Differential Cryptanalysis using QNN

https://youtu.be/cw4BO7VpEng

Differential Cryptanalysis

Neural Distinguisher

Quantum Neural Distinguisher

향후 계획

Differential Cryptanalysis

- 블록 암호의 대표적인 분석 방법
- **입력 값의 변화에 따른 출력 값의 변화를 일정 확률로 예측하여 분석**
- 선형 계층은 무조건 예측 가능하지만 **비선형은 확률적으로 예측** 가능
- S-box는 비선형 과정
- **옳은 키면 높은 확률로 입출력 차분이 만족** (특정 차분을 만족하는 경우들이 생기게 됨)
- **틀린 키면** 평문과 암호문의 관계가 사라져서 **랜덤 확률**이 될 것
- 선택 평문이 많을 수록 옳은 키를 더 잘 구별해낼 수 있음

Differential Cryptanalysis

- **차분 분석 과정**
  Step 1 : 차분 특성 찾기
  Step 2 : 차분 특성을 만족하는 평문 쌍(P, P') 찾기
  Step 3 : 라운드 키 전수 조사

Differential Cryptanalysis

차분 분포 표
(Differential Distribution Table(DDT))

Δ_I \ Δ_O	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	2	2	0	2	2	2	2	2	0	0	2
2	0	0	0	0	0	4	4	0	0	2	2	0	0	2	2	0
3	0	0	0	0	0	2	2	0	2	0	0	2	2	2	2	2
4	0	0	0	2	0	4	0	6	0	2	0	0	0	2	0	0
5	0	0	2	0	0	2	0	0	2	0	0	0	2	2	2	4
6	0	0	4	6	0	0	0	2	0	0	2	0	0	0	2	0
7	0	0	2	0	0	2	0	0	2	2	2	4	2	0	0	0
8	0	0	0	4	0	0	0	4	0	0	0	4	0	0	0	4
9	0	2	0	2	0	0	2	2	2	0	2	0	2	2	0	0
a	0	4	0	0	0	0	4	0	0	2	2	0	0	2	2	0
b	0	2	0	2	0	0	2	2	2	2	0	0	2	0	2	0
c	0	0	4	0	4	0	0	0	2	0	2	0	2	0	2	0
d	0	2	2	0	4	0	0	0	0	0	2	2	0	2	0	2
e	0	4	0	0	4	0	0	0	2	2	0	0	2	2	0	0
f	0	2	2	0	4	0	0	0	0	2	0	2	0	0	2	2

- **Step 1 : 차분 특성**
  - 차분 분포 표에서 많이 나오는 입출력 차분
  - **좋은 특성**이라면, 이러한 **차분 특성을 갖는 평문 쌍이 많아짐**
  - **이상적인 암호** 알고리즘의 경우, **랜덤 확률**이 될 것
  - 여러 연구를 통해 많이 알려져 있음
- **α'= 입력 차분, β'= 출력 차분, $(\Delta xr, \Delta y_r) = r$ 라운드 차분**

$$\alpha' = \underbrace{(\Delta x_0, \Delta y_0) \rightarrow (\Delta x_1, \Delta y_1) \rightarrow \cdots \rightarrow (\Delta x_r, \Delta y_r)}_{\text{Differential trails (차분 경로)}} = \beta',$$

- 차분 경로 : 중간 차분을 고려
  - **차분 경로는 여러 개**
    - 입력 차분에 대한 출력 차분의 경우의 수가 여러 개
    - 해당 출력 차분이 다음 라운드의 입력 차분이 되면, 여러 입력 차분에 대해 여러 출력 차분이 생겨서 그런 것 같음..
  - 차분을 만족하는 모든 차분 경로를 찾는 것이 불가능
    - 확률 계산이 어려우므로 중간 차분을 고려하여, 차분 경로의 확률을 바탕으로 차분 분석 복잡도 계산
  - **확률이 $\frac{1}{2^n}$인 경로**의 경우, **2^n 개 이상의 평문 쌍이 필요**함

Differential Cryptanalysis

- **Step 2 : 차분 특성을 만족하는 평문 쌍 (P, P')**
  - $P' = P \oplus \alpha'$, $\alpha' = P \oplus P'$
  - (P, P')에 대한 r 라운드 암호문 쌍 (C_r, C'_r) : $Cr' = C_r \oplus \beta'$ $(\beta' = \Delta xr, \Delta yr)$

