XOR 모델 설계 및 시각화

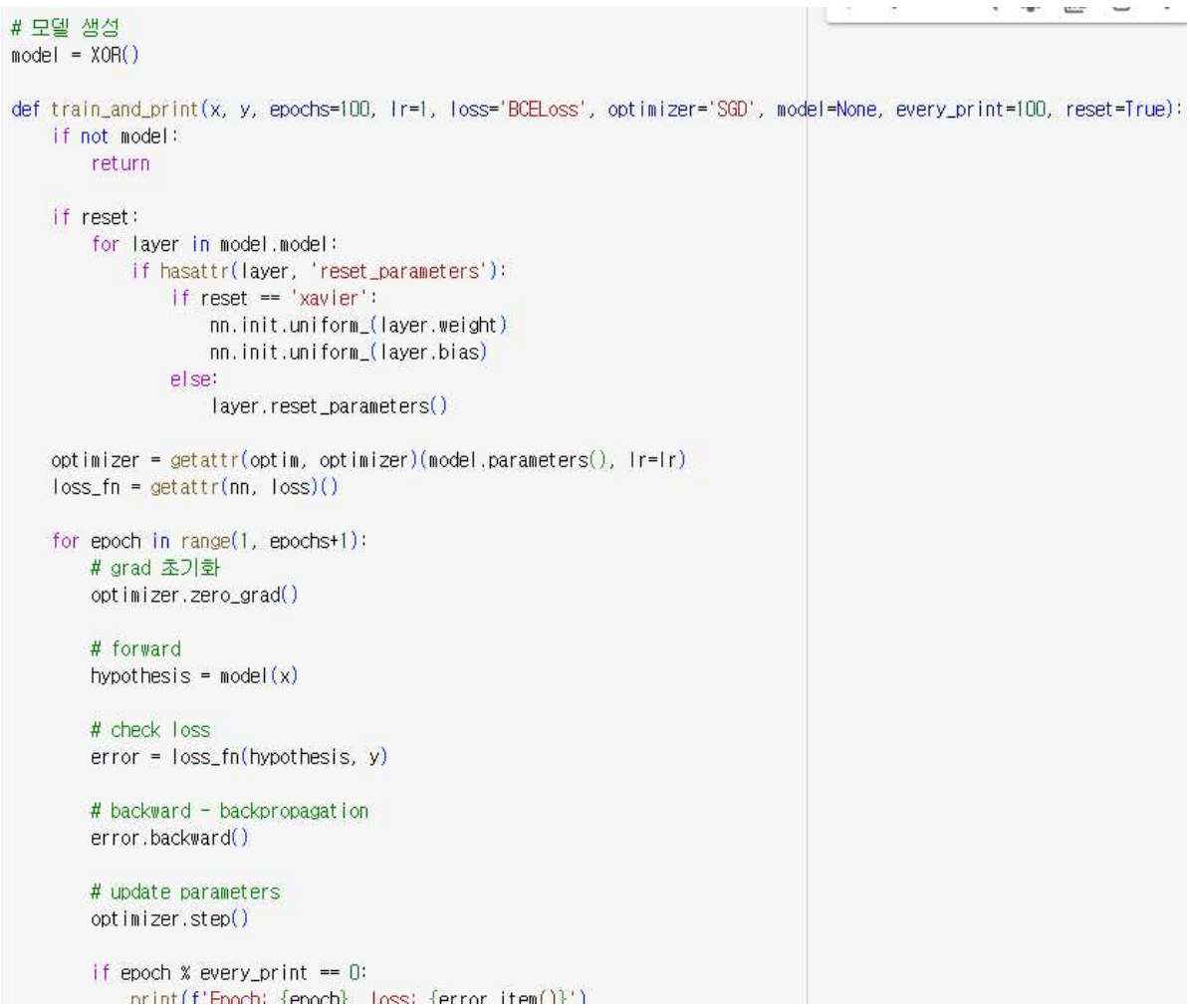
3개의 은닉층

<공통>

```
import torch
from torch import nn
from torch import optim

# X, y
X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float)
y = torch.tensor([[0], [1], [1], [0]], dtype=torch.float)

# XOR net
class XOR(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(2, 2),
            nn.Sigmoid(),
            nn.Linear(2, 2),
            nn.Sigmoid(),
            nn.Linear(2, 2),  # 히든 레이어 추가
            nn.Sigmoid(),
            nn.Linear(2, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x)
```

