인공지능

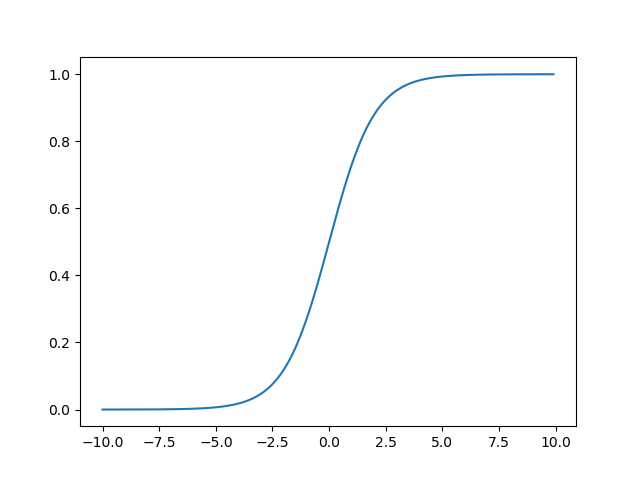
(기말고사)

20164091

송희령

@ 1번문제.

주로 쓰이는 4개의 함수들을 정리해 보았습니다.

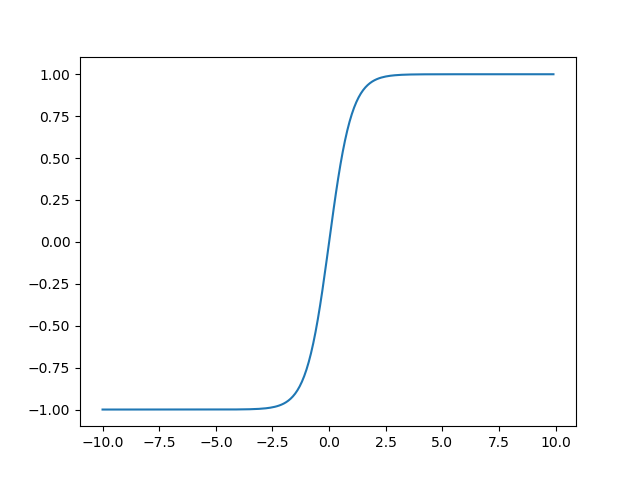


1> 시그모이드(Sigmoid) 함수 (

Output 값을 0에서 1로 설정. 출력층 노드에서 0~1사이값 구현시 사용

하지만 양 극단으로 갈수록 기울기가 점점 없어지므로 역전파로 학습시

Vanishing Gradient 문제가 발생하여 학습이 되지 않게되는 문제가 있다.

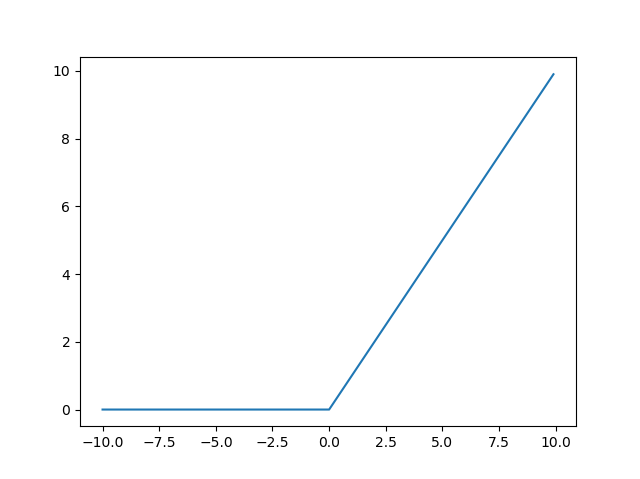


2> 하이퍼볼릭탄젠트 (tanh) 함수

시그모이드와 유사하며 -1~1이며 데이터 평균은 0이다.

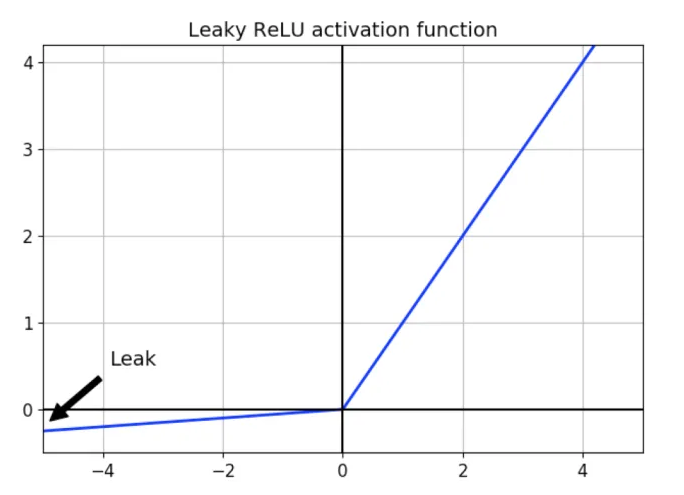
데이터 평균이 0이란 점에서 시그모이드보다 학습이 잘되는 장점이 있지만

이 역시 Vanishing Gradient가 발생하는 문제가 있다.



3> 렐루(ReLU) 함수

대부분의 Input 값에 대해서 기울기가 0이 아니기에 학습이 빨리되며 음수의 경우 0으로 치부하기 때문에 속도와 학습면에서 우수하다 할 수 있다.



4> 리키렐루(Leaky ReLU) 함수

렐루함수와 유사하며 인풋값이 0이 아닌 0.01로 기울기가 0이 아닌

0.01값을 가지게 되는 함수이다.

렐루함수보다 학습이 더 잘되는 장점이 있다.

이 외에 계단함수, 소프트맥스, ELU함수 등이 있습니다.

@ 2번문제.

