Machine Learning

# Multi-Layer Perceptron

HW2

32171550 박다은

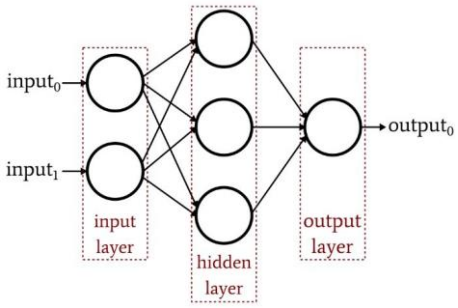
2020-11-19

# Contents

# 1. Overview

## (1) Introduction

다층퍼셉트론(Multi-Layer Perceptron)은 입력층과 출력층 사이에 중간층이 존재하는 신경망으로, 다음 그림과 같은 구조를 갖는다. 입력층과 출력층 사이의 중간층을 은닉층이라 부른다.




## (2) Goals

다층퍼셉트론을 통해 학습된 output 과 실제 목표 값 사이의 비용함수가 최소가 되는 weight 를 구하는 것이 주된 목표이다.

## (3) Concepts in MLP

다층퍼셉트론은 다음과 같이 동작한다.

1. 각 층에서의 가중치를 임의의 값으로 설정한다. 각 층에서 바이어스 값은 1 로 설정한다.

2. 하나의 트레이닝 데이터에 대해 각 층에서의 순입력 함수 값을 계산하고 활성함수에 의한 출력 값을 계산한다.

3. 출력층의 활성 함수에 의한 결과값이 실제 값과의 오차가 줄어들도록 각 층에서의 가중치를 업데이트한다.

4. 모든 데이터에 대해서 출력층의 활성 함수에 의한 결과값이 실제 값과의 오차가 매우 작아지면 학습을 종료한다.

다층퍼셉트론에는 은닉층이 존재하므로 가중치를 업데이트할 때 은닉층의 출력 값에 대한 기준 값을 정의할 수 없다. 따라서 이러한 문제를 해결하기 위해 역전파(backpropagation) 개념을 사용한다.

## (4) Program Structure

### ① Task1

```
+----------------------------
|        Set x, y and weight
|        training
|       +-----------------------
|       | forward
|       | backpropagation
|       | update weight
|       +-----------------------
+----------------------------
```

### ② Task2

Open_file

```
+----------------------------
|        Import x datas and y datas
+----------------------------
```

Close_file

```
+----------------------------
|        check data
|        MLP
|       +-----------------------
|       |     set number of nodes
|       |     set x,y and weight
|       |     training
|       +-----------------------
|        Graphing final output
```

+----------

## (5) Build Environment

Jupyter Notebook, Python3

# 2.Code & Consideration

| Input: hw2_data.csv<br>Output: standard output |
|---|

## (1) Task1 주요 코드

```
5  def sigmoid(x):
6      return 1/(1+ np.exp(-x))
7
8  def de_sigmoid(x):
9      return sigmoid(x) * (1 - sigmoid(x))
10
11 def add_bias(x):
12     x = np.c_[x,[0,0,0,0]]
13     bias = 1
14     for i in x:
15         i[2] = i[1]
16         i[1] = i[0]
17         i[0] = bias
18     return x
```

- 활성함수로 사용되는 sigmoid 를 함수로 만들어주었다. de_sigmoid()는 미분한 함수이다.
- add_bias()는 입력층과 은닉층에서 bias 를 추가시킬 때 사용되는 함수이다.

  Bias 는 1 로 설정하였고, 4x2 행렬의 1 열에 [1, 1, 1, 1]을 추가해 4x3 행렬로 만드는 역할을 하는 함수이다.

```
20 #input layer
21 Input = np.array([[0,0], [0,1], [1,0], [1,1]])
22 #output layer
23 y = np.array([[0], [1], [1], [0]])
24
25 x = add_bias(Input)
```

- XOR 의 input 과 output 을 각각 4x2, 4x1 행렬로 만들어주었다.
- x 는 bias 가 추가된 input 이 되고, y 는 output 그대로 되도록 설정하였다.

```
27 #hidden weight
28 hidden_w = np.random.rand(3,2)
29 #output weight
30 output_w = np.random.rand(3,1)
31 #learning rate
32 alpha = 0.1
```