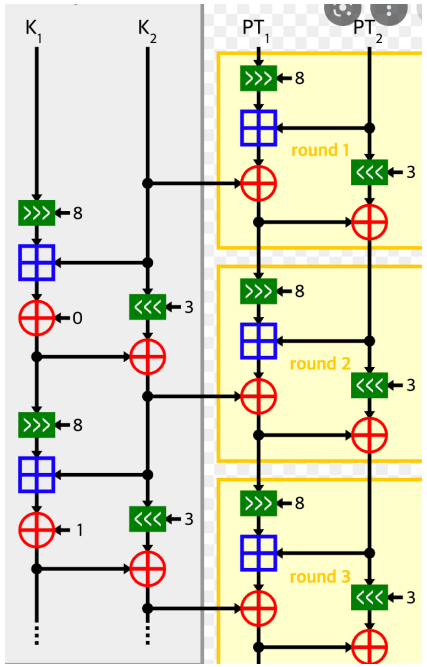
$$\alpha' = (\Delta x_0, \Delta y_0) \rightarrow (\Delta x_1, \Delta y_1) \rightarrow \cdots \rightarrow (\Delta x_r, \Delta y_r) = \beta',$$

Differential Cryptanalysis

- **Step 3 : r 라운드에 대한 라운드 키 전수 조사**
  1. 차분 만족하는 평문 쌍 (P, P') 을 암호화 하여 $r+1$ 라운드의 암호문 쌍 $(\boldsymbol{C_{r+1}}, \boldsymbol{C'_{r+1}})$을 구함
  2. 1 라운드 복호화 (전수 조사)
     $\boldsymbol{Dec(C_{r+1}, RK) \oplus Dec(C'_{r+1}, RK) = \Delta xr, \Delta yr}$ **를 만족하는** $\boldsymbol{RK}$ 찾기
  3. Speck의 경우, 2 라운드에 해당하는 라운드키를 알면 전체 키를 찾을 수 있으므로,
     $r-1$라운드에 대해서도 반복 (r 라운드 암호문 쌍을 가지고 동일 과정 반복)

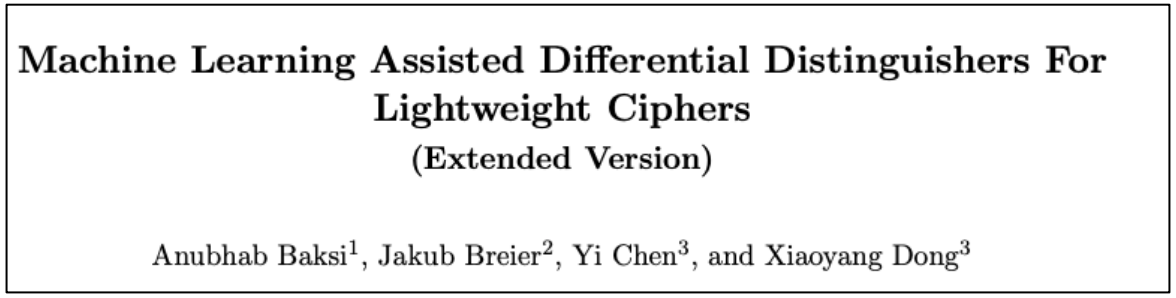
- **공격자가 알아야 하는 정보**
  - 차분을 만족하는 평문 쌍 (distinguisher)
  - $r+1$라운드의 암호문 쌍
  - $r, r-1$ 라운드의 차분

Speck 키스케줄



Neural Distinguisher

- **차분 분석을 위해서는 차분을 만족하는 평문 쌍을 찾아야함 (Step2)**
  - 이를 위해 Neural Distinguisher 사용
  - Random vs 암호문 쌍을 구별
- **Random과 암호문 쌍을 구별하는 이유**
  - 이상적인 암호는 특정 차분을 만족하지 않고 랜덤 확률이 됨
  - 특정 차분을 가지는 경우를 암호문 쌍으로 구별하고 이를 distinguisher로 사용하기 위함
- **딥러닝을 통한 분류 작업을 수행하여, 분류 확률을 계산**
  - 분류되는 클래스 / random인지 아닌지를 고려 (Distinguisher 로 사용할지 판단)


Machine Learning Assisted Differential Distinguishers For Lightweight Ciphers
(Extended Version)

Anubhab Baksi[1], Jakub Breier[2], Yi Chen[3], and Xiaoyang Dong[3]

Baksi, Anubhab. "Machine Learning-Assisted Differential Distinguishers for Lightweight Ciphers." *Classical and Physical Security of Symmetric Key Cryptographic Algorithms*. Springer, Singapore, 2022. 141-162.
해당 논문 참고했습니다..

