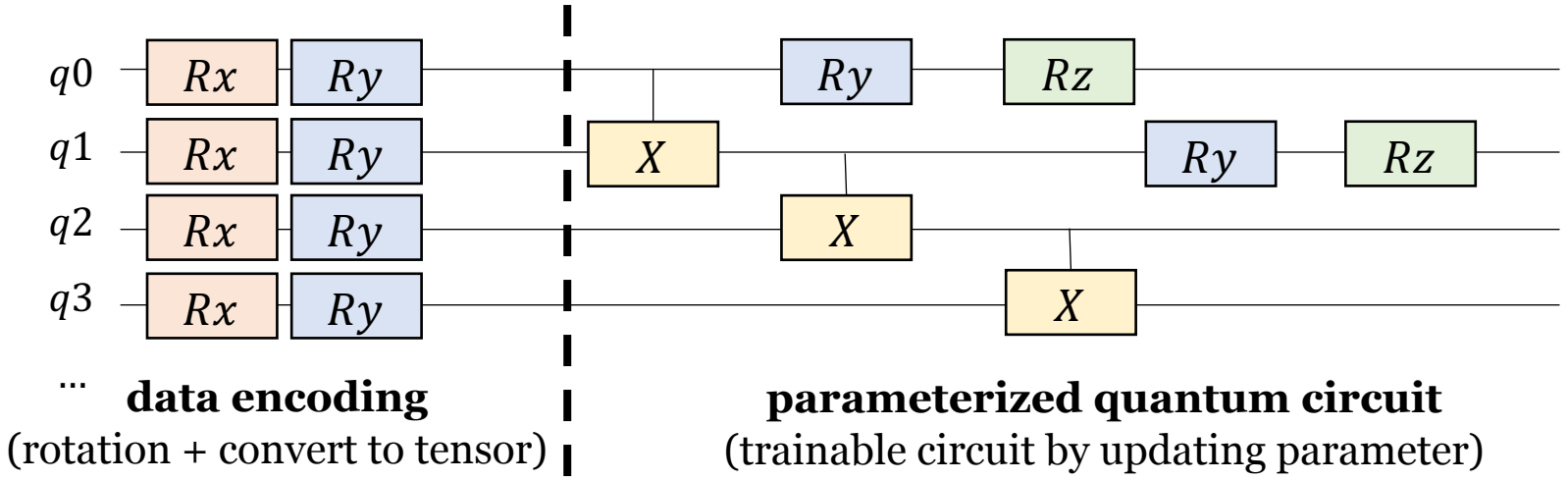
- **convert to tensor 필요**
  → quantum domain으로 인코딩한 후, tensor로 변환하기 위함 → **parameterized circuit**



Rotation(theta)

Parameterized Quantum Circuit

- Parameterized circuit (tensor로 변환된 회로) 이므로 하나의 학습 가능한 네트워크가 됨




```python
def layer(circuit, qubits, params):
    for i in range(len(qubits)):
        if i + 1 < len(qubits):
            circuit += cirq.CNOT(qubits[i], qubits[i + 1])
        circuit += cirq.ry(params[i * 2]).on(qubits[i])
        circuit += cirq.rz(params[i * 2 + 1]).on(qubits[i])
    circuit += cirq.CNOT(qubits[-1], qubits[0])
    return circuit
```

해당하는 parameter만큼 rotation
→ parameter에 의한 상태 변화

```
(0, 0): ──@──Ry(q0)──Rz(q1)──
          │
(0, 1): ──X──@──Ry(q2)──Rz(q3)──
             │
(0, 2): ─────X──@──Ry(q4)──
                │
(0, 3): ────────X──@──
                   │
```

Parameterized Quantum Circuit

- Quantum circuit (network model) generation

```
qs = [cirq.GridQubit(0, i) for i in range(13)]
d = 3
X_train, X_test, y_train, y_test = encode(x, y, qs)
c = model_circuit(qs, d)
```