- weight 는 랜덤으로 받도록 하였다.
- 입력층, 은닉층, 출력층의 노드의 개수(bias 포함)는 각각 3, 3, 1 이다. 은닉층의 노드의 수가 3 개이지만 bias 를 제외한 노드의 개수는 2 개이기 때문에 은닉층의 행렬 사이즈를 3x2 로 설정하였다. 출력층의 행렬사이즈는 예외적인 요소가 없으므로 3x1 로 설정하였다.
- Learning rate 의 값은 0.1 로 설정하였다.

```
37 #training
38 for i in range(100000):
39     hidden_z = np.dot(x, hidden_w)
40     hidden_a = sigmoid(hidden_z)
41 
42     output_i = add_bias(hidden_a)
43     output_z = np.dot(output_i, output_w)
44     output_a = sigmoid(output_z)
45 
46     #backpropagation
47     output_delta = (y - output_a) * de_sigmoid(output_z)
48     hidden_delta = output_delta.dot(output_w[1:].T) * de_sigmoid(hidden_z)
49 
50     #update weight
51     output_w += alpha * output_i.T.dot(output_delta)
52     hidden_w += alpha * x.T.dot(hidden_delta)
```

- Iteration 은 100,000 으로 설정하였다.
- 각 층의 z 는 각 층의 input 과 weight 를 내적한 값이고, 각 층의 a 는 이 값을 sigmoid 함수에 적용시킨 값이다. 은닉층의 a 와 출력층의 input 은 bias 의 존재 때문에 다르다. 은닉층의 a 에서 bias 를 추가시킨 것이 출력층의 input 이므로 미리 설정한 add_bias()함수를 통해 input 을 업데이트 시켜주었다.
- Weight 를 업데이트 시키며 for 문을 반복하는데, 이 때 weight 업데이트는 다음과 같다.

$$w := w - \alpha \frac{\partial J}{\partial w}$$

- 비용함수는 MSE 를 사용하였다. 즉, 비용함수 J 는 $\frac{1}{2}\|a-y\|^2$이다.

  여기서, $\frac{\partial J}{\partial w} = \frac{\partial J}{\partial a} * \frac{\partial a}{\partial z} * \frac{\partial z}{\partial w}$ 를 적용시킬 것이다. $\frac{\partial J}{\partial a} = (a-y)$ 이고, $\frac{\partial a}{\partial z} = \sigma'(z)$ 이다. $\frac{\partial J}{\partial a} * \frac{\partial a}{\partial z} = \delta$ 라고 하므로, 따라서 output_delta = (y – output_a) * de_sigmoid(output_z)이다.

  hidden_delta 에서는 output_delta 에 출력층 weight 의 역함수를 내적해야 하는데, 이 때 bias 의 weight 는 포함되지 않으므로 output_w[1:]로 설정하였다. 따라서 마찬가지로 hidden_delta = output_delta.dot(output_w[1:].T) * de_sigmoid(hidden_z)이다.

- 계속해서, $\frac{\partial z}{\partial w}$ 는 현재 layer 의 input 이다. 즉, 출력층의 $\frac{\partial z}{\partial w}$는 output_i 이고, 은닉층의 $\frac{\partial z}{\partial w}$는 x 인 것이다.

  따라서 $\frac{\partial J}{\partial w}$는 각 층의 delta 와 input 을 곱한 값이므로, $w := w - \alpha \frac{\partial J}{\partial w}$를 적용시키면,

  output_w += alpha * output_i.T.dot(output_delta)이고,

  hidden_w += alpha * x.T.dot(hidden_delta)가 된다.

```
Random hidden weight:
 [[0.97034754 0.32214423]
 [0.48036978 0.03356395]
 [0.3105447  0.83112821]]

Random output weight:
 [[0.14573028]
 [0.11057919]
 [0.83963978]]


Final hidden_weight:
 [[-7.35638994 -3.04461165]
 [ 4.79815221  6.78256474]
 [ 4.79146872  6.75395858]]

Final output_weight:
 [[ -4.78953985]
 [-11.051237  ]
 [ 10.31246147]]


Output:  [[0.01302367 0.98887058 0.98889123 0.01144408]]
```

- 위 알고리즘을 통해 구한 weight 는 다음과 같다.
- 출력은 처음 random 으로 받은 각 층의 weight 값, 학습을 통해 정해진 final weight 값과 학습을 통한 출력을 순서대로 하였다.
- 목표 출력인 [0 1 1 0]과 매우 근사하게 값이 나온 것을 확인할 수 있다.