XOR 클래스 : 은닉층을 추가할 때 중간에 nn.Linear(2, 2), nn.Sigmoid() 코드를 추가한다. 첫 번째 인수 '2'는 입력 특성의 수, 두 번째 인수 '2'는 출력 특성의 수를 나타낸다. 즉, 입력 차원과 출력 차원이 2인 것이다.

모델의 구조
- 입력층 : 2개의 뉴런
- 은닉층 : 총 3개, 각 2개의 뉴런
- 출력층 : 1개의 뉴런
-> 활성화 함수 : 시그모이드 함수 사용



```
# 모델 생성
model = XOR()

def train_and_print(x, y, epochs=100, lr=1, loss='BCELoss', optimizer='SGD', model=None, every_print=100, reset=True):
    if not model:
        return

    if reset:
        for layer in model.model:
            if hasattr(layer, 'reset_parameters'):
                if reset == 'xavier':
                    nn.init.uniform_(layer.weight)
                    nn.init.uniform_(layer.bias)
                else:
                    layer.reset_parameters()

    optimizer = getattr(optim, optimizer)(model.parameters(), lr=lr)
    loss_fn = getattr(nn, loss)()

    for epoch in range(1, epochs+1):
        # grad 초기화
        optimizer.zero_grad()

        # forward
        hypothesis = model(x)

        # check loss
        error = loss_fn(hypothesis, y)

        # backward - backpropagation
        error.backward()

        # update parameters
        optimizer.step()

        if epoch % every_print == 0:
            print(f'Epoch: {epoch}  loss: {error.item()}')
```

train 함수 (히든레이어 3개인 코드에서는 train_and_print) :
- epochs : 학습 반복 횟수 지정
- lr : 학습률
- loss : 사용할 손실 함수를 지정, 여기서는 이진 분류의 경우에 주로 사용되는 BCELoss를 사용
- optimizer : 최적화 알고리즘을 지정, SGD(확률적 경사 하강법)를 사용
- every_print : 주기적으로 출력할 학습 손실의 에포크 간격
- reset : 매 에포크마다 모델 파라미터를 재설정할지 여부를 결정, 여기서는 매 에

포크마다 파라미터를 재설정

학습 과정:

매 에포크마다 경사를 초기화하고, 순전파를 수행하여 예측값을 계산
손실을 계산하고, 역전파를 수행하여 경사를 계산
옵티마이저를 사용하여 파라미터를 업데이트
매 every_print 에포크마다 현재 손실을 출력

<은닉층 3개에서의 최적 train>

```
# train
print("최적 train:")
train_and_print(X, y, epochs=10000, lr=0.01, optimizer='Adam', model=model, every_print=1000)
predictions = model(X)
predicted_classes = torch.where(predictions > 0.5, 1, 0)
print("Predictions:", predictions)
print("Predicted Classes:", predicted_classes)
for name, param in model.named_parameters():
    print(name, param.data)
```

