# 수치최적화 기말과제 보고서 2019270109 권종서

## 0. 서론

초기 가중치, 편향 값은 테스트를 위해서 아래의 가중치, 편향 값을 이용하였다. 원래 정규분포 난수 값을 이용해서 초깃값을 설정하므로 numpy의 random.normal을 이용해서 생성해줘야 하지만 엑셀 파일에 설정된 값들 또한 같은 방식으로 생성한 값이므로 정확한 값 비교를 위해 아래의 값들을 이용하였다.

**숫자 0과 1의 식별**

| 학습률 η |
|---|
| 0.2 |

| 번호 | | | 1 | | |
|---|---|---|---|---|---|
| 입력층 | 이미지 패턴 | | 1 | 1 | 1 |
| | | | 1 | 0 | 1 |
| | | | 1 | 0 | 1 |
| | | | 1 | 1 | 1 |
| 정답 | t1 | | | | |
| | t2 | | | | |

| w、b값 | | w | | | b |
|---|---|---|---|---|---|
| 은닉층 | 1 | 0.679 | 0.089 | -0.667 | 0.032 |
| | | -0.742 | -1.534 | -1.140 | |
| | | -0.332 | -1.994 | -0.750 | |
| | | -0.336 | -0.174 | -0.267 | |
| | 2 | -0.309 | -0.505 | 0.126 | -0.662 |
| | | 1.746 | 0.507 | 1.819 | |
| | | -0.065 | 0.861 | -0.482 | |
| | | -0.966 | -0.682 | 0.993 | |
| | 3 | 2.042 | 0.074 | -0.041 | -0.296 |
| | | -0.510 | 1.069 | 0.436 | |
| | | 0.532 | -0.275 | 0.676 | |
| | | 0.382 | 1.301 | 0.115 | |
| 출력층 | 1 | -2.402 | 0.908 | -1.415 | 0.185 |
| | 2 | 1.499 | 0.474 | -1.456 | 1.676 |

| 1 | 회차 | | | | |
|---|---|---|---|---|---|
| | | | 1 | 2 | 3 |
| 변숫값 계산 | 은닉층 | $z^2_i$ | -3.607 | 1.015 | 4.712 |
| | | $a^2_i$ | 0.026 | 0.734 | 0.991 |
| | | $a'(z^2_i)$ | 0.026 | 0.195 | 0.009 |
| | 출력층 | $z^3_i$ | -0.615 | 0.621 | |
| | | $a^3_i$ | 0.351 | 0.650 | |
| | | $a'(z^3_i)$ | 0.228 | 0.227 | |
| | C | | 0.4222 | | |
| δ 계산 | 출력층 | $\partial C/\partial a^3$ | -0.649 | 0.650 | |
| | | $\delta^3$ | -0.148 | 0.148 | |
| | 은닉층 | $\Sigma w\delta^3$ | 0.577 | -0.064 | -0.006 |
| | | $\delta^2$ | 0.015 | -0.012 | 0.000 |

| 기울기 | | $\partial C_T/\partial w$ | | | $\partial C_T/\partial b$ |
|---|---|---|---|---|---|
| 은닉층 | 1 | 0.351 | 0.286 | 0.312 | 0.296 |
| | | 0.466 | -0.212 | 0.436 | |
| | | 0.490 | -0.173 | 0.465 | |
| | | 0.470 | 0.238 | 0.513 | |
| | 2 | -0.142 | -0.119 | -0.176 | -0.162 |
| | | -0.230 | 0.081 | -0.211 | |
| | | -0.254 | 0.118 | -0.257 | |
| | | -0.246 | -0.157 | -0.219 | |
| | 3 | 0.000 | -0.013 | -0.023 | -0.012 |
| | | -0.025 | 0.005 | -0.024 | |
| | | -0.022 | -0.009 | -0.012 | |
| | | 0.012 | 0.027 | -0.012 | |
| 출력층 | 1 | -0.088 | -2.297 | -2.446 | -2.496 |
| | 2 | 0.058 | 2.343 | 2.080 | 2.147 |
| | | | 1회 | $C_T$ | 17.285 |