- **qs** : 13개의 **qubit**, list type, 입력데이터가 13개
- **d** : circuit depth (**layer 수** 의미)
- **encode** : for quantum **data encoding**
- **layer** : **layer 내부**의 게이트 설정 (with parameter)

→ **생성된 qubit와 depth등으로 회로 생성**

```
def model_circuit(qubits, depth):
    cir = cirq.Circuit()
    num_params = depth * 2 * len(qubits)
    params = sympy.symbols("q0:%d"%num_params)
    for i in range(depth):
        cir = layer(cir, qubits, params[i * 2 * len(qubits):i * 2 * len(qubits) + 2 * len(qubits)])
    return cir
```

- 이 모델의 경우, **하나의 qubit에 Ry, Rz 적용했고 해당 연산 시 parameter가 사용**
  → 2 * len(qubit)
- **layer 수만큼 필요**하므로 depth 곱함

Parameterized Quantum Circuit

*|psi><psi|=1/2(I+aX+bY+cZ) (조건 : a^2+b^2+c^2=1)
이런 밀도행렬이 있는데 여기서 계수를 선택하여 기저 선택하여 측정
일반적으로 Z gate를 사용한다고 함

- Quantum circuit (network model) generation
  - **readout_operator**

    회로에서 정보를 추출하기 위함 → pauli measure 사용해서 **실제 state vector에 접근**

    qubit의 상태에서 **읽어낼 정보 define 필요** → tfq는 각 qubit에 **readout operator를 specify** 가능하도록 함 (Z 아니라 X,Y도 가능)*

```
readout_operators = [cirq.Z(qs[0])]
inputs = tf.keras.Input(shape=(), dtype=tf.dtypes.string)
```

→ input type : str → circuit과 tf가 처리하기 쉬운 형태가 str

  - **tfq.layers.PQC (**Parametrized Quantum Circuit)

    parameterized **quantum layer를 기존 keras의 layer에 매칭**

    → tf.keras.models.Model에 quantum layer를 쓸 수 있음

```
layer1 = tfq.layers.PQC(c, readout_operators, differentiator=tfq.differentiators.Adjoint())(inputs)
```

  - **tf와 동일하게 Model 생성 및 compile 가능**

```
vqc = tf.keras.models.Model(inputs=inputs, outputs=layer1)
vqc.compile(loss='mse', optimizer=tf.keras.optimizers.Adam(lr=3e-3))
```

Parameterized Quantum Circuit

- **tfq.layers.PQC (**Parametrized Quantum Circuit)

```
layer1 = tfq.layers.PQC(make_circuit(qubit), readout_operators, repetitions=32,
        differentiator=tfq.differentiators.ParameterShift(), initializer=tf.keras.initializers.Zeros)(inputs)
layer1 = tfq.layers.PQC(c, readout_operators, differentiator=tfq.differentiators.Adjoint())(inputs)
```

  - **readout** : qubit state 관측
  - **repetitions** : readout operator에 대해 회로를 몇 번 반복할 것인지 → 몇 번 측정할 것인지 → 아마 qiskit의 shot 개념인 것 같음
  - **differentiators :** 미분기 → for 회로의 기울기 계산 알고리즘 설정

    ParameterShift

    - 각 게이트를 2개의 고유 값으로 분해

    - forward의 2배의 back propagation 필요 → 하나의 gradient 계산 위해 **2배의 parameter**가 필요 → 느림

    Adjoint()

    - real quantum hardware에서는 안 쓰고 **simulator에서만 사용**

    - **classical back propagation** 방식 사용 → parameter shift에 비해 **매우 빠름** (forward pass만큼만 필요 (1번))

QNN Training

- **Training**

  classical NN (keras)와 호환

  → optimizer, loss function, fit 함수, epochs, batch_size 등 그냥 사용 가능

```
vqc = tf.keras.models.Model(inputs=inputs, outputs=layer1)
vqc.compile(loss='mse', optimizer=tf.keras.optimizers.Adam(lr=3e-3))
```