## (2) Task2 주요 코드

### csv에서 data 가져오기

```
x = []
x_list = []
y = []
y_list = []

f = open('hw2_data.csv', 'r', encoding='utf-8')
rdr = csv.reader(f)

i = 0
for line in rdr:
    x.append([float(line[0])])
    x_list.append(float(line[0]))
    y.append([float(line[1])])
    y_list.append(float(line[1]))
    i += 1
f.close()

x = np.array(x).T
y = np.array(y).T
```

- x, y 는 각각 data 들이 담길 행렬이고, x_list 와 y_list 는 데이터 분포를 확인하기 위해 데이터의 최대값과 데이터의 개수를 구할 때 사용될 리스트이다.
- x, y 는 각각 전치를 통해 nx1 행렬이 되도록 만들어주었다.

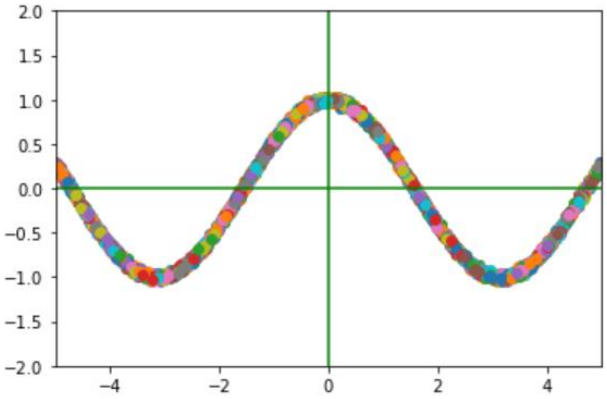
### 데이터 분포 확인

```
print("# of x data: ", len(x_list), " # of y data: ", len(y_list))
print("max(x):", max(x_list), "min(x):", min(x_list))
print("max(y):", max(y_list), "min(y):", min(y_list))
```

```
# of x data:  5000  # of y data:  5000
max(x): 4.999333569 min(x): -4.996034381
max(y): 1.036138896 min(y): -1.045201682
```

- data 분포는 다음과 같이 확인되었다.
- 각 데이터들의 개수는 5000 개이고 x 는 -5~5, y 는 -1.1~1.1 과 같은 분포를 보인다.

```
plt.plot(x,y,'o')
plt.xlim([-5,5]); plt.ylim([-2,2])
plt.axvline(x=0, c='g'); plt.axhline(y=0, c='g')
plt.show()
```




- 확인된 데이터분포를 통해 그래프를 그려보았다.
- 목표하는 데이터들의 값이 4 차방정식을 그리는 것을 확인할 수 있다.

## Multi-Layer Perceptron Training

```
#activation functions
def sigmoid(x):
    return 1/(1+ np.exp(-x))

def de_sigmoid(x):
    return sigmoid(x) * (1 - sigmoid(x))

def identity(x):
    return x

def de_identity(x):
    return 1
```

- 활성함수는 sigmoid 와 identity(f(x)=x)를 사용하였다.
- 은닉층의 활성함수는 sigmoid 를 사용하였고, 출력층의 활성함수는 identity 를 사용하였다.

```
#number of input nodes and output nodes
inputNode = 5000
outputNode = 5000

#number of hidden nodes
hiddenNode = 5
```

- 모델은 다음과 같이 설정하였다.
- 앞에서 확인해본 결과 x, y 의 데이터 개수가 각각 5000 개 이므로 입력층과 출력층의 노드의 개수를 둘 다 5000 으로 설정하였다. 은닉층의 노드 수는 임의의 수로 5 로

설정해보았다. 5 로 설정하였을 때 잘 작동하길래 노드 수를 늘리거나 줄이는 등의 작업은 하지 않았다.

```
#hidden weight
hidden_w = np.random.rand(inputNode,hiddenNode)
#output weight
output_w = np.random.rand(hiddenNode,outputNode)
#learning rate
alpha = 0.1
```

