Hidden 4 Layer

이 코드는 PyTorch 를 사용하여 XOR 문제를 해결하기 위한 신경망을 정의하고 있으며, 다층 퍼셉트론(Multi-Layer Perceptron, MLP)을 사용하여 XOR 문제를 해결하고 있는데, 코드는 다음과 같은 구조를 가지고 있다:

```
#  XOR  문제  해결  class
XOR(nn.Module):       def
__init__(self):
super().__init__()
        self.weight = torch.tensor([0.5, 0.5],
dtype=torch.float)
        self.bias = torch.tensor([-0.25], dtype=torch.float)
self.model = nn.Sequential( # nn.Module 클래스를 상속하여
정의
            nn.Linear(2, 2),  # 첫 번째 히든 레이어: 2 개의 입력을
받아 4 개의 출력 유닛으로 연결
            nn.Sigmoid(),     # 첫 번째 히든 레이어의 활성화 함수
            nn.Linear(2, 2),  # 두 번째 히든 레이어: 4 개의 입력을
받아 4 개의 출력 유닛으로 연결
            nn.Sigmoid(),     # 두 번째 히든 레이어의 활성화 함수
            nn.Linear(2, 2),  # 세 번째 히든 레이어: 4 개의 입력을
받아 4 개의 출력 유닛으로 연결
            nn.Sigmoid(),     # 세 번째 히든 레이어의 활성화 함수
            nn.Linear(2, 2),  # 네 번째 히든 레이어: 4 개의 입력을
받아 4 개의 출력 유닛으로 연결
            nn.Sigmoid(),     # 네 번째 히든 레이어의 활성화 함수
nn.Linear(2, 1),  # 출력 레이어: 4 개의 입력을 받아 1 개의
출력 유닛으로 연결
            nn.Sigmoid()      # 출력 레이어의 활성화 함수
        )
    def forward(self, x):
return self.model(x)
```

다음과 같이 4 개의 은닉층을 거치면서 최종적으로 하나의 출력을 내놓고자 구현 했다.

다음은 모델객체를 생성한뒤 주어진 데이터를 바탕으로 모델을 학습시킨다.

```
def train(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) # 초기화
함                    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()}')
```

reset 이 True 로 설정되어 있으면, 각 레이어의 파라미터를 초기화한다. 이 때, reset 이 'xavier'로 설정되어 있다면 Xavier 초기화를 수행하고 주어진 옵티마이저 (grad 초기화하는 부분)와 손실 함수를 설정한다. 이 함수를 진행하는 이유는 모 델을 최적화할 적절한 파라미터를 찾기 위해서인데,

```
# parameter initialization - nn.init
train(X, y, epochs=30000, lr=0.01, optimizer='Adam',
model=model, every_print=1000, reset='xavier') # 더 많은 에포크를
시도
```

```
model(X) ,torch.where(model(X) > 0.5, 1, 0)
```

에포크를 3 만번, 학습률을 0.01 로 설정하고 위에서 학습한 모델을 사용하여 직접 입력데이터에 대한 예측을 수행한다. (주석제외)세번째 줄에서는 조건을 사용하여 모델의 출력값이 0.5 보다 크면 1 로, 그렇지 않으면 0 으로 바꾸어서 이진 분류 결 과를 얻는다. 이렇게 하면 기본적으로 제공한 텐서데이터에서 xor 게이트를 통해 변환된 값을 스스로 예측할 수 있는 결과를 제공할 것이다.