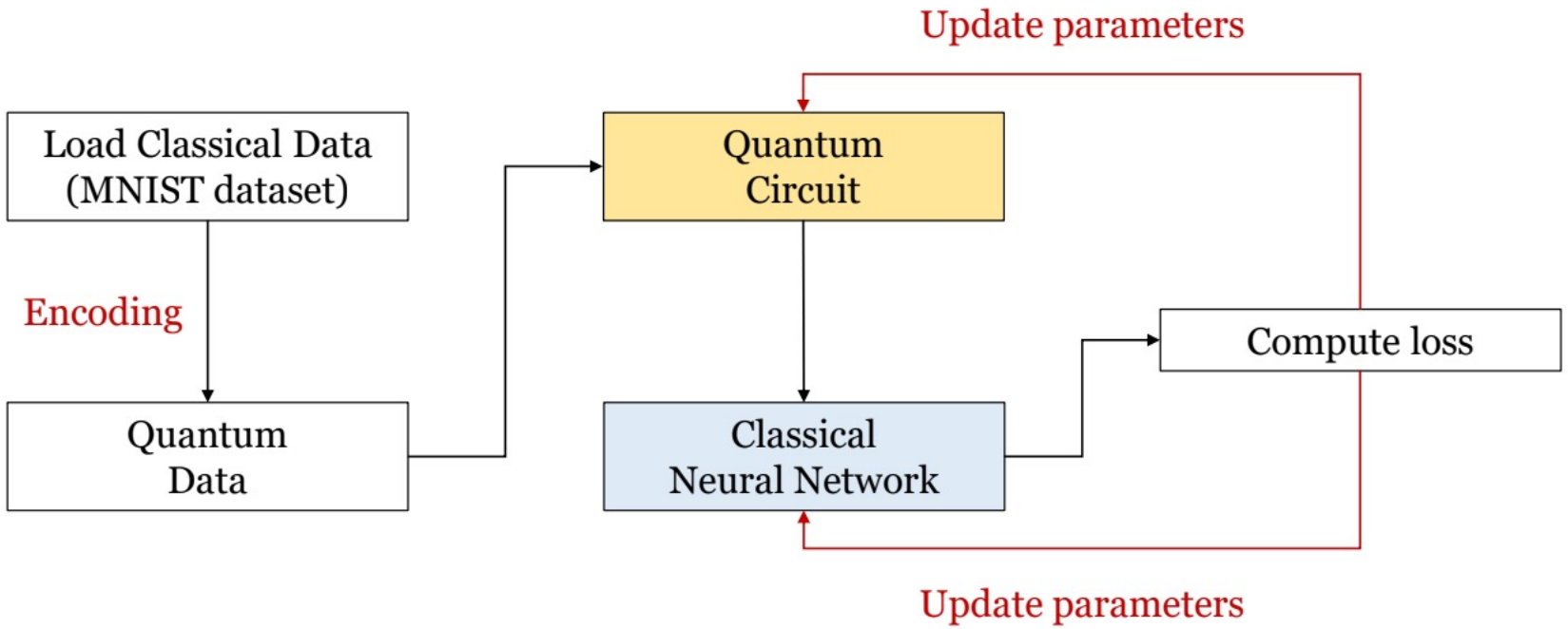
```
v_history = vqc.fit(X_train, y_train, epochs=100, batch_size=32, validation_data=(X_test, y_test), callbacks=[callback])
```

Classic-Quantum Hybrid Circuit

- **Classic-Quantum hybrid circuit**

  Quantum circuit까지 동일하고 measure하여 classic value로 결정되는 값을 classical NN의 input으로 사용

  이후 학습 과정은 classic NN과 동일 but loss 계산 후, Quantum circuit, classical NN의 parameter 둘 다 업데이트