Neural Distinguisher

- **n개의 입력 차분을 사용할 경우에 대한 분류 확률 구함**
  - **n > 1 (multiple input differences)**
    - **Input data** : 여러 입력 차분을 갖는 평문 쌍들을 암호화 한 (C, C') : P는 랜덤
    - **Label** : n개의 입력 차분
    - **Distinguisher** : $\frac{1}{n}$ 이상 → 특정 입력 차분(n 클래스 중 하나)을 가지는 것이므로 Distinguisher로 사용
    - **Random** : $\frac{1}{n}$ → Distinguisher 가 아니므로 버림

  - **$n = 1$ (one input difference)**
    - **Input data** : 하나의 입력 차분을 갖는 평문 쌍들을 암호화 한 (C, C') : P는 랜덤
    - **Label** : 0 or 1 (random or input difference)
    - **Distinguisher** : $\frac{1}{2}$ 이상 → Distinguisher로 사용
    - **Random** : $\frac{1}{2}$ → Distinguisher 가 아니므로 버림

Neural Distinguisher

- **Multiple input differences model (Model1)**
  - **필요한 것**
    - Random P
    - Ciphertext $C = Enc(P)$
    - t input differences ($\delta_0 \sim \delta_{t-1}$)

  - **각 input differences(δ_i)를 사용하여 차분 만족하는 평문을 구한 후 암호화**
    - $P_i = P \oplus \delta_i$ (여러 입력 차분을 갖는 평문 쌍들 (P, Pi) 얻음)
    - $C_i = Enc(P_i)$
    - (C, C_i)를 얻은 후, 해당 데이터를 i class로 labeling

  - **각 암호문을 각 클래스 (t 개의 입력 차분)로 분류**
    - 분류 결과 : 각 클래스에 대한 확률 값으로 나옴 (ex : [0.1, 0.02, 0.07, 0.01, 0.8])
    - δ_4로 분류될 확률이 0.8이며, $\frac{1}{5}$ 이상 → distinguisher
    - 즉, 특정 입력 차분을 갖는 암호문 쌍
      → 해당 암호문 쌍을 만든 평문 쌍은 특정 입력 차분을 갖는 평문 쌍으로 적합

Algorithm 6.2: Model 1 (multiple input differences) for differential distinguisher with machine learning

procedure OFFLINE PHASE (Training)
- $TD \leftarrow (\cdot)$ ▷ Training data
- Choose random P
- $C \leftarrow \texttt{CIPHER}(P)$
- **for** $i = 0; i \leq t-1; i \leftarrow i+1$ **do**
  - $P_i \leftarrow P \oplus \delta_i$
  - $C_i \leftarrow \texttt{CIPHER}(P_i)$
  - Append TD with $(i, C_i \oplus C)$
    ▷ $C_i \oplus C$ is from class i
- Repeat from Step 3 if required
- Train ML model with TD
- ML training reports accuracy a
- **if** $a > \frac{1}{t}$ **then**
  - Proceed to Online phase
- **else** ▷ $a = \frac{1}{t}$
  - Abort

procedure ONLINE PHASE (Testing)
- $TD' \leftarrow (\cdot)$ ▷ Testing data
- Choose random P
- $C \leftarrow \texttt{ORACLE}(P)$
- **for** $i = 0; i \leq t-1; i \leftarrow i+1$ **do**
  - $P_i \leftarrow P \oplus \delta_i$
  - $C_i \leftarrow \texttt{ORACLE}(P_i)$
  - Append TD' with $C_i \oplus C$
- Test ML model with TD' to get $\mathcal{C}$
  ▷ $\mathcal{C}$ is sequence of classes by ML
- $a' =$ probability that $\mathcal{C}$ matches $(0, 1, \ldots, t-1)$
- **if** $a' = a > \frac{1}{t}$ **then**
  - $\texttt{ORACLE} = \texttt{CIPHER}$
- **else** ▷ $a' = \frac{1}{t}$
  - $\texttt{ORACLE} = \texttt{RANDOM}$
- Repeat from Step 3 if required