| 제곱오차의 편미분 | | | $\partial C/\partial w$ | | |
|---|---|---|---|---|---|
| 은닉층 | 1 | | 0.015 | 0.015 | 0.015 |
| | | | 0.015 | 0.000 | 0.015 |
| | | | 0.015 | 0.000 | 0.015 |
| | | | 0.015 | 0.015 | 0.015 |
| | 2 | | -0.012 | -0.012 | -0.012 |
| | | | -0.012 | 0.000 | -0.012 |
| | | | -0.012 | 0.000 | -0.012 |
| | | | -0.012 | -0.012 | -0.012 |
| | 3 | | 0.000 | 0.000 | 0.000 |
| | | | 0.000 | 0.000 | 0.000 |
| | | | 0.000 | 0.000 | 0.000 |
| | | | 0.000 | 0.000 | 0.000 |
| 출력층 | 출력층 | 1 | -0.004 | -0.109 | -0.147 |
| | | 2 | 0.004 | 0.109 | 0.147 |

구현은 위의 첫 번째 데이터에 대한 계산값들을 비교하면서 알고리즘을 작성하였다. 먼저 전체적인 코드의 구조를 설명한 다음 순전파, 역전파 과정을 나누어서 세부적으로 설명하겠다.

1. 코드 구조

먼저 이미지 데이터 즉, 학습 데이터를 읽어와야 하는데, 간단하게 64개의 데이터 값을 작성해주었다.

```
# 1. 학습 데이터 준비 (이미지 패턴, 정답), 총 64개의 데이터
# 64x12 형태의 행렬
image_pattern_data = np.array([
    [1,1,1,1,0,1,1,0,1,1,1,1], # 1
    [0,1,1,1,0,1,1,0,1,1,1,1], # 2
    [1,1,0,1,0,1,1,0,1,1,1,1], # 3
    [1,1,1,1,0,1,1,0,1,1,1,0], # 4
    [1,1,1,1,0,1,1,0,1,0,1,1], # 5
    [0,0,0,1,1,1,1,0,1,1,1,1], # 6
    [0,0,0,0,1,1,1,0,1,1,1,1], # 7
    [0,0,0,1,1,0,1,0,1,1,1,1], # 8
    [0,0,0,1,1,1,1,0,1,1,1,0], # 9
    [0,0,0,1,1,1,1,0,1,0,1,1], # 10
    [1,1,1,1,0,1,1,1,1,0,0,0], # 11
    [0,1,1,1,0,1,1,1,1,0,0,0], # 12
    [1,1,0,1,0,1,1,1,1,0,0,0], # 13
    [1,1,1,1,0,1,1,1,0,0,0,0], # 14
    [1,1,1,1,0,1,0,1,1,0,0,0], # 15
    [1,0,1,1,0,1,1,0,1,1,1,1], # 16
    [1,1,1,1,0,0,1,0,1,1,1,1], # 17
    [1,1,1,1,0,1,1,0,0,1,1,1], # 18
    [1,1,1,1,0,1,1,0,1,1,0,1], # 19
    [1,1,1,1,0,1,0,0,1,1,1,1], # 20
    [1,1,1,0,0,1,1,0,1,1,1,1], # 21
    [0,0,1,1,0,1,1,0,1,1,1,1], # 22
    [0,1,1,1,0,0,1,0,1,1,1,1], # 23
    [0,1,1,1,0,1,1,0,0,1,1,1], # 24
    [0,1,1,1,0,1,1,0,1,1,0,1], # 25
    [0,1,1,1,0,1,0,0,1,1,1,1], # 26
    [0,1,1,0,0,1,1,0,1,1,1,1], # 27
    [1,1,0,1,0,0,1,0,1,1,1,1], # 28
    [1,1,0,1,0,1,1,0,0,1,1,1], # 29
    [1,1,0,1,0,1,1,0,1,1,0,1], # 30
    [1,1,0,1,0,1,0,0,1,1,1,1], # 31
    [1,1,0,0,0,1,1,0,1,1,1,1], # 32
    [0,1,0,0,1,0,0,1,0,0,1,0], # 33
    [1,1,0,0,1,0,0,1,0,0,1,0], # 34
    [0,1,0,0,1,0,0,1,0,0,1,0], # 35
    [0,1,0,0,1,0,0,1,0,1,1,0], # 36
    [0,1,0,0,1,0,0,1,0,0,1,1], # 37
    [1,1,0,0,1,0,0,1,0,1,1,0], # 38
```