오차함수를 나타낸 곡선에서 y를 최소화하는 x값을 찾는 것이 목표이다. 현재 x지점의 기울기가 양의 기울기이면 x값을 감소시키고, 음의 기울기면 x값을 증가시키며 y값을 찾아간다. 그러기 위해선 오차함수의 해당 지점에 대한 기울기를 구해야 하는데 기울기를 구하려면 미분을 적용해야 한다. 미분을 적용한 값이 아주 작아지는 지점이 곧 기울기가 가장 작은 y의 최소값이 되는 지점이므로 해당 과정을 반복하여 진행한다.

과정을 진행하면서 x값의 증감치는 학습률에 따라 결정한다. 학습률이 작으면 진행 속도는 느려지게 되고 너무 커지면 학습구간의 오버슈팅(손실함수값이 줄지 않고 오히려 커지는 문제)이 발생할 수 있기에 학습률은 적당히 설정해야한다. 해당 문제에선 0.3으로 주어졌으므로 0.3씩 증감하게 된다.

@ 3번문제.

챕터 13의 grad\_descent.py를 사용하였습니다.

# -\*- coding: cp949 -\*-

x = 20 #초기 x값

learning\_rate = 0.01 #학습률

max\_iterations = 100    # 반복회수

def loss\_func(x): return 0.33 \* pow(x, 3) + 50 \* pow(x, 2) - 100 \* x - 30 #손실함수

def gradient(x): return 0.99 \* pow(x, 2) + 100 \* x - 100 #그라디언트(손실함수 미분값)

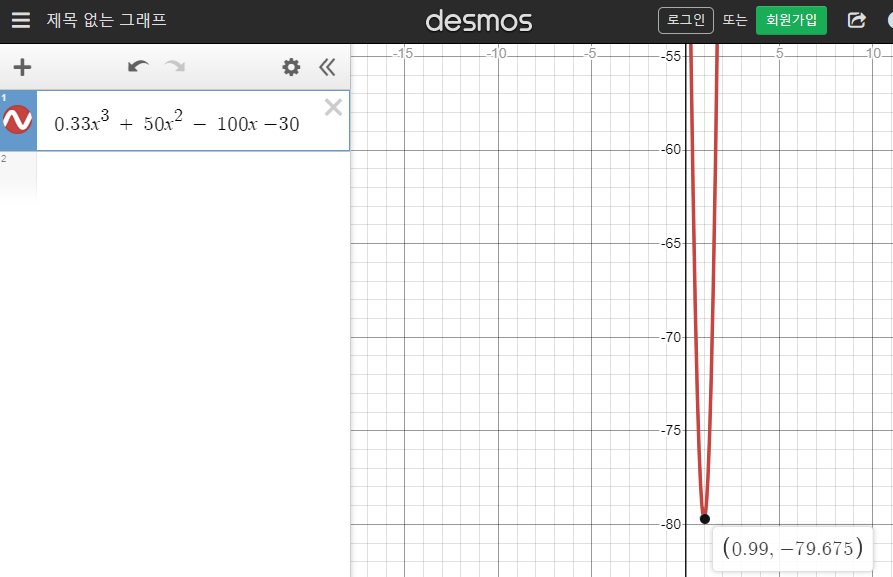
for i in range(max\_iterations):

    x = x - learning\_rate \* gradient(x)

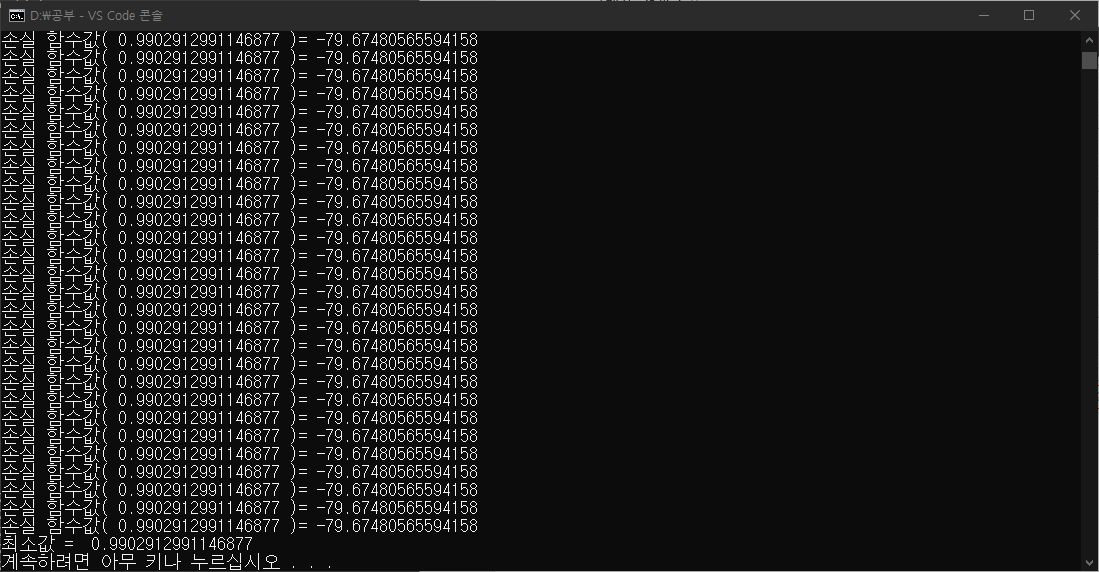
    print("손실 함수값(", x, ")=", loss\_func(x))

print("최소값 = ", x)

그래프 결과 :

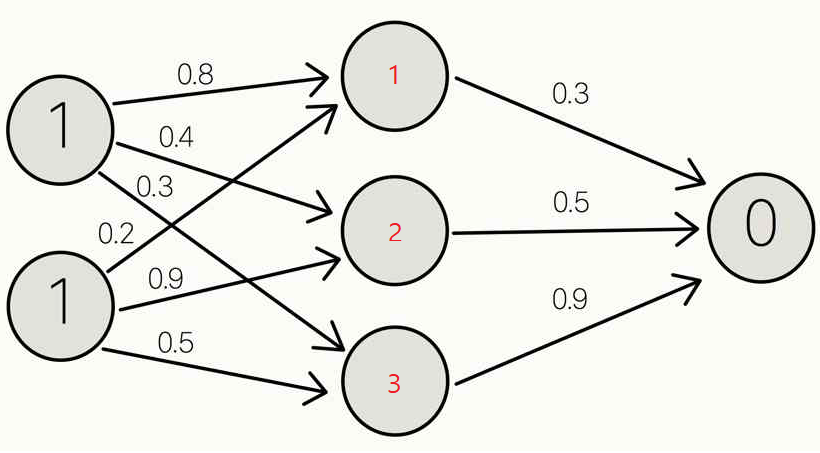


실행 결과 :



기울기 최저점을 찾은 것을 볼 수 있습니다.