Hyperparameter 결정 과정
- 학습률(learning rate): 0.01
- 최적화 기법(optimizer): Adam optimizer
- 손실 함수(loss function): 이진 교차 엔트로피 손실(Binary Cross-Entropy Loss)
- 에포크 수(epochs): 10,000

```
최적 시나리오:
Epoch: 1000, loss: 0.00987507775425911
Epoch: 2000, loss: 0.002397105097770691
Epoch: 3000, loss: 0.0010224109282717109
Epoch: 4000, loss: 0.0005195625126361847
Epoch: 5000, loss: 0.0002860369277186692
Epoch: 6000, loss: 0.00016417403821833432
Epoch: 7000, loss: 9.645272803027183e-05
Epoch: 8000, loss: 5.742339999414980 4e-05
Epoch: 9000, loss: 3.44630861945916e-05
Epoch: 10000, loss: 2.077104181807954e-05
Predictions: tensor([[2.1226e-05],
        [9.9998e-01],
        [9.9998e-01],
        [2.1185e-05]], grad_fn=<SigmoidBackward0>)
Predicted Classes: tensor([[0],
        [1],
        [1],
        [0]])
model.0.weight tensor([[-6.1799,  6.3918],
        [-7.2902,  7.3291]])
model.0.bias tensor([ 3.1900, -4.3039])
model.2.weight tensor([[-5.7749,  6.0148],
        [ 5.7013, -6.6082]])
model.2.bias tensor([ 2.8003, -2.6538])
model.4.weight tensor([[ 6.3596, -6.5419],
        [-6.6853,  7.0563]])
model.4.bias tensor([ 0.1071, -0.1949])
model.6.weight tensor([[ 11.2224, -10.4749]])
model.6.bias tensor([-0.3518])
```

-> 위 조건으로 train 결과 손실 값이 점점 감소하고 최종 손실은
2.077104181807954e-05로 매우 낮은 값을 나타고 있음.
예측된 클래스는 예측된 확률에 기반하여 0.5보다 크면 1로, 그렇지 않으면 0으로
표시되도록 설정되어 있는데, 이 예측된 클래스가 [0, 1, 1, 0]으로
실제클래스 [0, 1, 1, 0] 와 정확히 일치하므로 학습이 잘 되었다고 볼 수 있음.

학습 결과
- 첫 번째 은닉층(model.0) 가중치 [[-6.1799, 6.3918], [-7.2902, 7.3291]]
- 첫 번째 은닉층(model.0) 편향 [3.1900, -4.3039]
- 두 번째 은닉층(model.2) 가중치 [[-5.7749, 6.0148], [5.7013, -6.6082]]
- 두 번째 은닉층(model.2) 편향 [2.8003, -2.6538]
- 세 번째 은닉층(model.4) 가중치 [[6.3596, -6.5419],[-6.6853, 7.0563]]
- 세 번째 은닉층(model.4) 편향 [0.1071, -0.1949]
- 출력층 (model.6) 가중치 [[11.2224, -10.4749]]
- 출력층 (model.6) 편향 [-0.3518]

<최종 출력>

```
##최종출력

import torch


# 주어진 파라미터
w0 = torch.tensor([[-6.1799,  6.3918], [-7.2902,  7.3291]])
b0 = torch.tensor([ 3.1900, -4.3039])
w2 = torch.tensor([[-5.7749,  6.0148], [ 5.7013, -6.6082]])
b2 = torch.tensor([ 2.8003, -2.6538])
w4 = torch.tensor([[ 6.3596, -6.5419],[-6.6853,  7.0563]])
b4 = torch.tensor([ 0.1071, -0.1949])
w6 = torch.tensor([[ 11.2224, -10.4749]])
b6 =  torch.tensor([-0.3518])

# 입력 데이터
X = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float32)

sigmoid = nn.Sigmoid()

# 첫 번째 은닉층의 출력 계산
layer1_output = sigmoid(X @ w0.T + b0)

# 두 번째 은닉층의 출력 계산
layer2_output = sigmoid(layer1_output @ w2.T + b2)

# 세 번째 은닉층의 출력 계산
layer3_output = sigmoid(layer2_output @ w4.T + b4)

# 최종 출력 계산
final_output = sigmoid(layer3_output @ w6.T + b6)
print(final_output)
```

```
tensor([[2.1226e-05],
        [9.9998e-01],
        [9.9998e-01],
        [2.1185e-05]])
```

