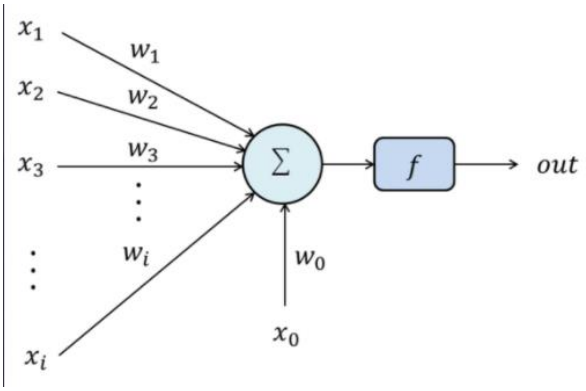
XOR을 위한 다층 퍼셉트론

1. 정의 및 1번 질문관련내용

이 과제를 수행하기위해서는 우선적으로, 다음과 같은 정의에 대해서 이해해야할 것이다.

- XOR 게이트(Exclusive-OR) : 배타적 논리합. 입력이 같을 때는 0을, 입력이 다르면 1을 축력한다.
- 퍼셉트론 : 다수의 신호를 입력으로 받아, 각각 가중치를 부여하며, 하나의 신호를 출력한다. (가중치가 클수록 해당 신호가 그만큼 더 중요함을 뜻함.)

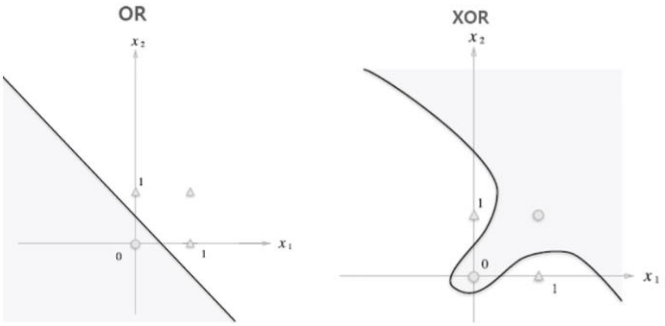


한편, 퍼셉트론은 다음과 같이 나뉜다.

- 단층 퍼셉트론 : 중간층이 하나의 노드로 구성되어 중간층과 출력층의 구분이 없는 구조
- 다층 퍼셉트론 : 중간층을 구성하는 노드가 여러 개이고, 이러한 중간층이 다수로 구성되어 있는 구조, 여러 개의 활성함수, 이에 따른 가중치도 여러 개

이에 따라 1) 단층 퍼셉트론에서 XOR게이트 구현이 불가능한 이유를 알 수 있다.

➔ 선형적인 단층 퍼셉트론으로는, 다음처럼 AND, NAND, OR 게이트는 구현이 가능하지만, XOR 게이트는 구현이 불가능하다. 따라서, 입력층과 출력층 사이에 하나 이상의 중간층을 두어 **비선형적**으로 분리되는 데이터에 대해서 학습 가능하도록 고안된 **다층 퍼셉트론**을 통해서, XOR 게이트는 구현이 가능하다.



2. 코드설명 및 2번,3번 질문내용

```
import matplotlib.pyplot as plt
import numpy as np

# 입력 데이터
#[0, 0], [1, 1], [0, 1], [1, 0]
X = np.array([0, 0, 1, 1, 0, 1, 0, 1]).reshape(2,4)

#학습 타겟 : XOR일 때 1
Y = np.array([0, 1, 1, 0]).reshape(1,4)
```

(6~7) 그래프표현을 위한 plt와 행렬 연산을 위한 numpy를 선언해준 모습입니다.
(10~14) 인풋 레이어와 아웃풋 레이어 설정.

```
def init_random_parameters (num_hidden = 2, deviation = 1):

    W1 = np.random.rand(2,num_hidden)*deviation
    B1 = np.random.random((num_hidden,1))*deviation
    W2 = np.random.rand(num_hidden,1)*deviation
    B2 = np.random.random((1,1))*deviation
    return W1, B1, W2, B2
```

get_gradients함수를 수행하기 위해서, 파라미터 4가지를 랜덤하게 생성하는 함수이다.
여기서 **3) 학습 파라미터를 어떤식으로 초기화하는게 좋은지 알 수 있다.** XOR게이트에서 optimize할 때, 파라미터를 0으로 초기화한다면 파라미터의 변화가 없어서 학습이 진행이 되지 않는다. 그래서 처음 그래프 결과가 아무런 변화가 없는것으로 나온것이다. 따라서, random하게 초기화하는 것이 좋다.

```
def affine (W, X, B):
    return np.dot(W.T, X) + B

def sigmoid (o):
    return 1./(1+np.exp(-1*o))
```

$$H = \begin{bmatrix} h_{00} & h_{01} & h_{02} & h_{03} \\ h_{10} & h_{11} & h_{12} & h_{13} \end{bmatrix}$$
$$= \sigma(Z^{[1]})$$
$$= \sigma(W^{[1]^T} X + B^{[1]})$$