Neural Distinguisher

Algorithm 6.3: Model 2 (one input difference) for differential distinguisher with machine learning

procedure OFFLINE PHASE (Training)
- $TD \leftarrow (\cdot)$ ▷ Training data
- Choose random P_0, P_1 $(\neq P_0 \oplus \delta)$
- $P_2 = P_1 \oplus \delta$
- $C_i \leftarrow \texttt{CIPHER}(P_i)$, for $i = 0, 1, 2$
- Append TD with:
  $(0, C_1 \parallel C_0)$, ▷ $C_1 \parallel C_0$ is from class 0
  $(1, C_1 \parallel C_2)$ ▷ $C_1 \parallel C_2$ is from class 1
- Repeat from Step 3 if required
- Train ML model with TD
- ML training reports accuracy a
- **if** $a > \frac{1}{2}$ **then**
  - Proceed to Online phase
- **else** ▷ $a = \frac{1}{2}$
  - Abort

procedure ONLINE PHASE (Testing)
- $TD' \leftarrow (\cdot)$ ▷ Testing data
- Choose random P_0, P_1 $(\neq P_0 \oplus \delta)$
- $P_2 = P_1 \oplus \delta$
- $C_i \leftarrow \texttt{ORACLE}(P_i)$, for $i = 0, 1, 2$
- Append TD' with $C_1 \parallel C_0$ and $C_1 \parallel C_2$ in order
- Test ML model with TD' to get $\mathcal{C}$
  ▷ $\mathcal{C}$ is sequence of classes by ML
- a' = probability that $\mathcal{C}$ matches $(0, 1)$
- **if** $a' = a > \frac{1}{2}$ **then**
  - $\texttt{ORACLE} = \texttt{CIPHER}$
- **else** ▷ $a' = \frac{1}{2}$
  - $\texttt{ORACLE} = \texttt{RANDOM}$
- Repeat from Step 3 if required

- **One input differences model (Model2)**
  - **필요한 것**
    - Random P_0, P_1 (차분 관계 아님)
    - 1 input difference (δ)
  - **input differences(δ)를 사용하여 차분 만족하는 평문을 구한 후 암호화**
    - $P_2 = P_1 \oplus \delta$
    - 즉, (P_0, P_1)는 차분을 가지지 않고, (P_1, P_2) 는 특정 차분을 가짐
    - $C_i = Enc(P_i)$
    - (C_0, C_1)는 0 (random) / (C_1, C_2) 는 1 (cipher)로 labeling
  - **각 암호문을 각 클래스 (Random or Cipher)로 분류**
    - 분류 결과 : 각 클래스에 대한 확률 값으로 나옴 (ex : [0.7, 0.3])
    - Random (class 0)으로 분류될 확률이 0.7이며, $\frac{1}{2}$ 이상 → distinguisher
    - 즉, 특정 입력 차분을 갖는 암호문 쌍
      → 해당 암호문 쌍을 만든 평문 쌍은 특정 입력 차분을 갖는 평문 쌍으로 적합

Quantum Neural Distinguisher

- 해당 논문의 **Model 2**를 **quantum-classical hybrid neural network로 구현**
- **Target cipher** : Speck 32/64
- **Qiskit + Pytorch**
- 이진 분류 문제 **(random or cipher)**
- 현재, 모델 최적화는 하지 않은 상태

Quantum Neural Distinguisher

- Data set

```
def make_train_data(n, nr, diff=(0x0040,0)):
  Y = np.frombuffer(urandom(n), dtype=np.uint8); Y = Y & 1;    # 0이면 랜덤, 1이
  keys = np.frombuffer(urandom(8*n),dtype=np.uint16).reshape(4,-1);
  plain0l = np.frombuffer(urandom(2*n),dtype=np.uint16);    # p0l p0r은 랜덤
  plain0r = np.frombuffer(urandom(2*n),dtype=np.uint16);
  plain1l = plain0l ^ diff[0]; plain1r = plain0r ^ diff[1];    # p1l p1r은 차분
  num_rand_samples = np.sum(Y==0);   # 랜덤 샘플 개수

  #print(Y)
  #print(plain1l)
  #print(plain1l[Y==0])

  plain1l[Y==0] = np.frombuffer(urandom(2*num_rand_samples),dtype=np.uint16);
  plain1r[Y==0] = np.frombuffer(urandom(2*num_rand_samples),dtype=np.uint16);

  #print(plain1l[Y==0])
  ks = expand_key(keys, nr);
  ctdata0l, ctdata0r = encrypt((plain0l, plain0r), ks);    # 암호화
  ctdata1l, ctdata1r = encrypt((plain1l, plain1r), ks);
  X = convert_to_binary([ctdata0l, ctdata0r, ctdata1l, ctdata1r]);    # ct0l,
  return(X,Y);
```