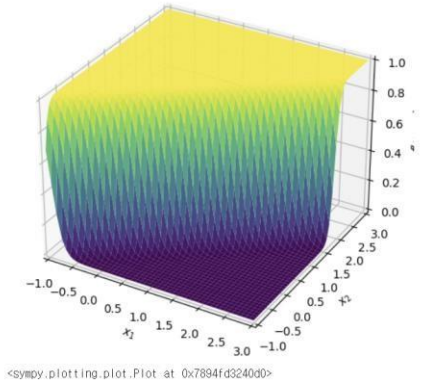
```
from sympy import symbols, exp from
sympy.plotting import plot3d

# # 변수 정의
# x, x1, x2 = symbols('x x1 x2')
# w1, w2, w3, w4, w5, w6 = symbols('w1 w2 w3 w4 w5 w6')
# b1, b2, b3 = symbols('b1 b2 b3')

#주어진 파라미터
w0 = torch.tensor([[-10.3267,  14.3530], [ -3.8846,   6.2506]])
b0 = torch.tensor([4.8114, -3.5770]) w2 = torch.tensor([[-4.5832,
5.1053], [6.1589, -5.0255]]) b2 = torch.tensor([1.5159, -3.0480])
w4 = torch.tensor([[5.6698, -6.0909],[-6.0369,  7.5744]]) b4 =
torch.tensor([0.4247, -1.0466]) w6 = torch.tensor([[7.1937, -
7.3540],[-7.5309,  7.5448]]) b6 =  torch.tensor([0.0227, 0.0733])
w8 = torch.tensor([[ 17.7575, -19.4533]]) b8 =  torch.tensor([-
1.0813])

# 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)
 final_output = sigmoid(layer3_output @ w8.T + b8) print(final_output)
```

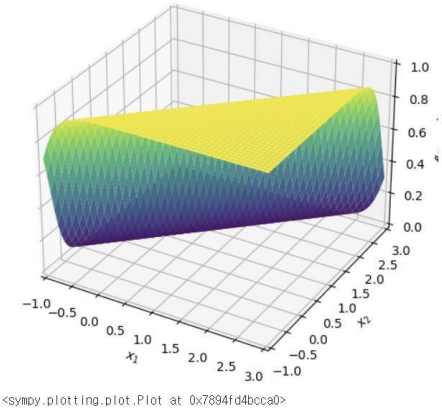
이제 주어진 파라미터와 입력데이터를 사용하여 다층 퍼셉트론의 각 은닉층과 출력층의 출력을 계산한다. 시그모이드 함수를 통해 이것을 계산하고, 각 층의 출력이 반환되면서 입력이 어떻게 변하는지 확인할 수 있는 동시에 신경망이
어떻게 입력을 처리하고 있는지도 이해할 수 있다. import torch

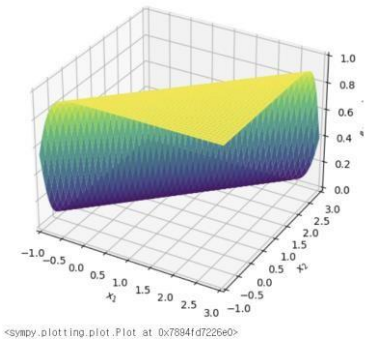
```
from sympy import symbols, exp from
sympy.plotting import plot3d

# 변수 정의
x, x1, x2 = symbols('x x1 x2') # 입력변수 w1, w2, w3, w4, w5, w6,
w7, w8 = symbols('w1 w2 w3 w4 w5 w6 w7 w8') b1, b2, b3 =
symbols('b1 b2 b3') #편향변수
sigmoid = 1 / (1+exp(-x))
```

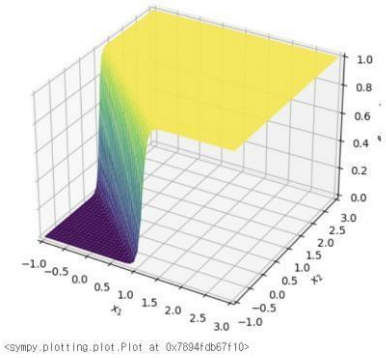
입력변수, 가중치변수, 편향변수를 심볼릭 변수로 정의한채 다음과 같이 각층의 히든층에 출력 계산식을 정의하여 h1~h3 를 다음과 같이 출력했다.