위에 출력된 파라미터를 대입하여 각 은닉층의 출력과 최종출력을 계산

<3d 출력>

```
import numpy as np
import torch
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# XOR 게이트를 위한 입력 데이터 정의
x_xor = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=torch.float)

class XOR(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = torch.nn.Linear(2, 2)
        self.sigmoid = torch.nn.Sigmoid()
        self.layer2 = torch.nn.Linear(2, 2)
        self.sigmoid2 = torch.nn.Sigmoid()
        self.layer3 = torch.nn.Linear(2, 2)
        self.sigmoid3 = torch.nn.Sigmoid()
        self.layer4 = torch.nn.Linear(2, 1)
        self.sigmoid4 = torch.nn.Sigmoid()

    def forward(self, x):
        out = self.layer1(x)
        out = self.sigmoid(out)
        out = self.layer2(out)
        out = self.sigmoid2(out)
        out = self.layer3(out)
        out = self.sigmoid3(out)
        out = self.layer4(out)
        out = self.sigmoid4(out)
        return out

xor_model = XOR()
```

__init__ 메서드 : 다층 퍼셉트론 정의
-> 3개의 은닉층, 1개의 출력층
각 층은 선형 변환과 활성화 함수인 시그모이드 함수로 구성

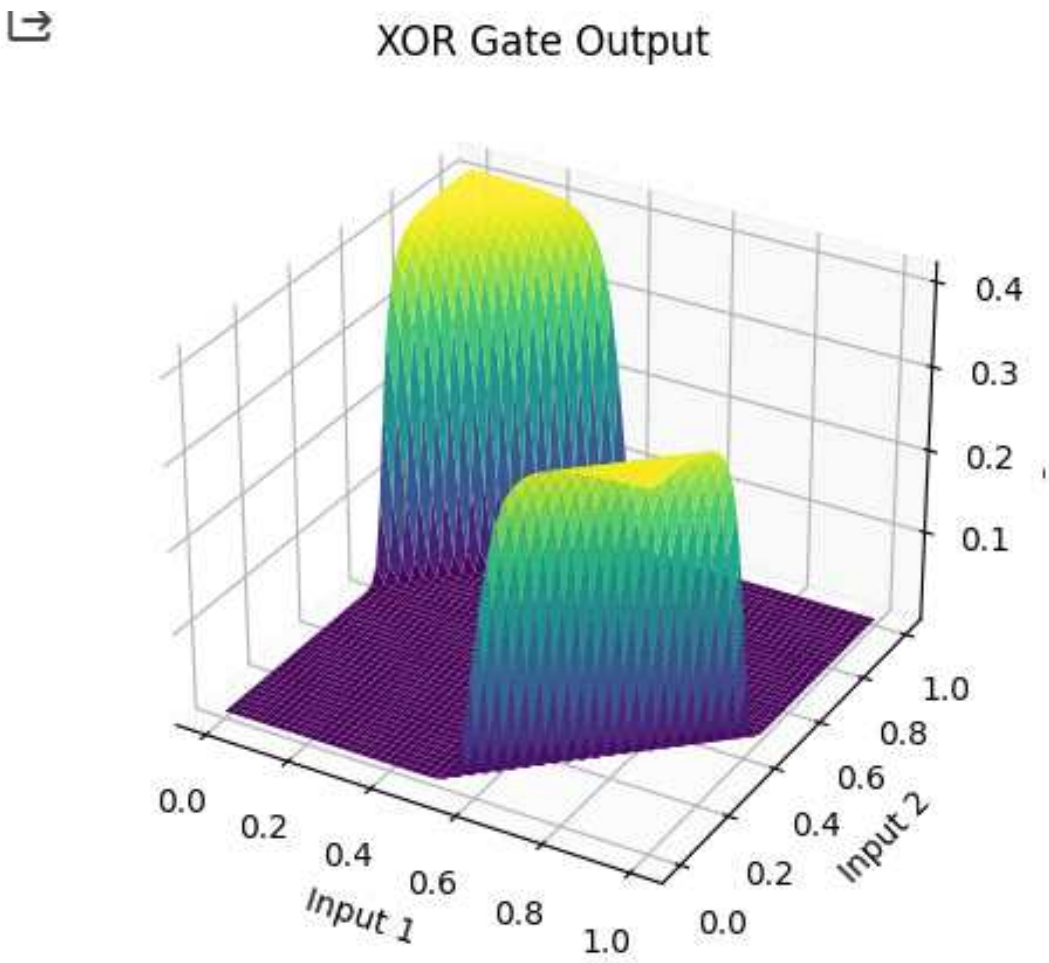
forward 메서드 : 입력 데이터를 각 층에 순전파 시킴

```
# 학습된 파라미터 값 대입
xor_model.layer1.weight.data = torch.tensor([[-6.1799, 6.3918],
                                             [-7.2902, 7.3291]])
xor_model.layer1.bias.data = torch.tensor([3.1900, -4.3039])
xor_model.layer2.weight.data = torch.tensor([[-5.7749, 6.0148],
                                             [5.7013, -6.6082]])
xor_model.layer2.bias.data = torch.tensor([2.8003, -2.6538])
xor_model.layer3.weight.data = torch.tensor([[-6.3596,   -6.5419],
                                             [-6.6853, 7.0563]])
xor_model.layer3.bias.data = torch.tensor([0.1071, -0.1949])
xor_model.layer4.weight.data = torch.tensor([[11.2224,   -10.4749]])
xor_model.layer4.bias.data = torch.tensor([-0.3518])
```