- **입력 차분 = (0x0040, 0x0000)**
- 랜덤으로 16진수 생성한 후,
  lsb가 0이면 랜덤 데이터, 1이면 차분 갖는 데이터로 설정하기 위함
- num_rand_samples : Y==0인 부분을 랜덤으로 채우기 위해 개수 설정
- plain0(l,r) : 랜덤 데이터
- **plain1(l,r)의 일부 : 차분 데이터 (Y가 0인 부분은 랜덤으로 채움)**
- plain0(l,r)과 plain1(l,r)을 암호화 → ctdata0(l,r)과 ctdata1(l,r)을 생성
  → [ctdata0(l,r), ctdata1(l,r)] 생성(암호문 쌍)

plain0 (l,r)	random	random	random	random	random
plain1 (l,r)	difference	random	difference	difference	random

ctdata0 (l,r)	random	random	random	random	random
ctdata1 (l,r)	difference	random	difference	difference	random
Y (label)	1	0	1	1	0

Quantum Neural Distinguisher

- **Quantum-classical hybrid neural network**
- quantum circuit을 하나의 레이어로 사용하는 신경망
- 기존 신경망과 동일하게 작성 + **quantum layer 추가**
- quantum circuit을 측정하여 얻은 **expectation을 통해 최종 output을 생성하고 loss 계산**
- quantum circuit 통해 얻은 expectation은 하나의 값 (뒤에 언급)

```
expectation :  0.55
```

→ torch.cat을 통해 각 클래스에 대한 확률 값으로 출력
ex : [0.55, 0.45]

- 이후, **고전 신경망의 optimizer, loss function 사용**하여 학습
- 역전파 과정의 **파라미터 갱신은 parameter shift** 사용

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.input = nn.Linear(64, 128) # 입력데이터랑 맞춰줘야함
        self.dropout = nn.Dropout2d()
        self.fc0 = nn.Linear(128, 128)

        self.fc1 = nn.Linear(128, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(64, 64)
        self.fc4 = nn.Linear(64, 64)
        self.fcout = nn.Linear(128, 1)
        #self.fc4 = nn.Linear(32, 1)
        self.hybrid = Hybrid(qiskit.Aer.get_backend('aer_simulator'), 1, np.pi / 2)
        #self.hybrid = Hybrid(provider.get_backend('simulator_statevector'), 1, np.pi

    def forward(self, x):
        x = F.relu(self.input(x))
        x = self.dropout(x)
        x = F.relu(self.fc0(x))

        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)

        #x = F.relu(self.fc3(x))
        #x = F.relu(self.fc4(x))
        #x = self.dropout(x)
        x = self.fcout(x)
        x = self.hybrid(x)
        # np.array([expectation]) 이 cat 되는 거 ..
        return torch.cat((x, 1 - x), -1)  #torch.cat((x, 1 - x), -1)
```

Quantum Neural Distinguisher

- **Parameter shift**
  - forward를 수행한 후 저장된 텐서 정보 가져옴
  - shift_right / left : 1로 설정된 배열 * ctx.shift 를 한 값을 더함 / 뺌
  - 두 값을 파라미터로 넣어 회로를 수행시킨 후, 얻은 값 → expectation_right / left
  - 두 값의 차를 기울기 배열에 저장한 후, 기존 기울기와 곱하여 새 기울기로 사용