@ 4번문제



1번 출력값 : 1 \* 0.8 + 1 \* 0.2 = 0.8 + 0.2 = 1, 시그모이드 통과시 : 0.73106

2번 출력값 : 1 \* 0.4 + 1 \* 0.9 = 0.4 + 0.9 = 1.3, 시그모이드 통과시 : 0.78583

3번 출력값 : 1 \* 0.3 + 1 \* 0.5 = 0.3 + 0.5 = 0.8, 시그모이드 통과시 : 0.68997

최종 예측값 : 0.73106 \* 0.3 + 0.78583 \* 0.5 + 0.68997 \* 0.9

= 0.219318 + 0.392915 + 0.620973 = 1.233206,

시그모이드 통과시 : 0.77438

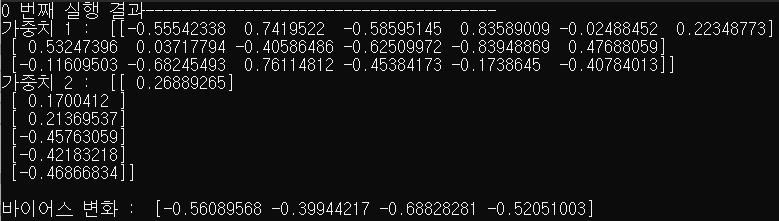
오차 : = = 0.2998321922

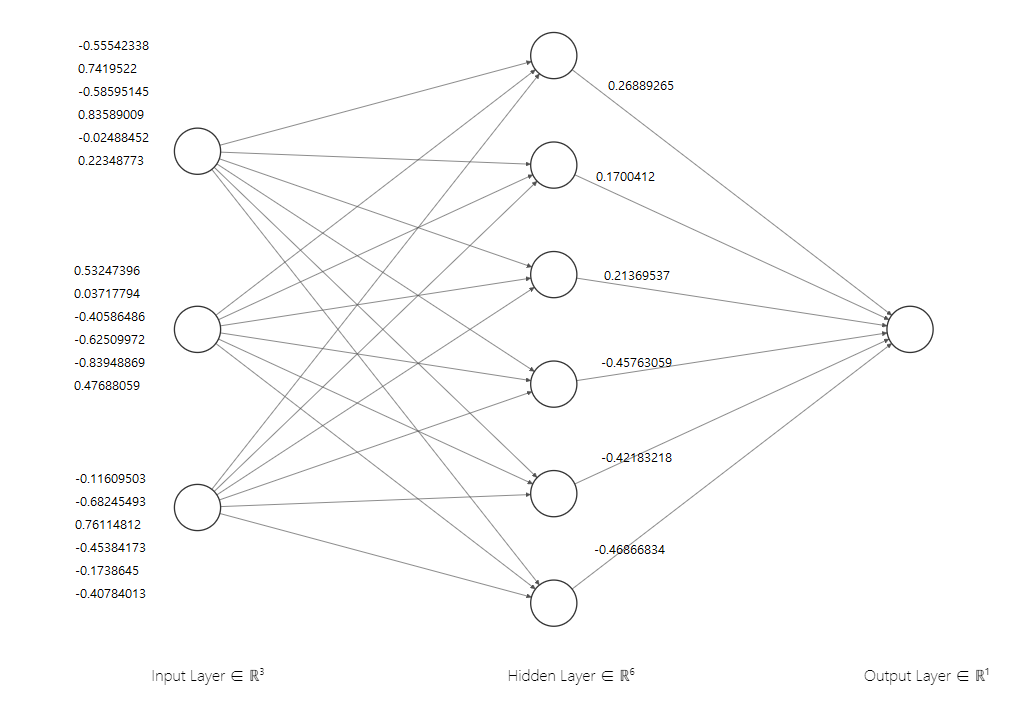
@ 5번문제

코드는 챕터 13의 mlp.py를 사용하였습니다.

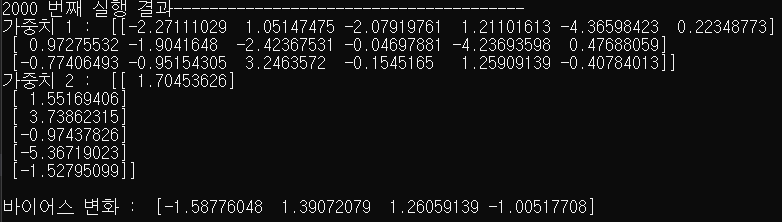
가중치의 변화량 표현은 1만번의 실행 중 매 2000번째 실행마다 표현하도록 하였습니다.