데이터를 설정하였으니 이제 각 층의 유닛들에 대한 초기 가중치 값과 편향 값을 설정해야 한다. 초깃값들은 위에서 설명했듯이 엑셀 파일의 값들을 가져와서 사용하였다.

```
# 2. 각 유닛의 가중치와 편향의 초기값 설정
# 정규분포 난수를 이용, 학습률은 적절한 작은 값을 갖는 양의 상수로 설정
iWeightInputToHidden = np.array([
    # 초기 가중치 값 (입력층 -> 은닉층), 3x12 형태의 행렬
    [0.679, 0.089, -0.667, -0.742, -1.534, -1.14, -0.332, -1.994, -0.75,
    -0.336, -0.174, -0.267],
    [-0.309, -0.505, 0.126, 1.746, 0.507, 1.819, -0.065, 0.861, -0.482,
    -0.966, -0.682, 0.993],
    [2.042, 0.074, -0.041, -0.51 , 1.069, 0.436, 0.532, -0.275, 0.676, 0.
    382, 1.301, 0.115]
])

iBiasInputToHidden = np.array([0.032, -0.662, -0.296]) # 초기 편향 값
(입력층 -> 은닉층)

iWeightHiddenToOutput = np.array([
    # 초기 가중치 값 (은닉층 -> 출력층)
    # 2x3 형태로 작성
    [-2.402, 0.908, -1.415],
    [1.499, 0.474, -1.456],
])
iBiasHiddenToOutput = np.array([0.185, 1.676])  # 초기 편향 값 (은닉층 ->
출력층)

learning_rate = 0.2
epochs = 100
```

데이터 저장 형식은 직관적으로 하나의 배열을 하나의 유닛으로 보고 저장하였고 계산할 때는 행렬 곱 형식에 맞춰서 전치행렬로 변환하여 계산하였다.

학습률은 작은 값을 갖는 양의 상수로 설정하는데, 엑셀파일과 동일하게 0.2로 설정하였다.

epoch 값은 많이 돌리면 돌릴수록 좋으므로 적당하게 50에서 100 사이의 값으로 설정하였다.

이제 오차역전파 계산과정을 간략하게 설명한 뒤, 순전파와 역전파 부분을 나누어서 세부적인 구현 내용을 설명하겠다.

```
for epoch in range(epochs):
    # forward
    hidden_layer_output, output_layer_output = calcValues
    (image_pattern_data, WeightInputToHidden, BiasInputToHidden,
    WeightHiddenToOutput, BiasHiddenToOutput)

    # 제곱오차 계산
    squared_error_output = SquaredError(output_layer_output)

    # backward
    # 오차역전파법으로 유닛의 오차 계산
    # 출력층의 유닛의 오차 계산 --> 64x2 형태의 행렬
    EoU3_i = calcErrorOfUnitOfOutputLayer(output_layer_output)

    # 은닉층의 유닛의 오차 계산 --> 64x3 형태의 행렬
    EoU2_i = calcErrorOfUnitOfHiddenLayer(EoU3_i,
    WeightHiddenToOutput, hidden_layer_output)

    # 유닛의 오차에서 제곱오차 C의 편미분 계산
    # 제곱오차 C 편미분 (은닉층)
    # dCdW_H = (64, 3, 12), dCdB_H = (64, 3)
    dCdW_H, dCdB_H = DerivativeOfSquaredError(EoU2_i,
    image_pattern_data)

    # 제곱오차 C 편미분 (출력층)
    # dCdW_O = (64, 2, 3), dCdB_O = (64, 2)
    dCdW_O, dCdB_O = DerivativeOfSquaredError(EoU3_i,
    hidden_layer_output)

    # 비용함수 C_t와 기울기 C_t 계산
    sumOfC = np.sum(squared_error_output)
    dCtdW_H, dCtdB_H = DerivativeOfCostFunction(dCdW_H, dCdB_H)
    dCtdW_O, dCtdB_O = DerivativeOfCostFunction(dCdW_O, dCdB_O)

    # 가중치, 편향 업데이트
    # 입력층 -> 은닉층
    WeightInputToHidden -= (learning_rate * dCtdW_H).T
    BiasInputToHidden -= (learning_rate * dCtdB_H).T
    # 은닉층 -> 출력층
    WeightHiddenToOutput -= (learning_rate * dCtdW_O).T
    BiasHiddenToOutput -= (learning_rate * dCtdB_O).T
```

먼저 순전파 부분으로, 출력층까지 계산하여 유닛의 출력 값을 계산한다.
그 과정에서 은닉층과 출력층의 가중 입력 값 z, 활성화 함수(시그모이드 함수 사용)를 거쳐 나온 출력 값 a를 계산하고 각 층의 출력 값을 저장한다.