- Weight 는 각 층에 맞는 사이즈로 랜덤으로 받아오도록 하였다.
- Learning rate 는 task1 과 같이 0.1 로 설정하였다.

```
#training
for i in range(10000):
    hidden_z = np.dot(x, hidden_w)
    hidden_a = sigmoid(hidden_z)

    output_z = np.dot(hidden_a, output_w)
    output_a = identity(output_z)

    #backpropagation
    output_delta = (y - output_a) * de_identity(output_z)
    hidden_delta = output_delta.dot(output_w.T) * de_sigmoid(hidden_z)

    #update weight
    output_w += alpha * hidden_a.T.dot(output_delta)
    hidden_w += alpha * x.T.dot(hidden_delta)
```

- 학습방법은 task1 과 같은 방법으로 작동하므로 부연설명은 하지 않겠다.
- 하나 다른 점은, task1 에서는 은닉층과 출력층의 활성함수를 둘 다 sigmoid 함수로 설정하였는데, task2 에서는 y 값에 음수가 존재하므로 양수의 값만 출력되는 sigmoid 함수 대신 identity 함수를 사용하였다.

```
Final hidden_weight:
 [[0.48747082 0.4411756  0.09830759 0.92190448 0.88592781]
 [0.02846539 0.76361737 0.30150176 0.68912704 0.26057913]
 [0.13103469 0.19651282 0.98186228 0.85061285 0.59808438]
 ...
 [0.07140572 0.64070233 0.54814943 0.91582434 0.33363052]
 [0.67242007 0.9277173  0.36344131 0.69590222 0.05510524]
 [0.40132613 0.82214831 0.2626464  0.81245042 0.57139609]]

Final output_weight:
 [[-0.35406361 -0.32588195 -0.11528307 ...  0.0403119   0.17290782
   0.14185319]
 [ 0.29305734  0.13287547  0.54903662 ...  0.37344641  0.12738166
   0.10429373]
 [-0.18118516  0.18916242  0.20834244 ... -0.45341133 -0.36198742
```

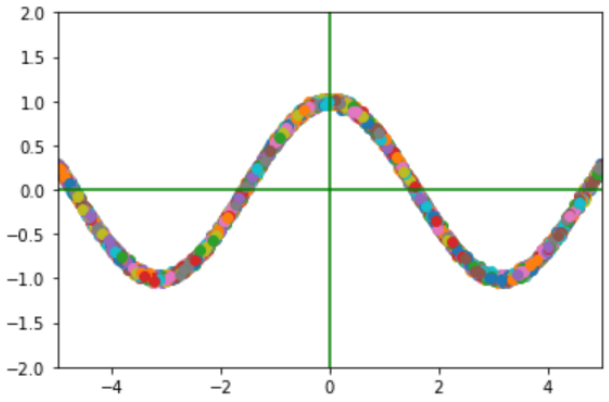
```
 -0.10820632]
[-0.66000564 -0.56285979  0.06132794 ...  0.38845486 -0.12149193
  0.44148883]
[-0.07155325 -0.17901451  0.24585093 ... -0.36711002  0.11278442
  0.38218305]]


Output:  [[-0.97375032]
 [-0.74571837]
 [ 0.94927486]
 ...
 [-0.01830818]
 [-0.07040545]
 [ 0.96161248]]
```

- 출력은 다음과 같다. task1 과 같은 순서로 출력하였다.

### Training 결과 확인

```
1  plt.plot(x,output_a,'o')
2  plt.xlim([-5,5]); plt.ylim([-2,2])
3  plt.axvline(x=0, c='g'); plt.axhline(y=0, c='g')
4  plt.show()
```




- 결과를 그래프로 그려보았다.
- 목표와 같은 4 차함수가 나온 것을 확인할 수 있다.

