HW#2
: Artificial Neural Network

제출일 : 2023-11-26
과목명 : 기계학습의 기초 및 응용
담당교수 : 이규행
전공 : 모바일 시스템 공학과
학번 : 32191097
이름 : 김준형

Introduction

이번 과제는 x, y값에 대한 데이터가 저장되어 있는 csv파일을 읽은 후 x값을 입력 층, y값을 출력 층으로 갖는 인공 신경망을 통해 선형 회귀를 구현하는 것이다.

인공 신경망의 레이어별 forwarding과정을 보다 편리하게 구현하기 위해 numpy 라이브러리를 통해 행렬 연산을 수행하였으며 validation error 및 모델의 결과를 출력하기 위해 matplotlib 라이브러리를 사용했다.

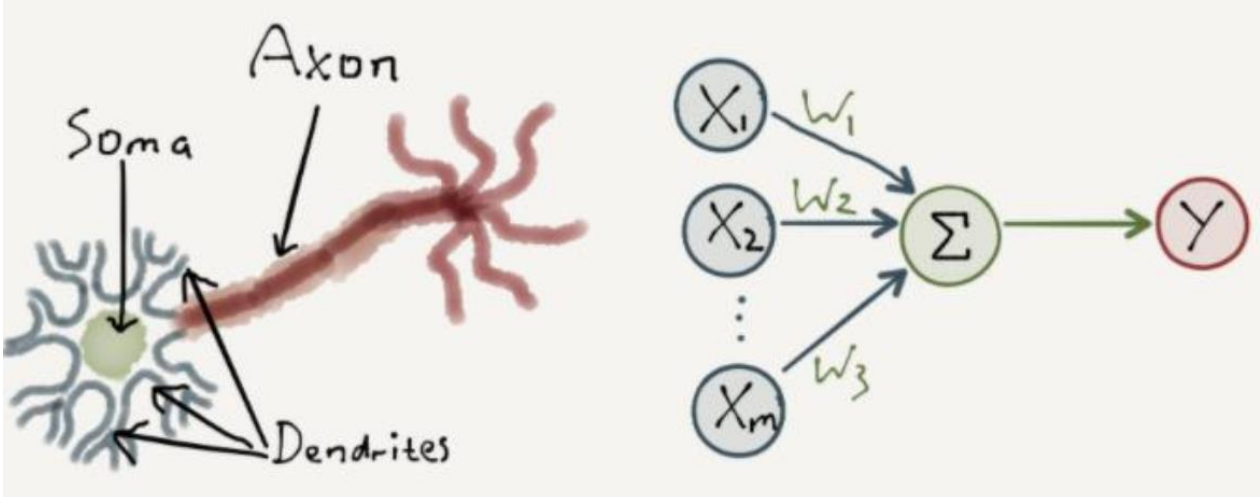
인공 신경망의 모델을 구현하기 위한 방법으로 각 레이어별 뉴런의 개수가 적힌 배열과 각 레이어별 적용할 활성화 함수가 적힌 배열을 객체 생성 인자로 받으며 해당 인자를 통해 각 레이어별 입력이 저장될z_s, 활성화 함수의 결과가 저장될 a_s, 가중치 및 편차가 저장될 w_s, b_s 인스턴스가 생성된다. 모델이 생성된 후 x,y 데이터를 인자로 training 메소드를 실행하면 일부 데이터는 validation error 검사를 위해 분리하고 남은 training 데이터를 지정된 batch size만큼 분해해 지정된 epoch횟수만큼 반복 학습한다.

학습이 1회 진행되는 과정은 다음과 같다. 우선 주어진 batch를 forwarding하여 예측된 결과값을 도출한다. 그 후 해당 결과값과 원본 결과값의 오차를 통해 역전파를 한다. 일정 epoch마다 진행률 및 training error, validation error를 출력하도록 하여 의도와 적합하게 모델이 학습되고 있는지 확인할 수 있도록 했으며 모델이 전부 학습된 후에는 training error, validation error의 변화와 전체 csv데이터를 통한 모델의 예측 결과를 그래프로 출력하도록 했다.

Background

인공 신경망이란 인간의 뇌 속 뉴런의 작용 방식을 모방한 인공지능의 일종으로 각 뉴런에서 다음 뉴런으로 정보가 전달되는 과정을 통해 특정한 결과값을 도출해 낼 수 있도록 뉴런간의 관계를 정의, 학습하는 알고리즘이다.