그 다음, 출력층의 출력 값을 가지고 제곱오차 C값을 계산한다. 여기까지가 순전파 과정이고, 다음은 역전파 과정이다.

역전파 과정은 오차역전파법으로 유닛의 오차를 계산한다. 반대 방향으로 진행하기 때문에 출력층의 오차를 먼저 계산한 뒤 은닉층의 오차를 계산한다.

유닛의 오차를 계산하였으면 이 오차를 이용해 제곱오차 C의 편미분을 계산한다. 내용은 은닉층, 출력층의 가중치와 편향을 제곱오차 C에 대해 편미분을 하는 것이다. 계산 과정에서 유닛의 오차 값을 이용한다.

여기까지가 역전파 과정이다.

다음으로 비용함수 Ct와 기울기를 계산한 뒤, 마지막으로 경사하강법을 사용해 가중치와 편향 값을 갱신시켜준다.

이 과정을 epoch 설정값에 따라 반복한다. 이제 세부적으로 구현 내용을 살펴보겠다.

## 2. 순전파 과정 설명

순전파 과정은 출력층까지의 가중입력, 출력 값을 계산한 뒤 제곱오차 C를 계산하는 과정까지를 말한다. 먼저 제곱오차 이전 단계를 살펴보겠다.

```python
# 오차역전파 계산
def BackPropagation(image_pattern_data, learning_rate, epochs):
    global iWeightInputToHidden, iWeightHiddenToOutput
    global iBiasInputToHidden, iBiasHiddenToOutput

    # 가중치, 편향
    WeightInputToHidden = iWeightInputToHidden.T
    WeightHiddenToOutput = iWeightHiddenToOutput.T
    BiasInputToHidden = iBiasInputToHidden
    BiasHiddenToOutput = iBiasHiddenToOutput

    for epoch in range(epochs):
        # forward
        hidden_layer_output, output_layer_output = calcValues
        (image_pattern_data, WeightInputToHidden, BiasInputToHidden,
        WeightHiddenToOutput, BiasHiddenToOutput)

        # 제곱오차 계산
        squared_error_output = SquaredError(output_layer_output)
```

먼저 가중치와 편향 초깃값은 행렬 곱 계산을 위해 전치행렬 형태로 저장하였다.

가중입력 값들과 출력 값들을 계산하는 함수 calcValues를 작성하였고 인자로는 64개의 이미지 데이터, 초기 가중치, 편향 값들을 전달하였다.

아래는 calcValues 함수 내부이다.

```python
# 순전파 단계 계산 (제곱오차 제외)
def calcValues(image_pattern_data, WeightInputToHidden, BiasInputToHidden,
               WeightHiddenToOutput, BiasHiddenToOutput):
    # z2_i 계산
    hidden_layer_input = np.dot(image_pattern_data, WeightInputToHidden) + BiasInputToHidden
    # a2_i 계산 --> 64x3 형태의 행렬
    hidden_layer_output = sigmoid(hidden_layer_input)

    # z3_i 계산
    output_layer_input = np.dot(hidden_layer_output, WeightHiddenToOutput) + BiasHiddenToOutput
    # a3_i 계산 --> 64x2 형태의 행렬
    output_layer_output = sigmoid(output_layer_input)

    return hidden_layer_output, output_layer_output
```

가중입력은 가중치와 입력값의 곱 형태를 띄므로 np.dot을 이용해 계산을 해준 뒤, 편향 값을 더해주었다.

출력 값은 활성화 함수를 거쳐서 나온 값으로, 활성화 함수는 시그모이드 함수를 사용하였다. 엑셀 파일의 값과 비교했을 때 값이 거의 비슷하게 나왔다.