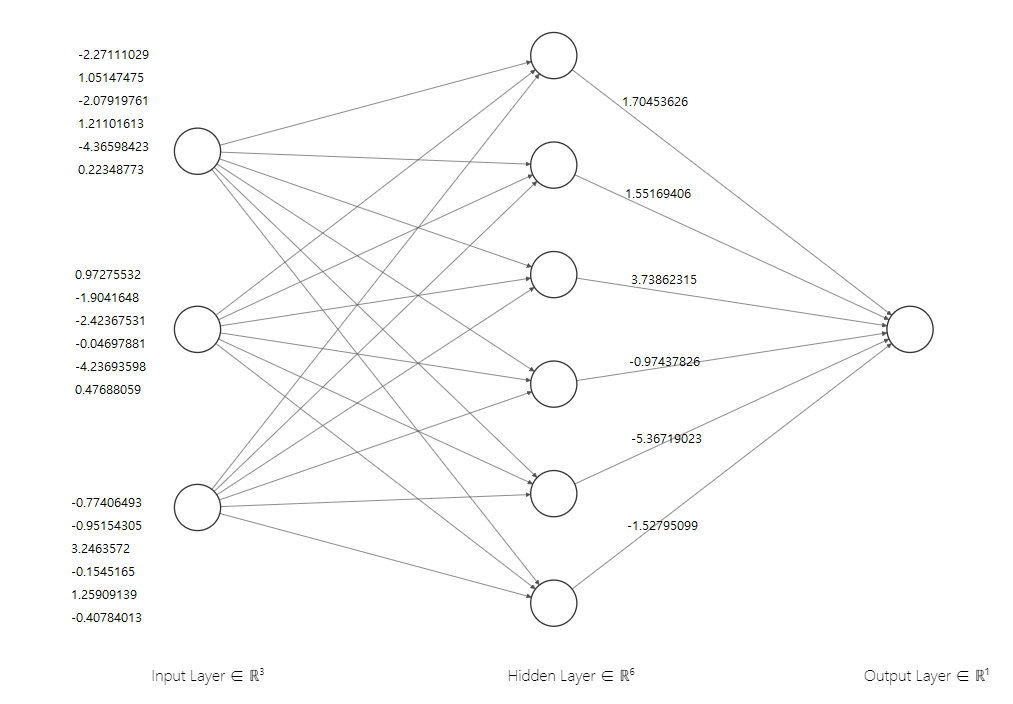
- 0번째(초기) 실행화면 -



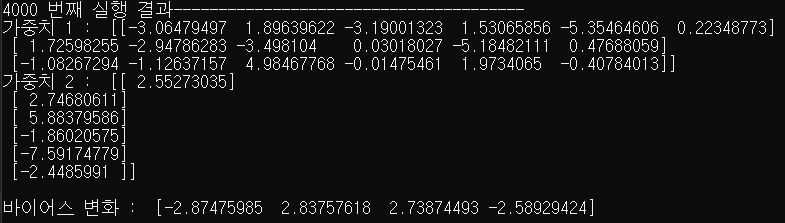


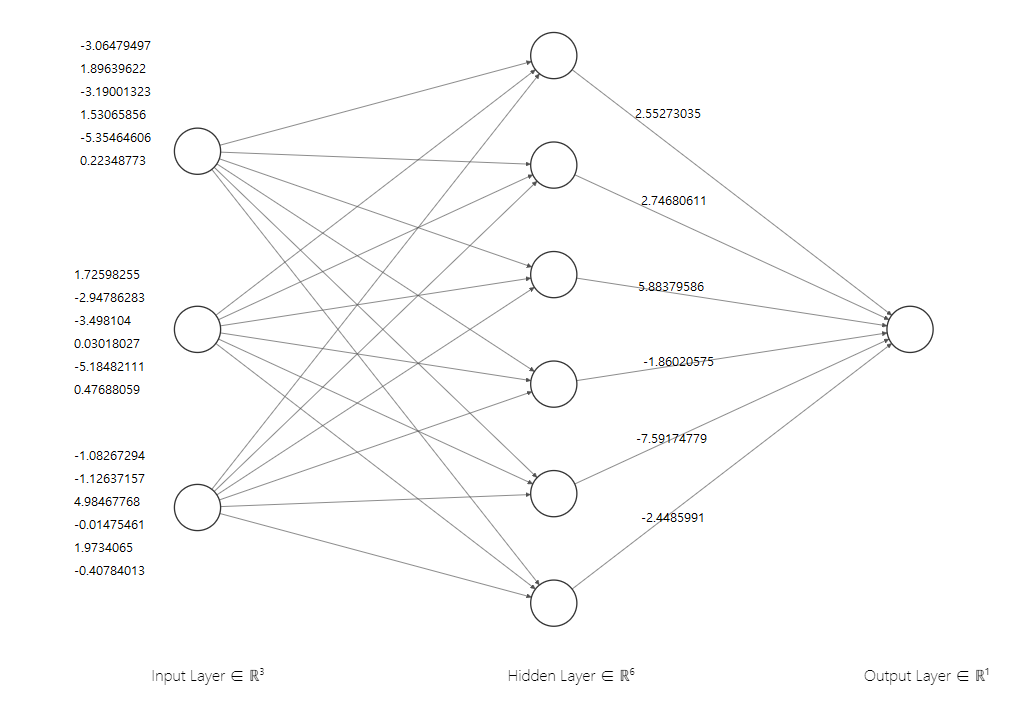
- 2000번째 실행 결과 –



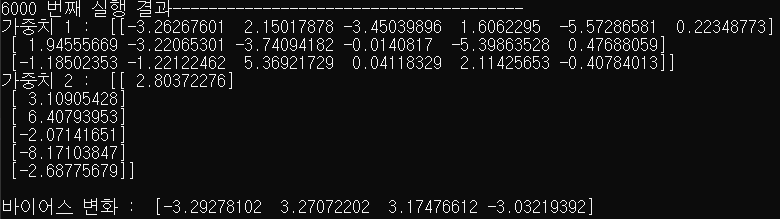


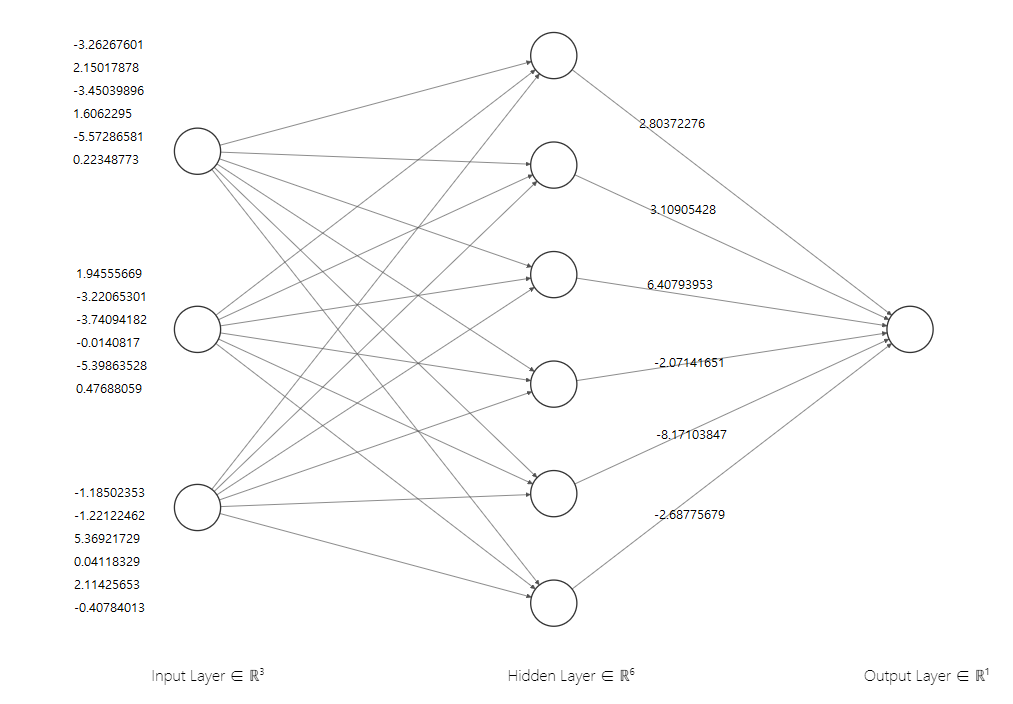
- 4000번째 실행 결과 –



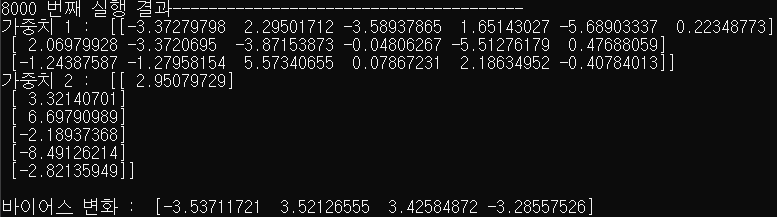


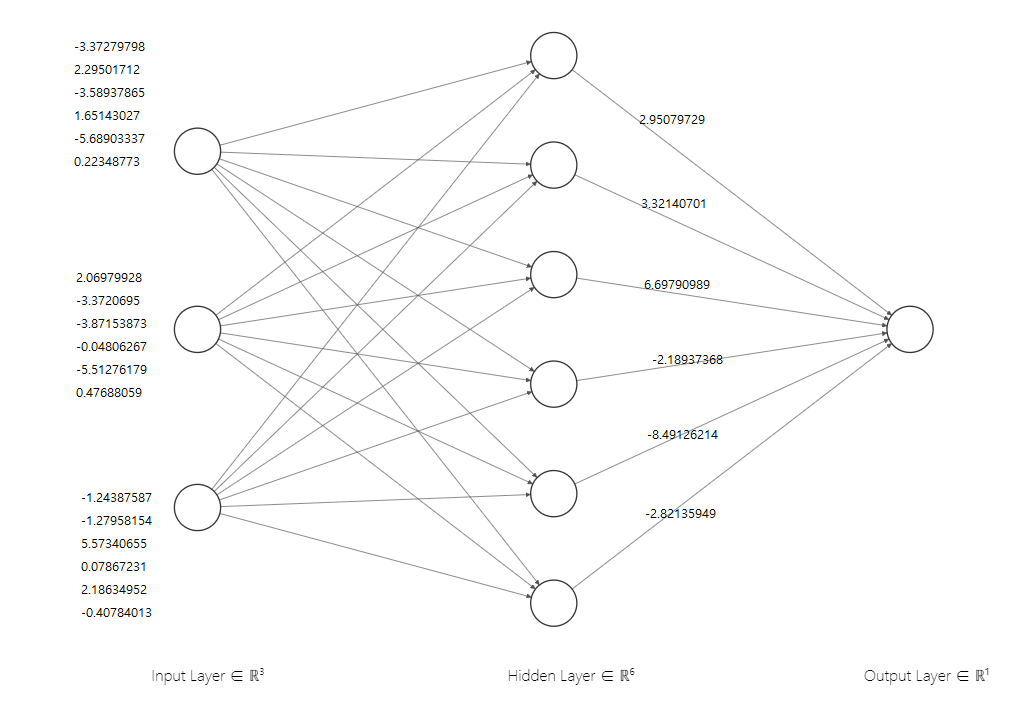
- 6000번째 실행 결과 –



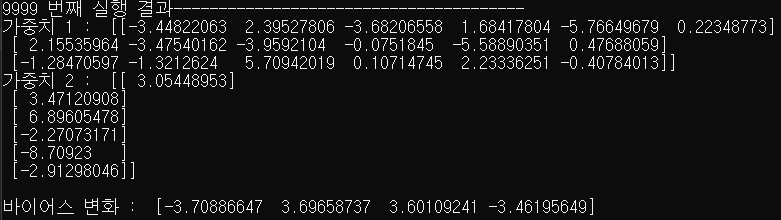


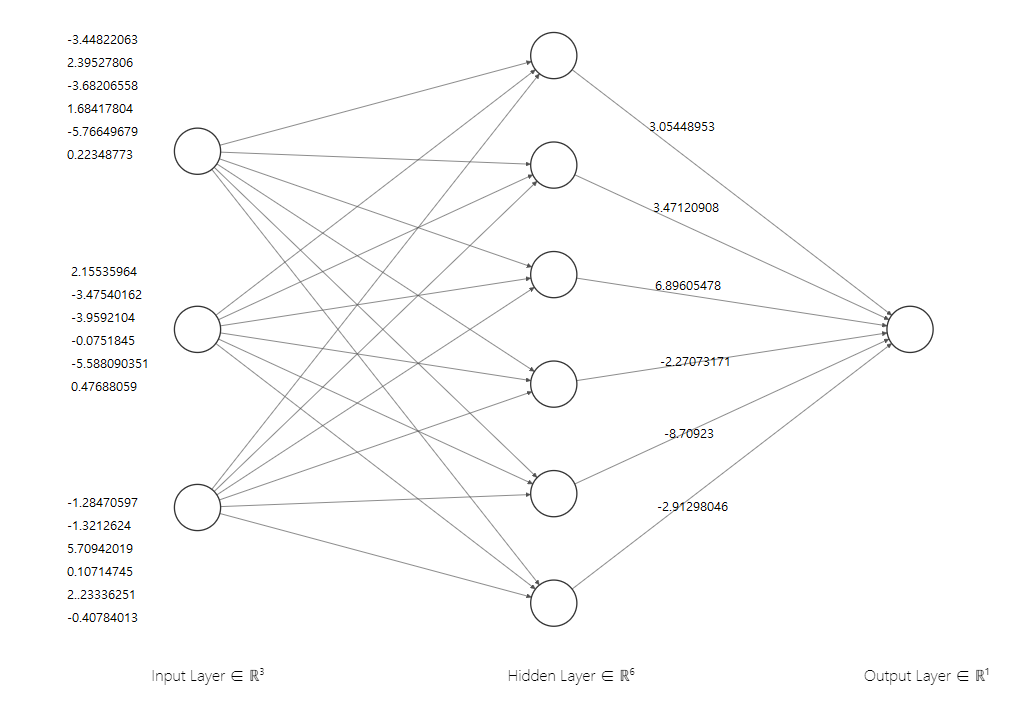
- 8000번째 수행 결과 –





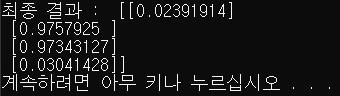
- 9999(마지막)번째 수행 결과 –





최종 결과 :

입력값 [0,0,1], [0,1,1], [1,0,1], [1,1,1] 에 대한 결과



0,1,1 과 1,0,1 의 or 결과에 대해 1에 근사한 값을 보여주고

이외에 값은 0에 근사한 값을 보여주는 것을 확인할 수 있습니다.

코드

# -\*- coding: cp949 -\*-

import numpy as np

# 시그모이드 함수

def actf(x):

    return 1/(1+np.exp(-x))

# 시그모이드 함수의 미분값

def actf\_deriv(x):

        return x\*(1-x)