인공 신경망에서 뉴런에서 다음 뉴런으로 정보가 전달되는 과정은 다음과 같이 표현된다.



위 사진과 같이 다음 뉴런에 전달받은 모든 입력이 정해진 가중치만큼 다음 뉴런에 반영되는데 모델의 구조(레이어들의 구조)와 해당 가중치가 결과를 결정하게 된다.

모델의 구조는 사용자가 초기에 결정하게 된다. 그렇기에 인공 신경망을 학습시키기 위해서는 가중치를 수정시킬 필요가 있는데 이를 역전파를 통해 해결한다. 역전파란 현재 모델을 통한 예측 결과와 목표 결과 사이의 오차를 통해 가중치를 뒤에서부터 앞으로 수정해가는 과정으로 오차를 통해 점진적으로 개선해가는 경사 하강법을 사용하며 오차의 누적을 위해 chain rule이 매우 중요하게 작용한다.

Chain rule이란 합성함수의 미분에 대한 공식으로 합성 함수의 미분은 각 함수를 분해한 결과의 미분의 곱으로 계산할 수 있다는 공식이다.

The formula $f(x) = f(g(x))$ means
that f is a function of g and g is a function of x.

We can calculate $\frac{\partial f}{\partial x}$ indirectly as follows:

$$\frac{\partial f}{\partial x} = \frac{\partial f(g)}{\partial g} \cdot \frac{\partial g(x)}{\partial x}$$

이 공식은 역전파를 진행하는 과정 중 최종 예측의 결과가 현재 수정하고자 하는 가중치에 미치는 정도를 구하기 위해 사용된다. 모델의 cost function이 계산되기 바로 직전의 가중치는 바로 이전의 활성화 함수 결과값과 결합되어 상대적으로 직접적으로 오차에 영향을 미치지만 그 이전의 가중치들은 이후 레이어의 각 뉴런에 분산되기에 영향을 계산하기 힘들게 된다. 하지만 이를 chain rule을 통해 계산한다면 미분의 곱을 통해 편리하게 가중치가 오차에 미친 영향을 확인할 수 있게 되며 이를 통해 경사 하강법을 보다 쉽게 구현할 수 있게 된다. 자세한 공식은 아래와 같다.

$$J = \frac{1}{2}(a^{(L)} - y)^2$$

$$\frac{\partial J}{\partial a^{(L)}} = a^{(L)} - y$$

$$\frac{\partial J}{\partial z^{(L)}} = \frac{\partial J}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} = (a^{(L)} - y)\sigma'(z^{(L)})$$

$$\frac{\partial J}{\partial W^{(L-1)}} = \frac{\partial J}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial W^{(L-1)}} = (a^{(L)} - y)\sigma'(z^{(L)})a^{(L-1)}$$

(J: cost function, a: activation function, z: layer input, W: weight)

Implementation

구현된 코드는 크게 3가지 단계로 나누어진다.

첫번째 단계는 데이터 전처리 및 라이브러리 import 과정이다.

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

f = open('hw2_data.csv', 'r', encoding='utf-8') #파일 디렉토리 변경시 수정 필요
data_f = csv.reader(f)

xs = []
ys = []

for i, line in enumerate(data_f):
    if i == 0:
        continue
    xs.append(float(line[0]))
    ys.append(float(line[1]))

xs = np.array(xs)
ys = np.array(ys)
f.close()
```

1차 과제와 동일하게 csv, matplotlib, numpy 라이브러리를 import했으며 동일한 과정을 거쳐 ndarray객체로 x와 y값들을 변환해 두었다.

두번째 단계는 인공 신경망 class 구현 단계이다.

```
class FCNN:
    def __init__(self, layers, active_funcs, epochs, learning_rate = 0.00001,
        self.layers = layers
        self.b_s = []
        self.z_s = [] # n*1 n*10 n*20 ...
        self.a_s = [] # n*1 n*10 n*20 ...
        self.w_s = [] # 1*10 10*20 20*40 ...
        self.active_funcs = active_funcs
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.validation_err = []
        self.training_err = []
```

인공 신경망 객체의 인스턴스는 모델의 레이어 구조 및 활성화함수 구조가 담긴 layers, active_funcs와 학습에 필요한 정보인 z_s, a_s, w_s, b_s 및 epochs, learning_rate, 마지막으로 validation err 검사를 위한 validation_err, training_err가 있다.

인공 신경망 객체를 생성한 후 학습을 위한 메소드는 다음과 같다.