시그모이드 함수는 다음과 같이 구현하였다.

```
# activation function (시그모이드 함수 사용)
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

hidden_layer_input이 은닉층의 가중입력 값을 계산한 것이고,
hidden_layer_output이 출력 값을 계산한 것이다.

마찬가지로 output_layer_input과 output_layer_output이 출력층의 가중입력 값과 출력 값을 계산한 것이다.

계산이 끝난 뒤, 추후에 필요한 값들인 은닉층, 출력층의 출력 값을 리턴시켜준다.

이제 마지막으로 제곱오차 C를 계산해준다.

```
# 제곱오차 계산
# t1 = '0'의 정답 데이터 변수, 이미지가 0인 경우 1
# t2 = '1'의 정답 데이터 변수, 이미지가 0인 경우 0
def SquaredError(output_layer_output):
    # 1 ~ 32번 데이터는 이미지가 0이므로 t1 = 1, t2 = 0
    # 33 ~ 64번은 이미지가 1이므로 t1 = 0, t2 = 1

    # 1. 64x2 행렬을 32x2, 32x2 행렬 두 개로 나눔
    zero = output_layer_output[:32]     # 0 ~ 31
    one = output_layer_output[32:]      # 32 ~ 63

    # 2. (t1 - a3_1)^2, (t2 - a3_2)^2 계산
    zero = np.square([1,0]-zero)
    one = np.square([0,1]-one)

    # 3. 2에서 계산한 값에서 1열과 2열의 값 더한 후 나누기 2 진행
    zero = (zero[:,0] + zero[:,1])/2
    one = (one[:,0] + one[:,1])/2

    # 64x1 행렬로 변경 후 합치기
    zero = zero.reshape(32,1)
    one = one.reshape(32,1)
    squared_error_output = np.concatenate((zero, one), axis=0)

    return squared_error_output
```

제곱오차는 $\frac{1}{2}\times((t_1-a_1^3)^2+(t_2-a_2^3)^2)$ 로 계산하므로 출력층의 출력 값이 필요하다. 따라서 인자로 전달해주었고, t1과 t2의 값에 대해서는 이미지 데이터가 1번부터 32번까지는 0에 대한 이미지고, 33번부터 64번까지는 1에 대한 이미지이다. 그에 맞춰서 64개의 출력 값을 저장한 64x2 행렬을 절반으로 나누어서 계산하였다.

1열과 2열은 각각 $a_1^3, a_2^3$의 값을 담고 있으므로 각각 계산한 다음 두 값을 합하였다. 그 다음 절반으로 나누었던 행렬을 64x1 행렬 형태로 바꿔준 다음 다시 붙여주었다.

여기까지가 순전파 과정이었고 이제 역전파 과정을 살펴보겠다.

## 3. 역전파 과정 설명

역전파 과정에는 유닛의 오차를 계산하고, 유닛의 오차를 이용해 제곱오차에 대한 가중치, 편향의 편미분 값을 계산하는 과정이 있다.

```
# 2. backward
# 오차역전파법으로 유닛의 오차 계산
# 출력층의 유닛의 오차 계산 --> 64x2 형태의 행렬
EoU3_i = calcErrorOfUnitOfOutputLayer(output_layer_output)

# 은닉층의 유닛의 오차 계산 --> 64x3 형태의 행렬
EoU2_i = calcErrorOfUnitOfHiddenLayer(EoU3_i, WeightHiddenToOutput, hidden_layer_output)

# 유닛의 오차에서 제곱오차 C의 편미분 계산
# 제곱오차 C 편미분 (은닉층)
# dCdW_H = (64, 3, 12), dCdB_H = (64, 3)
dCdW_H, dCdB_H = DerivativeOfSquaredError(EoU2_i, image_pattern_data)

# 제곱오차 C 편미분 (출력층)
# dCdW_O = (64, 2, 3), dCdB_O = (64, 2)
dCdW_O, dCdB_O = DerivativeOfSquaredError(EoU3_i, hidden_layer_output)
```

먼저 유닛의 오차는 calcErrorOfUnitOfOutputLayer 함수와 calcErrorOfUnitOfHiddenLayer 함수에서 계산하였고, 각각 출력층과 은닉층의 유닛의 오차를 계산하였다.

먼저 출력층의 유닛 오차를 계산하는 함수의 내부 구현은 다음과 같다.

