XOR을 위한 다층 퍼셉트론

전기컴퓨터공학부 정보컴퓨터공학전공 201524404 강민진

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

: matplotlib와 numpy를 사용하기 위한 코드이다.

```
# 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)
```

: 행렬 형태로 X, Y 를 나타내기 위한 코드이다.

```
def init_parameters (num_hidden=2):
    W1 = np.zeros((2,num_hidden))
    B1 = np.zeros((num_hidden,1))
    W2 =  np.zeros((num_hidden,1))
    B2 = np.zeros((1,1))
    return W1, B1, W2, B2
```

: 학습에 사용하는 parameter 생성 W1, B1, W2, B2는 각각 Weight1, Weight2, Bias1, Bias2.

layer가 2개인 다층 퍼셉트론 구현 위한 것이다. 모든 행렬을 0으로 초기화 시키기 위한 코드이다.

```
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
```

: 모든 행렬을 랜덤하게 초기화시키기 위한 코드이다.(수정한 코드)

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

: W, X, B : 각각 가중치(Weight), 입력값(Input), Bias

Wx + b를 계산하여 리턴해주는 함수이다.

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

: sigmoid를 적용하기 위한 함수

```
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
```

: loss를 parameter로 affine해서 출력값을 구하고 sigmoid 적용. Y_hat이라는 예측값 생성, loss를 행렬로 계산한다.

```
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
    # backpropagation 으로 gradient
    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
    # 첫 가중치와 첫 가중치에 대한 gradient 를 얻는다.
    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
```

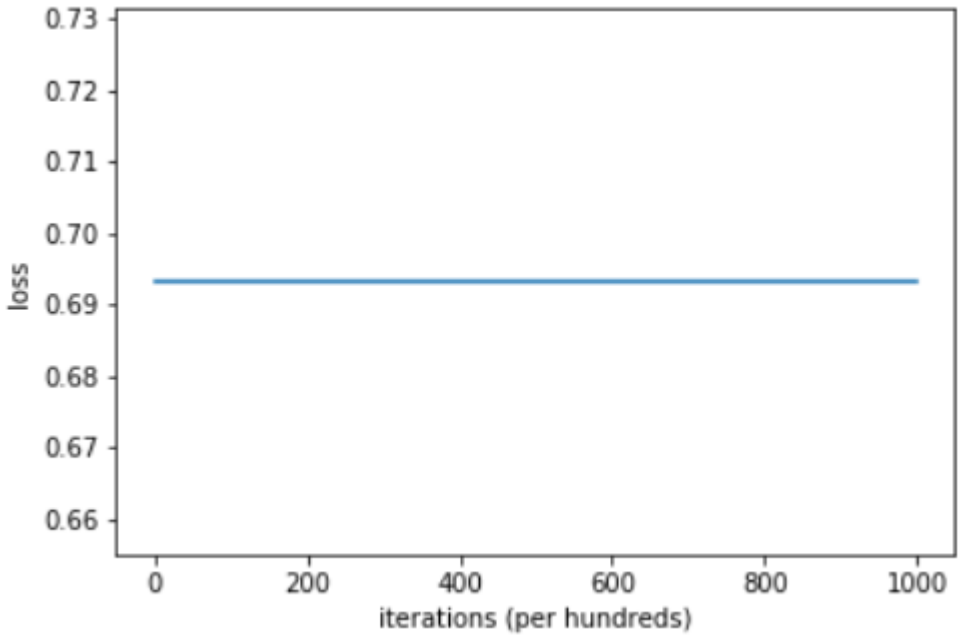
forward propagation으로 결과 값을 구할 수 있다. 구한 결과값을 이용하여 backward propagation 을 수행하여 편미분을 통해 영향을 끼치는 정도를 이용하여 학습을 진행할 수 있다.

```python
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 # learning rate를 곱해서 parameter를 얻게 됨.

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

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

    return params, loss_trace, Y_hat_predict
```

Learning rate를 곱해서 기존의 parameter로부터 update된 parameter를 얻게 된다.

optimize 함수는 learning rate를 gradient에 적용하여 parameter를 지속적으로 업데이트하기 위한 소스코드이다.

위의 함수는 learning rate수치를 0.1로 해서 1000번의 iteration 동안 학습시키는 코드이다.