```
def backward(ctx, grad_output):
    """ Backward pass computation """
    input, expectation_z = ctx.saved_tensors
    input_list = np.array(input.tolist())

    shift_right = input_list + np.ones(input_list.shape) * ctx.shift
    shift_left = input_list - np.ones(input_list.shape) * ctx.shift

    gradients = []
    for i in range(len(input_list)):
        expectation_right = ctx.quantum_circuit.run(shift_right[i])
        expectation_left  = ctx.quantum_circuit.run(shift_left[i])

        gradient = torch.tensor([expectation_right]) - torch.tensor([expectation_left])
        gradients.append(gradient)
    gradients = np.array([gradients]).T
    return torch.tensor([gradients]).float() * grad_output.float(), None, None
```

Quantum Neural Distinguisher

- quantum circuit 설정 (qnn 세미나에 있어서 넘어가겠습니다..)
- **run 함수**
  - **transpile** : 회로 설정, 최적화
  - **qobj** : shots, parameters(theta), backend, circuit 정보 가짐
  - **job** : 해당 회로를 실행한 결과
  - **result** : 아래와 같은 정보 저장

```
result: Result(backend_name='aer_simulator', backend_version='0.10.3', qobj_id='e1694462-627f-46d3-b0b7-e862d87959fe', job_id='cec66c77-6c27-4f0a-ace3-25bd072cca83', success=True, resu
lts=[ExperimentResult(shots=1024, success=True, meas_level=2, data=ExperimentResultData(counts={'0x6': 33, '0x4': 38, '0x3': 40, '0x1d': 35, '0x14': 28, '0x7': 32, '0x1b': 34, '0x1f':
28, '0x16': 45, '0x12': 31, '0xc': 36, '0x1a': 40, '0x1': 25, '0xd': 32, '0x19': 35, '0x8': 34, '0x10': 22, '0x15': 34, '0x5': 35, '0x9': 24, '0x2': 31, '0x1e': 29, '0x1c': 41, '0x11':
32, '0xa': 33, '0xf': 33, '0xb': 32, '0x0': 32, '0x18': 34, '0x17': 18, '0xe': 22, '0x13': 26}), header=QobjExperimentHeader(clbit_labels=[['meas', 0], ['meas', 1], ['meas', 2], ['mea
s', 3], ['meas', 4]], creg_sizes=[['meas', 5]], global_phase=0.0, memory_slots=5, metadata=None, n_qubits=5, name='circuit-275', qreg_sizes=[['q122', 5]], qubit_labels=[['q122', 0],
['q122', 1], ['q122', 2], ['q122', 3], ['q122', 4]]), status=DONE, seed_simulator=3193588876, metadata={'parallel_state_update': 8, 'noise': 'ideal', 'batched_shots_optimization': Fals
e, 'measure_sampling': True, 'device': 'CPU', 'num_qubits': 5, 'parallel_shots': 1, 'remapped_qubits': False, 'method': 'statevector', 'active_input_qubits': [0, 1, 2, 3, 4], 'num_clbi
ts': 5, 'input_qubit_map': [[4, 4], [3, 3], [2, 2], [0, 0], [1, 1]], 'fusion': {'applied': False, 'max_fused_qubits': 5, 'enabled': True, 'threshold': 14}}, time_taken=0.00286638600000
00003)], date=2022-06-01T08:49:39.437529, status=COMPLETED, status=QobjHeader(backend_name='aer_simulator', backend_version='0.10.3'), metadata={'time_taken': 0.003080263, 'time_taken_
execute': 0.00290682, 'parallel_experiments': 1, 'omp_enabled': True, 'max_gpu_memory_mb': 0, 'num_mpi_processes': 1, 'time_taken_load_qobj': 0.000165569, 'max_memory_mb': 31970, 'mpi_
rank': 0}, time_taken=0.003206968307495117)
```

```python
class QuantumCircuit:

    def __init__(self, n_qubits, backend, shots):
        self._circuit = qiskit.QuantumCircuit(n_qubits)

        all_qubits = [i for i in range(n_qubits)]
        self.theta = qiskit.circuit.Parameter('theta')

        self._circuit.h(all_qubits)
        self._circuit.barrier()
        self._circuit.ry(self.theta, all_qubits)

        self._circuit.measure_all()
        # ---------------------------
        self.backend = backend
        self.shots = shots

    def run(self, thetas):

        #print(type(thetas))
        t_qc = transpile(self._circuit,
                         self.backend)
        qobj = assemble(t_qc,
                        shots=self.shots,
                        parameter_binds = [{self.theta: theta} for theta in thetas])
        job = self.backend.run(qobj)
        result = job.result().get_counts()