순서대로 h1, h2, h3,h4









이제 각 층의 형태를 고려하면서 최종출력단계를 거칠것이다.

```
# 첫 번째 히든 레이어의 가중치와 편향
w0 = torch.tensor([[-10.3267,  14.3530],
[ -3.8846,   6.2506]]) b0 =
torch.tensor([ 4.8114, -3.5770])

# 두 번째 히든 레이어의 가중치와 편향
w2 = torch.tensor([[-4.5832,  5.1053],
[ 6.1589, -5.0255]]) b2 =
torch.tensor([ 1.5159, -3.0480])

# 세 번째 히든 레이어의 가중치와 편향
w4 = torch.tensor([[ 5.6698, -6.0909],
[-6.0369,         7.5744]])    b4    =
torch.tensor([ 0.4247, -1.0466])

# 네 번째 히든 레이어의 가중치와 편향
w6 = torch.tensor([[ 7.1937, -7.3540],
[-7.5309,         7.5448]])    b6    =
torch.tensor([0.0227, 0.0733])

# 출력층의 가중치와 편향
w8 = torch.tensor([[ 17.7575, -19.4533]])
b8 = torch.tensor([-1.0813])
```

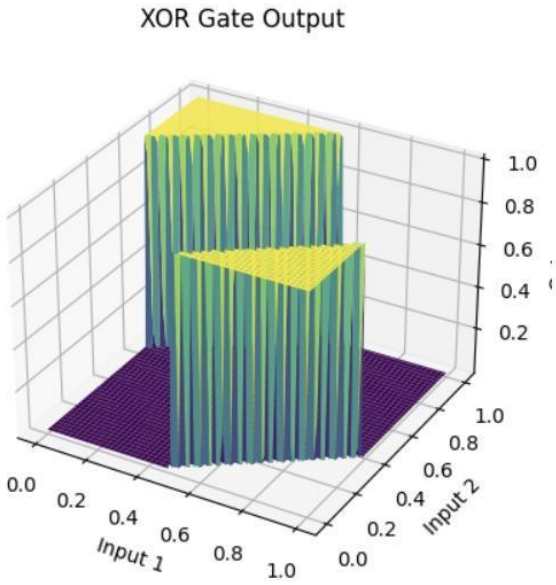
이제 학습한 모델을 기반으로 직접학습한 데이터를 가져와 3d 그래프를 그려보고 자
한다.

```
class XOR(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = torch.nn.Linear(2, 2)
        self.sigmoid1 = torch.nn.Sigmoid()  # 첫 번째 은닉층의 활성화 함수
        self.layer2 = torch.nn.Linear(2, 2)
        self.sigmoid2 = torch.nn.Sigmoid()  # 두 번째 은닉층의 활성화 함수
        self.layer3 = torch.nn.Linear(2, 2)
        self.sigmoid3 = torch.nn.Sigmoid()  # 세 번째 은닉층의 활성화 함수
        self.layer4 = torch.nn.Linear(2, 2)
        self.sigmoid4 = torch.nn.Sigmoid()  # 네 번째 은닉층의 활성화 함수
        self.output_layer = torch.nn.Linear(2, 1)  # 출력층
    def forward(self, x):
        out = self.layer1(x)
        out = self.sigmoid1(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)

        out = self.output_layer(out)  # 출력층
        return out
```

헷갈리는 개념 제시:

__init__ 메서드: 네 개의 은닉층과 출력층으로 구성된 신경망을 정의하고 있다. 각 은닉층은 선형 레이어와 활성화 함수로 구성되어 있다 forward 메서드: 입력 데이터 x 를 각 레이어를 통과시켜 순전파를 수행할 것이다. 각 은닉층과 활성화 함수를 번갈아가면서 적용한 후, 최종 출력층에서 최종 출력 을 계산한다.

가장 마지막 단계에서는 모델을 사용하여 입력데이터 공간에서 출력을 시각화한다. 구조는 2 개의 입력을 받아 4 개의 은닉층을 거쳐 1 개의 출력을 내놓는 구조로, 아까 정의한 그니까 미리 학습된 파라미터 값을 해당모델에 대입해서 모델을 초 기화시킨다. 이후, 입력공간을 시각화하기위해 100*100 형태의 그리드를 생성하 고, 예측값을 시각화한다. 이때 matplotlib 과 mpl_toolkits 모듈을 사용하였다. 결과는 다음과 같다.



이전 레이어들과 형태가 다른 이유는 학습한 파라미터값의 크기, 레이어의 개수 와 구성을 변경한 것에 있는데, 단순히 그리드를 조작하는 것만으로는 유연한 형 태의 모양을 얻을 수 없다.