# XOR 연산을 위한 4행\*2열의 입력 행렬

# 마지막 열은 바이어스를 나타낸다.

X = np.array([[0,0,1], [0,1,1], [1,0,1], [1,1,1]])

# XOR 연산을 위한 4행\*1열의 목표 행렬

y = np.array([[0], [1], [1], [0]])

np.random.seed(5)

inputs = 3  # 입력층의 노드 개수

hiddens = 6 # 은닉층의 노드 개수

outputs = 1 # 출력층의 노드 개수

# 가중치를 –1.0에서 1.0 사이의 난수로 초기화한다.

weight0 = 2\*np.random.random((inputs, hiddens))-1

weight1 = 2\*np.random.random((hiddens, outputs))-1

# 반복한다.

for i in range(10000):

    # 순방향 계산

    layer0 = X              # 입력을 layer0에 대입한다.

    net1 = np.dot(layer0, weight0)  # 행렬의 곱을 계산한다.

    layer1 = actf(net1)     # 활성화 함수를 적용한다.

    layer1[:,-1] = 1.0          #  마지막 열은 바이어스를 나타낸다. 1.0으로 만든다.

    net2 = np.dot(layer1, weight1)  # 행렬의 곱을 계산한다.

    layer2 = actf(net2)     # 활성화 함수를 적용한다.

    # 출력층에서의 오차를 계산한다.

    layer2\_error = layer2-y

    # 출력층에서의 델타값을 계산한다.