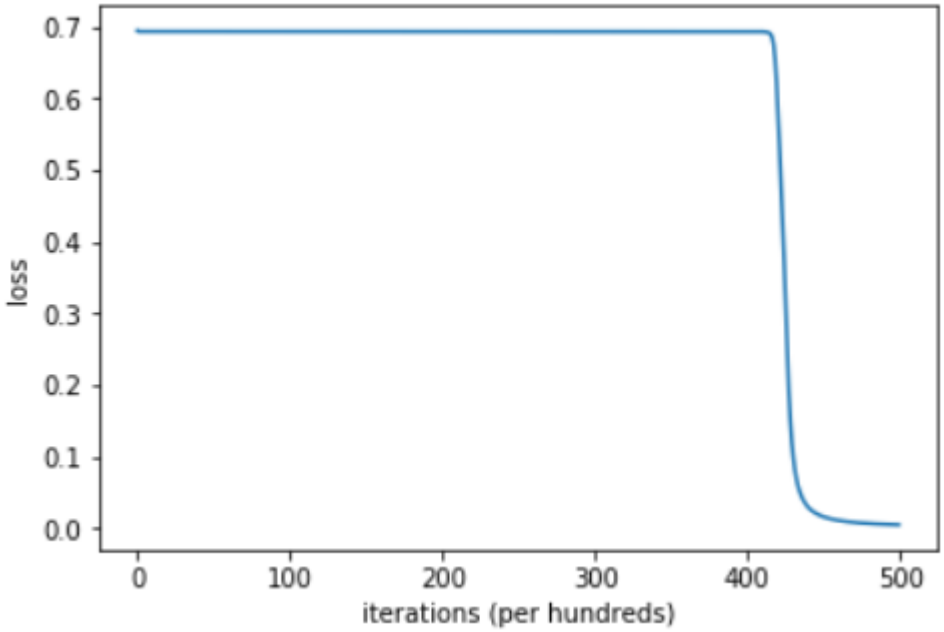
init_parameter함수를 통해서 초기화시켜주게 되면 AND와 같은 연산시에는 제대로 동작하지만, XOR연산에서는 제대로 동작하지 않는다. 그 이유는 AND는 linear하게 parameter input, output이 동작하지만 XOR에서는 nonlinear하게 동작하기 때문이다. 그래서 랜덤하게 학습 parameter를 초기화시켜주는 init_random_parameters 함수를 적용시켜주었다.

```python
params = init_random_parameters(2, 0.1) # Sol
new_params, loss_trace, Y_hat_predict = optimize(params, 0.1, 100000)
```

```
print(Y_hat_predict)
```

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

loss function을 gradient descent를 적용하여 추적해나감. 최적 값을 찾고 그래프를 출력한다.(iterations, loss 값을 통하여 update에 따른 정확성을 시각적으로 보여준다.)




```
def tanh(x):
    ex = np.exp(x)
    enx = np.exp(-x)
    return (ex-enx)/(ex+enx)
```

: tanh를 적용시키기 위한 식을 함수로 구현하였다.

```
def loss_eval_tanh (_params):

    W1, B1, W2, B2 = _params

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

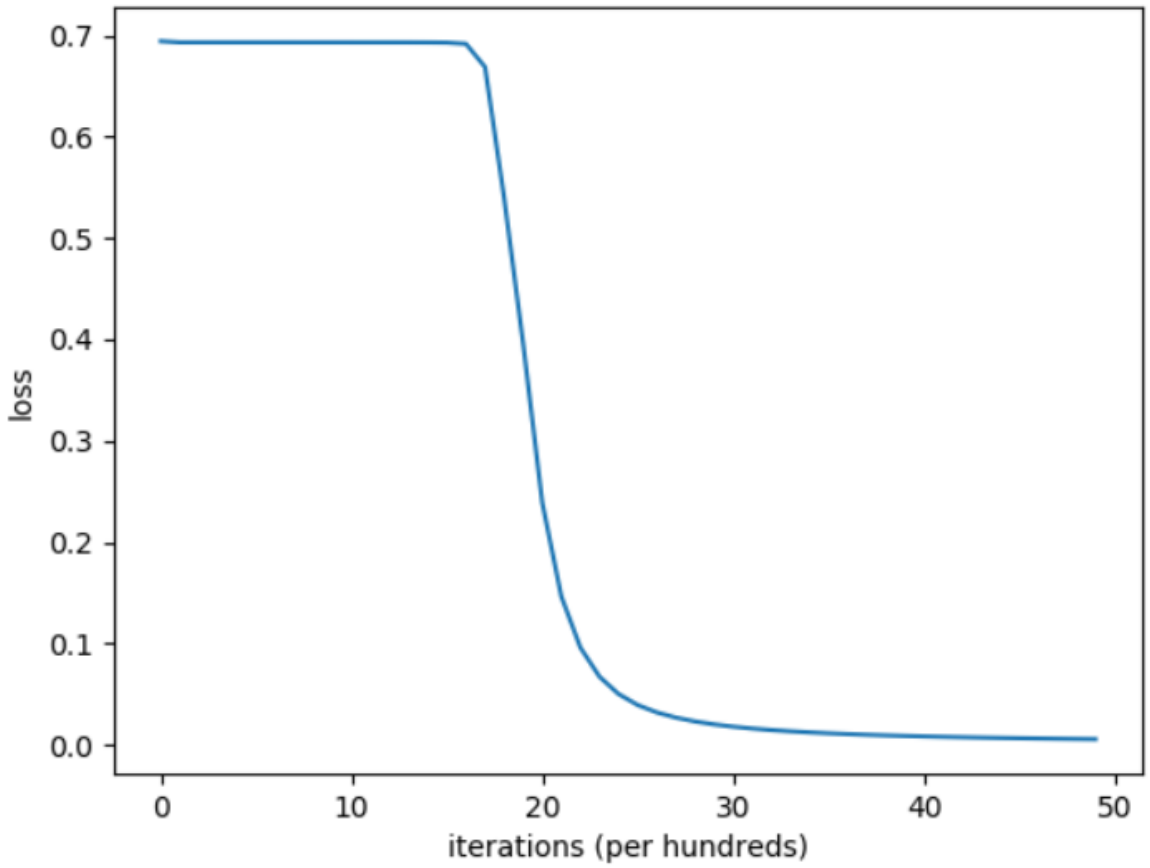
: tanh에서 forward propagation 해서 손실 값을 계산하기 위한 함수이다.