## (3) Problems & Solutions

### ① bias

Task1 에서 처음에 bias 를 설정하고 weight 를 계산할 때 포함시키지 않았었다. 코드를 거의 완성시킨 후에 깨닫아서 완성된 코드에 bias 를 추가하였는데 이 때 오류가 굉장히 많이 발생하였었다. 행렬 내적할 때 크기도 맞지 않고, 오류를 하나 해결해도 또 발생하는 등

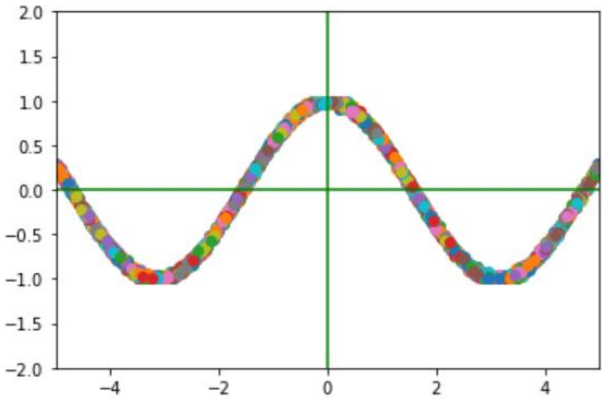
끊임없이 발생하였었다. 이 때 add_bias 라는 함수도 만들었던 것인데, 처음부터 bias 에 weight 를 넣는 것을 고려하였으면 더 보기 좋은 코드가 나올 수 있었을 것 같아서 아쉽다.

### ② matrix

Task2 에서 csv 파일을 불러올 때, 리스트로 받은 다음에 np.array 함수를 이용해 행렬로 바꾸는 방법을 사용하였었다. 겉으로 보기에는 문제가 없어 보였는데, 전치가 되지 않았다. 오류가 나지는 않았지만 전치를 하기 전과 후가 똑같았다. 그래서 사이즈를 출력해보니 (5000, )라는 식으로 출력이 되었다. 겉으로 보기에는 5000x1 행렬이지만 컴퓨터 상에서는 그렇지 않았던 것이다. 리스트를 제대로 된 행렬로 바꾸는 방법을 계속 검색해보았지만 1 차원리스트를 바꾸는 방법은 명확하게 나오지 않았다. 결국 구현은 처음 파일을 불러올 때 1x5000 행렬로 받고 그것을 전치해서 이용하는 방법을 사용했다.

### ③ activation function

출력층의 활성함수를 sigmoid 를 사용하면 안되는 것을 인지하고 있었다. 따라서 활성함수 표를 보며 출력층의 활성함수를 정할 때 처음에는 탄젠트함수를 사용했었다. 그리고 학습을 시켜보았을 때 다음과 같이 결과가 나왔다.




겉으로 보기에는 큰 문제가 없어 보이지만, 1 이상과 -1 이하에서는 그래프가 잘리는 것을 확인할 수 있다. 탄젠트함수의 y 값이 -1~1 이기 때문이다. 따라서 활성함수를 y=x 로 바꾸어 주었더니 제대로 작동하였다.

## (4) Personal Feeling

첫번째 과제 이후에 중간고사가 끝나고 주어진 두번째 과제였다. 과제를 다 끝내고 보고서를 쓰고 있는 지금 굉장히 뿌듯한 기분이지만, 개념을 정확히 확립하지 않은 채 코드를 짠 것은 매우 아쉬운 부분으로 남는다. 그 결과로 task1 에서 2-(3)-①같은 시행착오를 낳게 되었기

때문이다. 필요 없는 오류를 생성했고 결과적으로 시간도 더 오래 걸리게 되었다. 앞으로 과제가 더 남아있는 것으로 아는데, 다음 과제에서는 같은 실수를 반복하지 않도록 할 것이다.

처음 과제를 시작할 때는 복잡해 보이는 식들로 과연 해낼 수 있을지 많이 걱정이 되었었다. 특히 task2 는 정말 시도만 해봐야겠다는 생각을 가지고 있었는데 task1 만 제출하기에 아쉬운 마음도 들고 잠을 덜자면 해낼 수도 있겠다는 생각에 더욱 노력을 하였고, 결국 해냈다.

저번 과제였던 Gradient Descent 에 비해 이번 과제는 난이도가 좀 있던 과제였다. 저번 과제처럼 코드를 작성하는 게 순탄하진 않았지만 그만큼 뿌듯함은 더 큰 것 같다. 이렇게 큰 과제를 하나하나씩 해내다 보면 점점 실력이 늘어가는 게 느껴진다. 기계학습에 관해 흥미도 많이 생겼고, 다음 과제도 열심히 참여하여 완벽히 해내고 싶다.