    layer2\_delta = layer2\_error\*actf\_deriv(layer2)

    # 은닉층에서의 오차를 계산한다.

    # 여기서 T는 행렬의 전치를 의미한다.

    # 역방향으로 오차를 전파할 때는 반대방향이므로 행렬이 전치되어야 한다.

    layer1\_error = np.dot(layer2\_delta, weight1.T)

    # 은닉층에서의 델타를 계산한다.

    layer1\_delta = layer1\_error\*actf\_deriv(layer1)

    # 은닉층->출력층을 연결하는 가중치를 수정한다.

    weight1 += -0.2\*np.dot(layer1.T, layer2\_delta)

    # 입력층->은닉층을 연결하는 가중치를 수정한다.

    weight0 += -0.2\*np.dot(layer0.T, layer1\_delta)

    #print("가중치 1 : " , weight0, ", 가중치 2 : " , weight1)

    if(i % 2000 == 0 or i == 9999):

        print(i,"번째 실행 결과---------------------------------------")

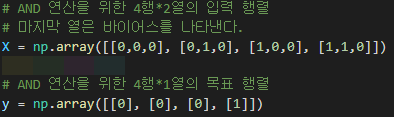
        print("가중치 1 : " , weight0, "\n가중치 2 : " , weight1, "\n")

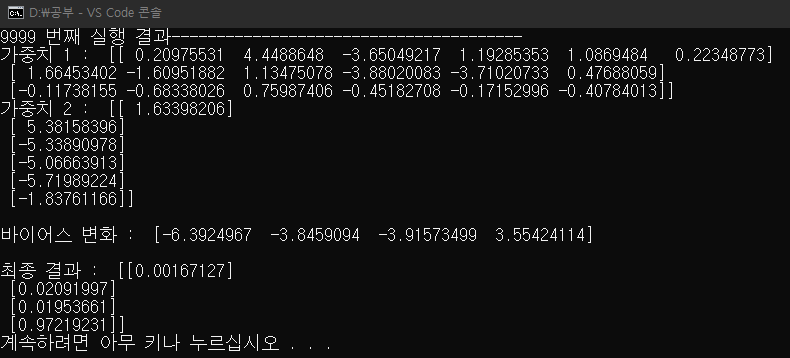
        print("바이어스 변화 : " , net2[:,-1], "\n")

print("최종 결과 : ", layer2)           # 현재 출력층의 값을 출력한다.