```
def get_gradients_tanh (_params):

    W1, B1, W2, B2 = _params

    Z1, H, Z2, Y_hat, loss = loss_eval_tanh([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 * (1 - (H * H)) # <- Changed!
    dW1 = np.dot(X, dZ1.T)
    dB1 = 1./4. * np.sum(dZ1, axis=1, keepdims=True)

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

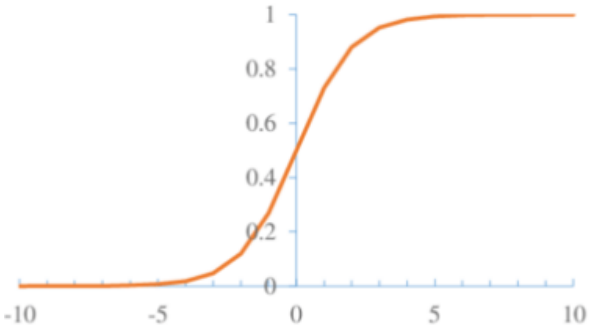
: tanh에서 backpropagation(backward propagation 을 적용해서 gradient 를 계산하기 위한 함수이다.

```python
def optimize_tanh (_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_tanh(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_tanh(params)

    return params,loss_trace, Y_hat_predict
```

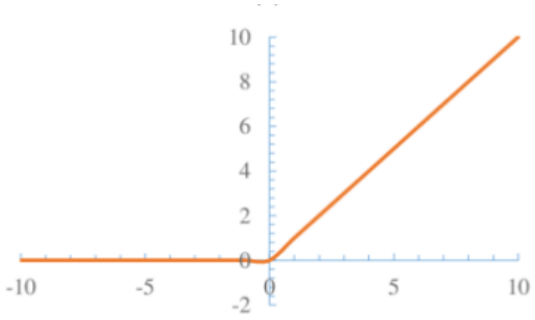
: learning rate = 0.1, iteration = 1000으로 해서 학습을 진행하였다.



nonlinear function의 종류들에 sigmoid 와 tanh 가 있는데, 일반적으로 sigmoid 보다 tanh를 사용하는 것이 더 좋은 성능을 낸다고 알려져 있다.



[sigmoid]



[tanh]

sigmoid, tanh외에도 ReLu, Leaky ReLu function 등이 존재한다.

1. 단층 퍼셉트론에서는 XOR게이트를 구현할 수 없는 이유?

: x1, x2 이렇게 입력이 두개이고, y가 출력이라고 가정하면

(1) x1 = 0 인 경우 -> x2 = 1이면 y = 1이 되어야 하고,

-> x2 = 0이면 y = 0이 되어야 한다. 즉, x2가 증가하면 y가 증가한다.

(2) x1 = 1 인 경우 -> x2 = 1이면 y = 0이 되어야 하고,

-> x2 = 0이면 y = 1이 되어야 한다. 즉, x2가 증가하면 y가 감소한다.

x2의 값에 따라 x1고 y가 XOR게이트를 만족하도록 구성되어야 하는데, x1, y에 영향을 미치는 파라미터는 w1(weight1)뿐이며, w1만 조절함으로써 y의 증가, 감소를 유연하게 변경불가능하다.

-> 단층 퍼셉트론의 linear한 처리로는 XOR게이트를 구현할 수 없기 때문에 다층 퍼셉트론을 이용해야 한다.

2. 다층 퍼셉트론 학습 시 learning rate(학습률)이 모델의 학습에 미치는 영향은?

: NN structure 가 커져 parameter가 늘어날 수록 learning rate를 낮게 잡는 편이 낫다. 그러나, learning rate가 낮아지면 학습속도가 느려질 뿐 아니라, 지역적 최소점에 갇혀 빠져나오지 못하는 결과가 초래될 수 있다. 반면, learning rate가 크면 목표 근처에서 정확하게 바닥(loss function의 최솟값)을 찾는 능력이 무뎌진다. 이러한 점을 극복한 learning parameter의 최적화 방법에는 stochastic gradient와 Adam 기법이 있다.

3. 학습 파라미터를 어떤 식으로 초기화하는 것이 좋은가?

: 학습 파라미터를 초기화시키는데 절대적인 정답은 없다. 너무 작지도, 크지도 않은 딱 맞는 learning rate를 찾는 것은 어렵다고 생각한다. learning rate를 일정하게 감소해나가는 방식으로 학습 parameter를 초기화하는 방식 등이 있다. AND 등과 같은 linear한 것이 사용가능한 것들과 달리 XOR의 경우에는 linear한 방법이 적용되지 않으므로 random 하게 학습 parameter를 초기화시켜주는 방식을 사용하였다.