이후 위에 결과로 나온 파라미터를 대입

```
# 3차원 그래프 생성을 위한 입력 데이터 정의
x_values_xor = np.linspace(0, 1, 50)
y_values_xor = np.linspace(0, 1, 50)
xx_xor, yy_xor = np.meshgrid(x_values_xor, y_values_xor)
xy_xor = np.column_stack([xx_xor.ravel(), yy_xor.ravel()])
xy_tensor_xor = torch.tensor(xy_xor, dtype=torch.float)

# 각 점에서의 예측값 계산
with torch.no_grad():
    predictions_xor = xor_model(xy_tensor_xor).numpy().reshape(xx_xor.shape)

# 면 그래프로 출력 시각화
fig = plt.figure()
ax_xor = fig.add_subplot(111, projection='3d')
ax_xor.plot_surface(xx_xor, yy_xor, predictions_xor, cmap='viridis')

ax_xor.set_xlabel('Input 1')
ax_xor.set_ylabel('Input 2')
ax_xor.set_zlabel('Output')
ax_xor.set_title('XOR Gate Output')
plt.show()
```

출력결과 :



<팀 내 토론 과정에서 발견한 insight 등>

- 학습률(lr)은 가중치 및 편향 업데이트의 속도를 결정한다.
0.001~0.1 사이로 숫자를 넣어보며 최적의 값을 찾는다.
학습률이 올라가면 속도가 빨라지지만 최적화 과정이 불안정해질 수 있기에 무조건 높은 학습률이 좋은 것은 아니다.

- 에포크, 에폭(epochs)은 데이터셋 전체를 한 번 학습하는 단위로 적은 수의 에폭으로 시작하여 학습이 제대로 될 때까지 에폭수를 늘려가는게 좋은 방법이다. 에폭이 너무 높으면 과적합이 발생할 수 있다.

- 히든레이어 또한 많다고 좋은 것이 아니다. 물론 히든레이어가 많으면 더 복잡한

패턴을 학습할 수 있으나 과적합의 위험성이 있고 더 많은 계산 비용과 학습 시간이 들어간다.