이와 같은 hidden layer를 위해 생성된 함수이다.

```python
def loss_eval(_params):
    W1, B1, W2, B2 = _params

    # Forward: input Layer
    Z1 = affine(W1, X, B1)
    H = sigmoid(Z1)

    # Forward: Hidden Layer
    Z2 = affine(W2, H, B2)
    Y_hat = sigmoid(Z2)

    loss = 1. / X.shape[1] * np.sum(-1 * (Y * np.log(Y_hat) + (1 - Y) * np.log(1 - Y_hat)))
    return Z1, H, Z2, Y_hat, loss
```

$$\mathcal{L} = -\frac{1}{4}\sum_{i}\{y \log \hat{y} + (1-y)\log(1-\hat{y})\}$$

위와 같은 식에 의해서, loss 계산 함수를 정의하면 loss_eval과 같다.

```python
def get_gradients(_params):
    W1, B1, W2, B2 = _params
    m = X.shape[1]

    Z1, H, Z2, Y_hat, loss = loss_eval([W1, B1, W2, B2])

    # BackPropagate: Hidden Layer
    dW2 = np.dot(H, (Y_hat - Y).T)
    dB2 = 1. / 4. * np.sum(Y_hat - Y, axis=1, keepdims=True)
    dH = np.dot(W2, Y_hat - Y)

    # BackPropagate: Input Layer
    dZ1 = dH * H * (1 - H)
    dW1 = np.dot(X, dZ1.T)
    dB1 = 1. / 4. * np.sum(dZ1, axis=1, keepdims=True)

    return [dW1, dB1, dW2, dB2], loss
```

파라미터별로 편미분을 사용하여, 파라미터 update를 해주는 gradient이다.

여기서, **2) 다층 퍼셉트론을 학습할 때, 학습률(learning rate)이 모델의 학습에 미치는 영향**을 알 수 있다. 다층 퍼셉트론을 학습할 때, gradient descent를 사용하는데, 이 방법의 원리는 산 정상에서 골짜기로 가장 빠르게 내려오는 방법인 가장 경사가 가파른 길을 골라 내려오는 것과 같다. 따라서, 학습률이 너무 작으면 반복을 너무 많이 진행해야 하므로 시간이 오래 걸리며, 전역 최솟값에 수렴하기 전에 지역 최솟값에서 모델이 학습을 중단할 수도 있다.

반면에 학습률이 너무 크면 골짜기를 가로질러 골짜기 아래로 내려가는 것이 아니라 반대로 올라갈 수 도 있으며 알고리즘이 더 큰 값으로 발산하게 되어 적절한 해법을 찾을 수 없게 된다. 이를 overshooting문제라 하며, 따라서 적절한 해법을 찾기 위해서는 여러 가지 학습률을 적용해보고 비교하는 과정이 필요하다.

```python
#Gradient를 이용하여 학습 파라미터를 업데이트하기 위한 소스 코드
def optimize(_params, learning_rate=0.1, iteration=1000, sample_size=0):
    params = np.copy(_params)

    loss_trace = []

    for epoch in range(iteration):

        dparams, loss = get_gradients(params)


        for param, dparam in zip(params, dparams):
            param += - learning_rate * dparam

        if (epoch % 100 == 0):
            loss_trace.append(loss)

    _, _, _, Y_hat_predict, _ = loss_eval(params)



    return params, loss_trace, Y_hat_predict
```

learning_rate가 0.1일 때 1000번의 iteration동안 학습시키는 함수이다.

```python
params = init_random_parameters (2, 0.1)


new_params, loss_trace, Y_hat_predict = optimize(params, 0.1, 100000)

print(Y_hat_predict)
print(new_params)

plt.plot(loss_trace)
plt.ylabel('loss')
plt.xlabel('iterations (per hundreds)')
plt.show()
```

메인 코드이다.

우선, init_random_paremeters함수를 사용하여 파라미터를 랜덤하게 생성한다.

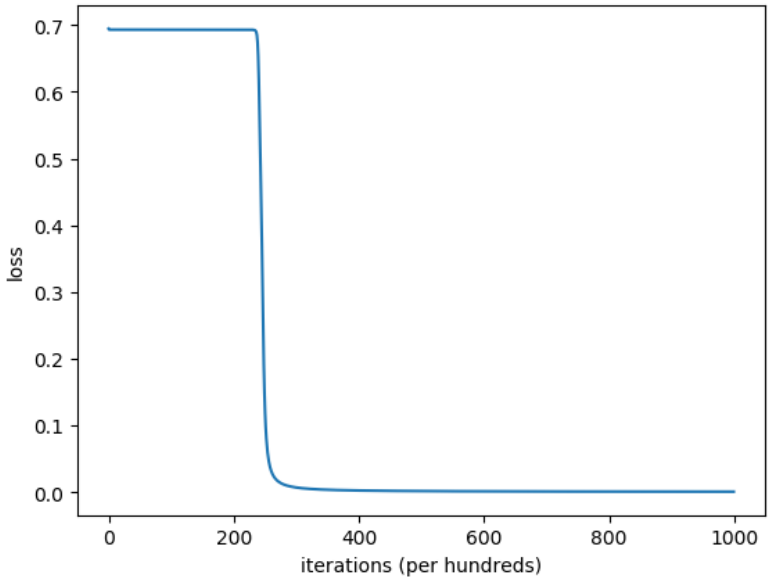
그 후, optimize함수를 사용하여 파라미터를 지속적으로 업데이트해주면서, 학습을 진행한다.

그 후, 새로운 파라미터와 결과 등을 표시한다.

3. 결과 출력

```
[[7.86935227e-04 9.99716114e-01 9.99716110e-01 3.03205414e-04]]
[array([[8.68385834, 4.95882037],
       [8.68411691, 4.95884257]])
 array([[-4.12494146],
       [-7.47565924]])
 array([[ 17.20734841],
       [-19.45578917]]) array([[-7.40928038]])]
```

학습 타겟인 [0, 1, 1, 0]에 맞게 Y_hat_predict가 나오는 것을 볼 수 있으며, 갱신된 파라미터를 확인할 수 있다.



iteration이 진행되면서, loss가 낮아지는 이상적인 그래프를 확인할 수 있다.