@ 6번문제

위의 5번 문제 코드에서 입력 값을 AND 조건에 맞게 수정 시켜준 후 실행해 보았습니다. 초기 바이어스 값은 필요 없을 것 같아 0으로 제시해주었습니다.

입력값 : 

실행 결과 : 

AND 조건에 해당한 마지막 열부분이 1에 수렴하는 것을 볼 수 있습니다.

@ 7번문제

코드 :

# -\*- coding: cp949 -\*-

import numpy as np

from sklearn.datasets import load\_iris

def actf(x):

    return 1/(1+np.exp(-x))

def actf\_deriv(x):

    return x\*(1-x)

iris = load\_iris()

X = np.array(iris.data[:, (0, 1)])

#세토사 출력값들을 이차원배열 형태로 변환

y = np.reshape(np.array((iris.target == 0).astype(np.int)), (150, 1))

print(X)

print(y)

np.random.seed(5)

inputs = 2  #기존코드에서 가중치는 제거되고 너비와 높이만을 이용

hiddens = 6

outputs = 1

weight0 = np.zeros((inputs, hiddens)) # 가중치 0으로 세트

weight1 = np.zeros((hiddens, outputs)) # 가중치 0으로 세트

for i in range(10000):

    layer0 = X

    net1 = np.dot(layer0, weight0)

    layer1 = actf(net1)

    net2 = np.dot(layer1, weight1)

    layer2 = actf(net2)

    layer2\_error = layer2-y

    layer2\_delta = layer2\_error\*actf\_deriv(layer2)

    layer1\_error = np.dot(layer2\_delta, weight1.T)

    layer1\_delta = layer1\_error\*actf\_deriv(layer1)

    weight1 += -0.2\*np.dot(layer1.T, layer2\_delta)

    weight0 += -0.2\*np.dot(layer0.T, layer1\_delta)

print(layer2)           # 현재 출력층의 값을 출력한다.

#전체 결과 출력

i = 0

while i < np.size(y):

    print("붓꽃 크기 : [", '{:0.1f} '.format(X[i][0]), '{:0.1f}'.format(X[i][1]), "], 품종이 세토사인가? :", bool(y[i]), ", 학습값 : ", layer2[i])

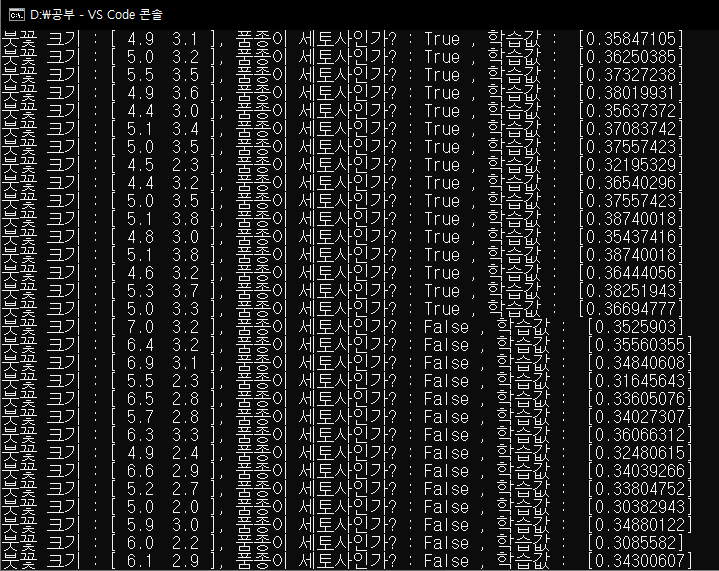
    i = i + 1

이전 MLP 예제를 기반으로 테스트 해보았습니다.

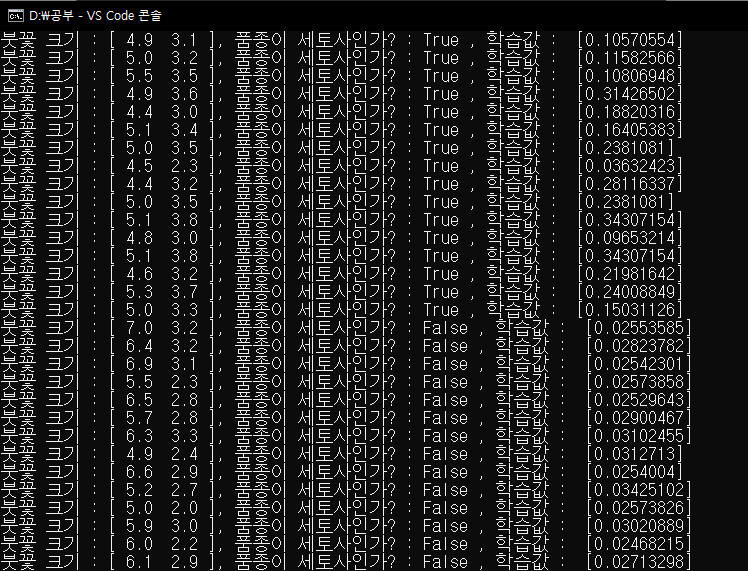
기존 MLP예제에선 각 입력값 마다 바이어스가 제공된 것과 달리 이번 붓꽃테스트에는 꽃의 너비와 크기만을 가지고 실행하기 때문에 입력값은 2로 정의하고, 바이어스 값은 전부 0으로 세트해두었습니다. 이후 반복량을 10번, 100번, 1000번, 10000번, 100000번 순으로 실행하여 변화값을 지켜보았습니다.