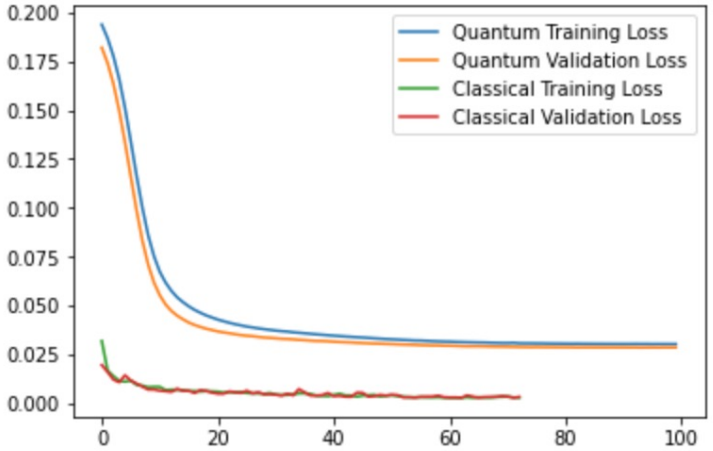
Classification

- **Quantum / Classical data & Binary / multiclass classification 가능**
  - 앞에서 본 코드들이 classification 실습에 사용했던 코드

    → qubit 수, data, quantum circuit (gate), readout operator, differentiator 등의 **세부 설정 부분만 다름** (큰 틀은 동일)
  - **실험해본 task :** (hybrid는 해커톤에서 해서 quantum NN만 수행)
    - 그냥 랜덤 rotation으로 데이터 생성하여 0 or 1 (quantum data)
    - 동심원, 달모양 등의 분류 (임의의 좌표가 어디에 속할지 추론 ; 좌표이므로 2-qubit 사용, classical data)
    - boston 집 값 추론 (13-qubit, classical data)
  - **학습해본 결과 :**
    - classical NN이 성능이 더 좋으나 문제될 정도의 성능 저하 아님 (loss 충분히 감소)
    - 이미지의 경우 qubit 제한 때문에 2x2, 4x4 이런식으로 줄여서 써야함
    - 시간은 좀 더 소요됨

      (재보진 않았는데 classic NN은 거의 1초에 1epoch, QNN은 좀 기다렸음..)
    - qiskit은 시뮬레이터로 해도 좀 더 오래 걸렸는데, shot 때문에 그랬던 것 같음

      (default가 1024고 100정도로 실험했었는데 여기서는 repetition 안 하거나 32로 했음)



더 나은 Qunatum neural network training 위한..부분..?

*⊗ : 텐서 곱

- **Qubit**
  - 적게 쓸 수록 좋음, 양자 하드웨어가 지원하는 큐비트의 수 파악 필요
- **Depth**
  - epoch 반복, repetition 같은 요소로 인해 회로 실행이 훨씬 더 느려질 것으로 생각됨 → depth가 작을수록 빠름
- **Differentiator**
  - parameter shift (느리지만 실제 하드웨어에서 가능), adjoint (빠르지만 시뮬레이터만 가능)
- **Layer re-upload (for acceleration by reducing depth)** → 나중에 더 자세히 볼 생각..
  - Layer (L) = E(parameter encoding) + U (unitary (theta)로 큐비트 상태 변화 (operation))

    → 하나의 operation에 대한 U에 대해 E가 선행 → 두 gate를 하나의 layer로 합쳐서 회로 depth를 줄임

    → $L = U(theta_i + w_i \otimes x)$

    → 가속화

- Layer re-upload 맥락으로 **data re-upload**도 가능 → **qubit 줄이고 가속화 가능**
  - qubit 하나에 2개의 data를 encoding하여 넣음

    ex : (x,y) 좌표 → 각 data의 x 좌표를 rx하여 qubit에 저장 → y에 ry 적용하여 동일 qubit에 재저장 → 하나의 qubit에 두 개의 정보

향후 계획

- 기존 NN으로 암호 분석한 논문 코드 자세하게 분석 후 Quantum NN에 응용
- 나머지는 해봐야 알 것 같습니다..

감사합니다.