```python
# 출력층의 유닛 오차 계산
def calcErrorOfUnitOfOutputLayer(output_layer_output):
    # 1. 64x2 행렬을 32x2, 32x2 행렬 두 개로 나눔
    zero = output_layer_output[:32]      # 0 ~ 31 -> t1 = 1, t2 = 0
    one = output_layer_output[32:]       # 32 ~ 63 -> t1 = 0, t2 = 1

    # 2. 계산 및 합치기
    zero = (zero-[1,0])*DerivativeOfSigmoid(zero)
    one = (one-[0,1])*DerivativeOfSigmoid(one)
    EoU3 = np.concatenate((zero, one), axis=0)

    return EoU3
```

시그모이드 함수의 미분은 다음과 같이 구현하였다.

```python
# 시그모이드 함수의 미분값 계산
def DerivativeOfSigmoid(a):
    return a*(1-a)
```

출력층 유닛의 오차 값은 출력값에서 t1 혹은 t2 값을 뺀 뒤 해당 가중입력 값에 대한 시그모이드 함수의 미분 값을 곱해주면 된다. 마찬가지로 행렬을 반으로 구분하여 계산한 다음 합쳐주었다.

다음은 은닉층 유닛의 오차를 계산하는 함수의 내부 구현이다.

```python
# 은닉층의 유닛 오차 계산
def calcErrorOfUnitOfHiddenLayer(EoU3, WeightHiddenToOutput, hidden_layer_output):
    # 3x2 --> 2x3
    w3_i = WeightHiddenToOutput.T

    # 64x3 형태의 행렬
    EoU2 = np.dot(EoU3, w3_i) * DerivativeOfSigmoid(hidden_layer_output)

    return EoU2
```

은닉층 유닛 오차 값은 가중치와 오차의 내적 값과 시그모이드 미분 값을 곱해주면 된다.

먼저 가중치를 저장한 행렬이 3x2 형태이다. (전치행렬 상태)
이 행렬을 행렬 곱 계산을 위해 2x3 행렬로 만들어주었고 위의 식대로 계산하였다.

이제 제곱오차에 대한 편미분을 계산하는 과정이다.

```
# 2. backward
# 오차역전파법으로 유닛의 오차 계산
# 출력층의 유닛의 오차 계산 --> 64x2 형태의 행렬
EoU3_i = calcErrorOfUnitOfOutputLayer(output_layer_output)

# 은닉층의 유닛의 오차 계산 --> 64x3 형태의 행렬
EoU2_i = calcErrorOfUnitOfHiddenLayer(EoU3_i, WeightHiddenToOutput, hidden_layer_output)

# 유닛의 오차에서 제곱오차 C의 편미분 계산
# 제곱오차 C 편미분 (은닉층)
# dCdW_H = (64, 3, 12), dCdB_H = (64, 3)
dCdW_H, dCdB_H = DerivativeOfSquaredError(EoU2_i, image_pattern_data)

# 제곱오차 C 편미분 (출력층)
# dCdW_O = (64, 2, 3), dCdB_O = (64, 2)
dCdW_O, dCdB_O = DerivativeOfSquaredError(EoU3_i, hidden_layer_output)
```

제곱오차를 계산하는 과정은 DerivativeOfSquaredError 함수에서 작성하였고
인자로는 유닛의 오차 값과 출력 값을 전달해준다.

내부는 다음과 같이 구현하였다.

```
# 제곱오차 C의 가중치에 관한 편미분 계산
def DerivativeOfSquaredError(EoU, ai_j):
    dataEoU = EoU.shape[1]      # 유닛의 오차의 데이터 개수(은닉층=3, 출력층=2)
    dataAi_j = ai_j.shape[1]    # 각각의 데이터의 원소의 개수

    total_w = []
    total_b = []

    for i in range(0, 64):
        res_w = []
        res_b = []

        for j in range(dataEoU):
            res_w.append(ai_j[i]*EoU[i][j])
            res_b.append(EoU[i][j])
        total_w.append(res_w)
        total_b.append(res_b)

    dCdW = np.array(total_w)    # 64개의 데이터에 관하여 은닉층은 3x12 형태
                                # 출력층은 2x3 형태 -> (64, 2, 3)

    dCdB = np.array(total_b)    # 64개의 데이터에 관해 은닉층은 64x3 형태
                                # 출력층은 64x2 형태 -> (64, 2)