        # result type is result, so we have to convert to dictionary..
        counts = np.array(list(result.values()))
        #print("counts : ", counts)
        #print(result.keys())

        states = np.array(list(result.keys())).astype(float)
        #print("states : ", states)

        probabilities = counts / self.shots
        #print("probabilities: ", probabilities)

        expectation = np.sum(states * probabilities)
        #print("expectation : ", expectation)

        return np.array([expectation])
```

- **counts**
  - n번의 shots에 대해 각 states가 몇 번 나왔는지
- **states**
  - 1-qubit의 경우, 0 또는 1이 나옴
  - 2-qubit일 경우, 00~11 모두 states가 아니라 측정 시 나온 값만 states
- **probabilities** : 각 states에 대한 확률 값 (counts / shots)
- **expectation** : states에 대한 확률 (0은 곱해서 사라져서 1에 대한 확률이 나옴(1-qubit인 경우)
  → 앞에서 나온 torch.cat 통해 각 클래스에 대한 확률로 바꿔주는 것 같음

```
counts :  [45 55]
states :  [0. 1.]
probabilities:  [0.45 0.55]
expectation :  0.55
```

Quantum Neural Distinguisher

- **실험**
  - 기본 양자 회로 사용 (최적화 되지 않음)
  - 학습 데이터 10^7 개 사용 (Speck 32/64, 7 라운드)
  - 신경망의 하이퍼 파라미터 최적화 아직 진행하지 않음
- **결과**

```
Performance on test data:
    Loss: 0.7212
    Accuracy: 53.8%
```

- baksi 논문이 speck을 대상으로 하지 않았지만, 다른 암호에 대한 결과는 다음과 같음
  - **2개의 입력 차분**을 갖는 **8라운드** 암호에 대해서 약 **0.53~0.57**의 정확도 (다중 입력 차분이지만 2개이므로 이진 분류)
  - **1개의 입력 차분**을 갖는 **5라운드** 암호에 대해서 약 **0.62**의 정확도 (random or cipher 이므로 이진 분류)
  - 양자 회로 튜닝, 하이퍼파라미터 튜닝 등을 거치면 지금보다는 좀 더 높은 성능 달성할 수 있을 것으로 예상..

8-round GIMLI-Permutation (2개의 입력 차분 (Model 1))

Table 6.8: Results for architecture search with 8-round `GIMLI-PERMUTATION`

Network	Architecture	Activation Function	Number of Parameters	Training Time (s)	Accuracy
MLP I	128, 296, 258, 207, 112, 160, 2	ReLU	226,633	1267.39	0.5588
MLP II	128, 1024, 2	ReLU	150,658	1081.67	0.5569
MLP III	128, 1024, 1024, 2	ReLU	1,200,256	1162.21	0.5652
MLP IV	128, 256, 128, 64, 2	LeakyReLU	90,818	510.9	0.5478
MLP V	128, 1024, 2	LeakyReLU	150,658	934.89	0.5516
MLP VI	128, 1024, 1024, 2	LeakyReLU	1,200,256	1778.5	0.5509
MLP VII	128, 1024, 1024, 1024, 2	ReLU	2,249,858	2410.1	0.5689
CNN I	128, 128, 128, 100, 2	ReLU	128,046	2951.7	0.5000
CNN II	128, 1024, 128, 128, 100, 2	ReLU	604,206	11503.0	0.5000
LSTM I	128, 256, 128, 2	tanh/sigmoid	444,162	50460.7	0.5316
LSTM II	128, 200, 100, 128, 2	tanh/sigmoid	313,170	39825.9	0.5325

5-round Chaskey (1개의 입력 차분 (Model 2))

Rounds	Differential Probability	Accuracy
$0 \rightarrow 4$	2^{-37}	0.61618899

향후 작업

- 양자 회로 및 신경망 최적화 필요
- Model1도 2개의 입력 차분은 현재 네트워크로 가능할 것도 같음
- 암호 연구회 과제(알려진 평문 공격)를 위해서는 quantum circuit의 출력 (expectation) 변경 필요
  - Expectation이 뭔가 1-qubit에만 맞춰져 있는 것 같아서 확인 필요 (진행 중)
  - MSE 적용이 가능하도록 수정 필요 (진행 중)

감사합니다.