실행 결과 모습은 품종데이터 중 세토사인지 아닌지의 경계선부분에서 캡쳐하였습니다.

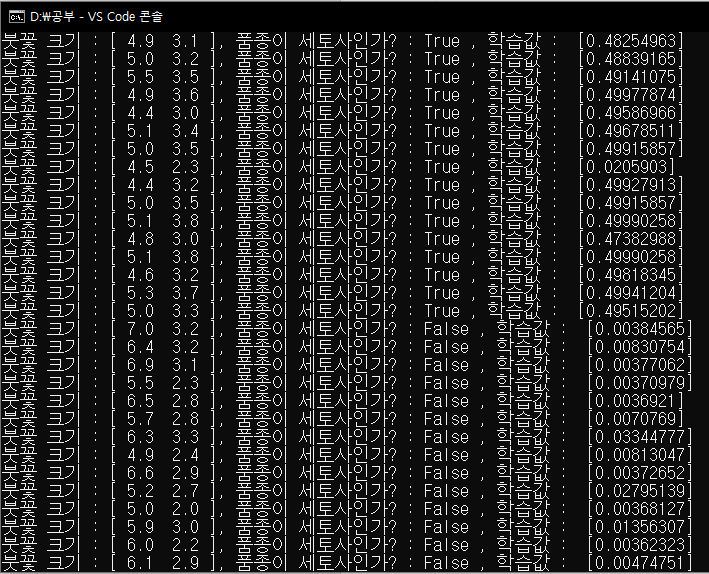
10번 반복 실행 결과



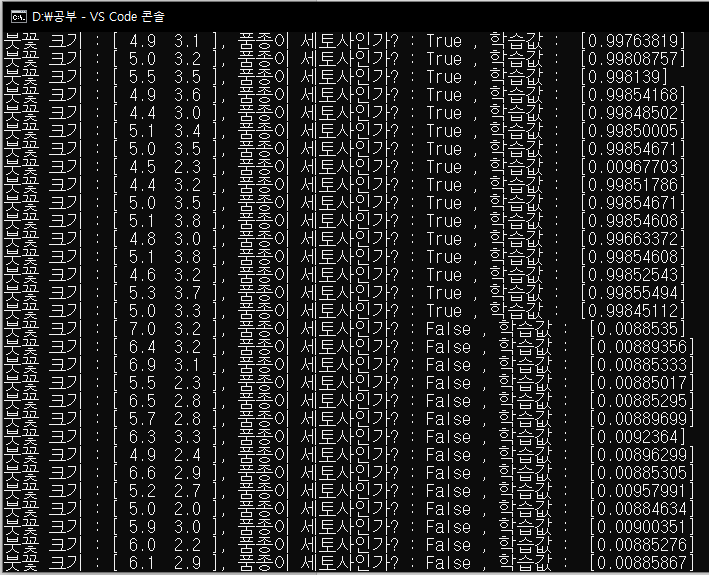
100번 반복 실행 결과



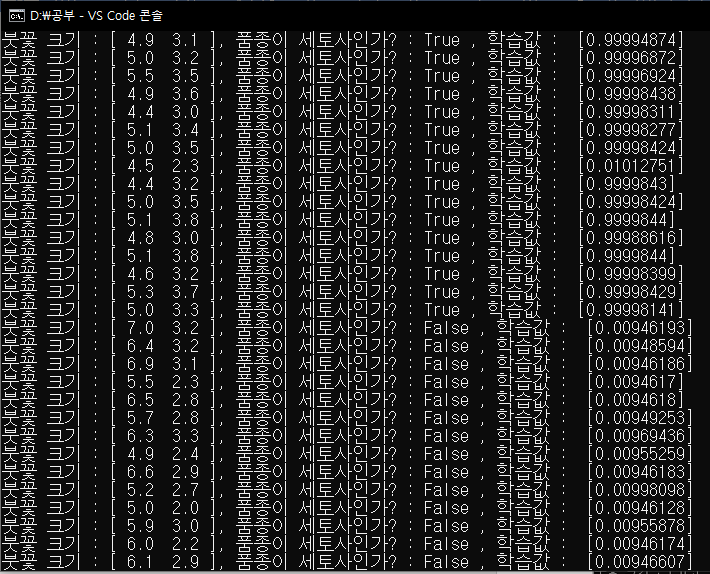
1000번 반복 실행 결과



10000번 반복 실행 결과



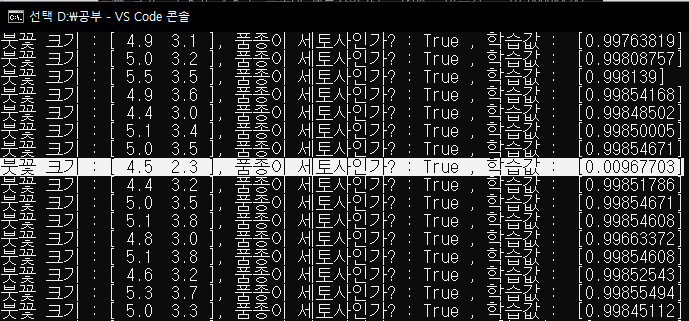
100000번 반복 실행 결과



결론 : 반복횟수가 증가함에 따라 학습값이 원하는 값에 도달하는 것을 볼 수 있습니다.

유의미한 데이터는 1만번 반복때부터 진행되는 것 같습니다.

하지만 세토사 품종중 원하는 학습값에 도달하지 못하는 데이터가 있었습니다.



기존 다른 세토사 품종들 대비 꽃밭침의 높이가 상대적으로 많이 짧은 케이스여서 그런지 학습이 잘 안되는 것 같습니다. 기존 세토사 비교 목록인 꽃잎의 너비, 높이까지 테스트해봐야 하는 케이스인듯 합니다.