    return dCdW, dCdB
```

편미분 값은 유닛의 오차 값과 출력 값을 곱한 형태를 띄기 때문에 공식에 맞춰서

각 유닛의 가중치, 편향에 대한 편미분 값을 계산하여 저장하였다.

최종적으로는 은닉층의 경우, 64개의 데이터에 대해서 3개의 유닛에 대한 각 12개의 가중치에 대한 편미분 값과 각 1개의 편향 값에 대한 편미분 값을 계산하여 저장하였고 행렬의 형태가 가중치의 경우 64개의 3x12 형태의 행렬로 구성이 되었고, 편향은 64x3 형태가 되었다.

출력층도 마찬가지로 64개의 데이터에 대해 2개의 유닛에 대해 각 3개의 가중치와 각 1개의 편향에 대한 편미분 값을 계산하여 2x3 행렬의 형태와 64x2 행렬의 형태로 저장하였다.

여기까지가 역전파 과정이다.

## 4. 비용함수와 기울기 계산

이제 비용함수와 기울기를 계산해준다.

```
# 3. 비용함수 C_t와 기울기 C_t 계산
sumOfC = np.sum(squared_error_output)
dCtdW_H, dCtdB_H = DerivativeOfCostFunction(dCdW_H, dCdB_H)
dCtdW_O, dCtdB_O = DerivativeOfCostFunction(dCdW_O, dCdB_O)
```

먼저 비용함수는 64개 데이터에 대한 제곱오차 값을 더해주면 되므로 np.sum을 이용하여 제곱오차 값을 저장한 행렬의 모든 요소의 값을 더하여 계산하였다.

기울기는 DerivativeOfCostFunction 함수를 작성하여 계산하였고, 계산 과정은 간단하게 64개 데이터에 대한 각 유닛의 같은 인덱스의 편미분 값들을 더해주면 된다. 다음과 같이 구현하였다.

```
# 가중치, 편향 기울기 계산
def DerivativeOfCostFunction(dCdW, dCdB):
    # 은닉층은 가중치는 3x12, 편향은 1x3 형태
    # 출력층은 가중치는 2x3, 편향은 1x2 형태

    dCtdW_H = dCdW[0]
    dCtdB_H = dCdB[0]

    for i in range(1,64):
        dCtdW_H += dCdW[i]
        dCtdB_H += dCdB[i]

    return dCtdW_H, dCtdB_H
```

64개 데이터에 관한 편미분 값들을 같은 자리에 있는 값끼리 더해주면 된다.

이제 가중치 업데이트만 해주면 된다.

## 5. 가중치, 편향 업데이트

```
# 4. 가중치, 편향 업데이트
# 입력층 -> 은닉층
WeightInputToHidden -= (learning_rate * dCtdW_H).T
BiasInputToHidden -= (learning_rate * dCtdB_H).T
# 은닉층 -> 출력층
WeightHiddenToOutput -= (learning_rate * dCtdW_O).T
BiasHiddenToOutput -= (learning_rate * dCtdB_O).T
```

경사하강법을 이용하여 가중치와 편향을 갱신시켜준다.
기존의 가중치에서 학습률과 4번에서 구한 비용함수에 대한 편미분 값을 곱하여 빼주면 된다.

따라서 위의 코드처럼 작성해주면 된다.

이제 테스트 데이터를 넣어서 학습이 잘 되었는지, 결과가 알맞게 나오는지 확인해보면 된다.

## 6. 학습 과정과 테스트 결과

테스트 데이터를 작성하여 넣어주었고 다음과 같이 테스트를 진행하였다.

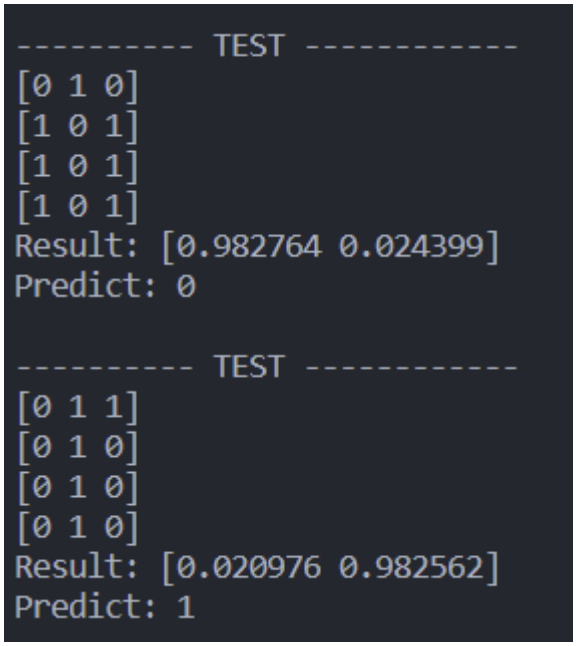
```
# training 및 test
wH, bH, wO, bO = BackPropagation(image_pattern_data, learning_rate, epochs)
# Test
image_test_data = np.array([
    [0,1,0,1,0,1,1,0,1,1,0,1],
    [0,1,1,0,1,0,0,1,0,0,1,0],
])

print('\n\n')

for data in image_test_data:
    a2i, a3i = calcValues(data, wH, bH, wO, bO)
    print("---------- TEST ------------")
    print(data[0:3])
    print(data[3:6])
    print(data[6:9])
    print(data[9:12])
    print("Result:", a3i)

    if a3i[0] > a3i[1]:
        print("Predict:", 0, '\n')
    else:
        print("Predict:", 1, '\n')
```

아래는 학습 과정을 나타낸 결과이다. epoch는 50번으로 설정하였다.
오차가 17에서 0.169로 줄어든 것을 확인할 수 있다.

```
[*] epoch = 1, error = 17.282379219983497
[*] epoch = 2, error = 15.305646877170993
[*] epoch = 3, error = 14.166693124717666
[*] epoch = 4, error = 12.847964535538736
[*] epoch = 5, error = 11.365572304399752
[*] epoch = 6, error = 9.895890582795037
[*] epoch = 7, error = 8.506059200915216
[*] epoch = 8, error = 7.459156359787953
[*] epoch = 9, error = 5.677616610315738
[*] epoch = 10, error = 4.676655444584904
[*] epoch = 11, error = 2.831529900957028
[*] epoch = 12, error = 2.1274854442384283
[*] epoch = 13, error = 1.5838730391288074
[*] epoch = 14, error = 1.33466170917717
[*] epoch = 15, error = 1.1624698911570484
[*] epoch = 16, error = 1.031348926582883
[*] epoch = 17, error = 0.9255367100011308
[*] epoch = 18, error = 0.8379426227779798
[*] epoch = 19, error = 0.7640541641250651
[*] epoch = 20, error = 0.7007702371410551
[*] epoch = 21, error = 0.6458942762194552
[*] epoch = 22, error = 0.5978339945398465
[*] epoch = 23, error = 0.5554087867001076
[*] epoch = 24, error = 0.5177227647271204
[*] epoch = 25, error = 0.4840793190453519
```

```
[*] epoch = 26, error = 0.45392301716595507
[*] epoch = 27, error = 0.4268000878075387
[*] epoch = 28, error = 0.4023317316643465
[*] epoch = 29, error = 0.3801962270543682
[*] epoch = 30, error = 0.3601169170408227
[*] epoch = 31, error = 0.34185399225245416
[*] epoch = 32, error = 0.3251986371997681
[*] epoch = 33, error = 0.30996861441685597
[*] epoch = 34, error = 0.2960047278620812
[*] epoch = 35, error = 0.2831678511412281
[*] epoch = 36, error = 0.2713363541286558
[*] epoch = 37, error = 0.2604038429387584
[*] epoch = 38, error = 0.2502771682865904
[*] epoch = 39, error = 0.24087467446675837
[*] epoch = 40, error = 0.23212466716911664
[*] epoch = 41, error = 0.22396407975016136
[*] epoch = 42, error = 0.21633731769220485
[*] epoch = 43, error = 0.20919526120248966
[*] epoch = 44, error = 0.2024944066477133
[*] epoch = 45, error = 0.19619612876949058
[*] epoch = 46, error = 0.19026604721942625
[*] epoch = 47, error = 0.18467348270989203
[*] epoch = 48, error = 0.179390989854338
[*] epoch = 49, error = 0.17439395547070563
[*] epoch = 50, error = 0.16966025268528234
```

다음은 엑셀에 있는 테스트용 이미지 데이터를 넣었을 때의 결과이다.




엑셀 파일의 테스트 시트에 있는 값을 넣어주었고 결과는 올바르게 나오는 것을 확인할 수 있다.