```
37      def training(self, xs, ys):
38          x_train, x_val = xs[:int((len(xs)/10)*8)], xs[int((len(xs)/10)*8):]
39          y_train, y_val = ys[:int((len(ys)/10)*8)], ys[int((len(ys)/10)*8):]
40
41
42          self.initialize_weight()
43          cost_function = self.get_cost_function()
44          count = 0
45          for epoch in range(self.epochs):
46              for i in range(0, len(x_train), int(len(x_train)*0.2)):
47                  x_batch = x_train[i:i+int(len(x_train)*0.2)]
48                  y_batch = y_train[i:i+int(len(y_train)*0.2)]
49                  self.forwarding(x_batch)
50                  loss = self.a_s[-1] - y_batch
51                  self.back_propagation(loss)
52              self.forwarding(x_train)
53              self.training_err.append(np.mean(cost_function(y_train, self.a_s[-1])))
54              self.forwarding(x_val)
55              self.validation_err.append(np.mean(cost_function(y_val, self.a_s[-1])))
56              if epoch % (self.epochs//100) == 0:
57                  print(self.training_err[epoch], self.validation_err[epoch])
58                  print(count,"percent done")
59                  count+=1
```

1차적으로 입력된 데이터를 학습용 데이터와 검증용 데이터로 분리하며 initialize_weight 메소드를 통해 z_s, a_s, w_s, b_s 인스턴스를 초기화한다. 그후 학습용 데이터를 배치로 나누어 minibatch GD를 수행하는데 한번의 epoch이 종료된 후에는 학습용 데이터와 검증용 데이터의 cost function 결과값을 저장한다. 또한 전체 과정을 100등분하여 진행률과 학습 에러, 검증 에러를 각기 출력한다.

```
    def forwarding(self, xs):
        self.z_s[0] = xs
        for i in range(len(self.layers)):
            self.a_s[i] = self.get_activ(self.active_funcs[i])(self.z_s[i])
            if i < len(self.layers) - 1:
                tmp = np.matmul(self.a_s[i], self.w_s[i])
                tmp += self.b_s[i]

                self.z_s[i+1] = tmp
        return

    def back_propagation(self, grad):
        for i in range(len(self.layers)-2, -1, -1):
            grad = grad * self.get_grad(self.active_funcs[i+1])(self.a_s[i+1])
            self.w_s[i] = self.w_s[i] - self.learning_rate * np.matmul(self.z_s[i].T, grad)
            self.b_s[i] = self.b_s[i] - self.learning_rate * grad.sum(axis=0)
            grad = np.matmul(grad, self.w_s[i].T)
        return
```

forwarding과 back propagation은 다음과 같이 구현되어 있다.

forwarding은 단순히 활성화 함수를 통과시킨 후 가중치와 행렬 곱 연산을 수행하고 편차를 더해주는 과정의 반복으로 이루어져 있지만 back propagation에는 Background에서 설명한 핵심인 chain rule이 구현되어 있다. 일차적으로 grad값을 해당 레이어의 활성화 함수의 미분값과 곱해주어 활성화 함수에 대한 chain rule 과정을 끝낸 후 가중치와 편차를 학습시켜주는데 이 때 편차는 전부 1인 값이 가중치를 통해 변경되는 것 이므로 복잡한 행렬 곱 연산 대신 sum 연산을 통해 해결했다. 이후 다음 가중치 단계를 위해 grad값을 수정해 주는데 이전 레이어의 a_s에 대한 chain rule 결과는 grad와 가중치의 행렬 곱이므로 위와 같이 수정한 후 반복 수행한다.

```
111     def get_activ(self, func):
112         if func == 'Relu':
113             return self.Relu
114         elif func == 'Leaky_Relu':
115             return self.Leaky_Relu
116         elif func == 'Sigmoid':
117             return self.Sigmoid
118         else:
119             return self.Identity
120 
121     def get_grad(self, func):
122         def Relu(x):
123             return np.heaviside(x, 0)
124         def Leaky_Relu(x):
125             return np.heaviside(x>0, 0.2)
126         def Sigmoid(x):
127             return self.Sigmoid(x)*(1-self.Sigmoid(x))
128         def Identity(x):
129             return 1
130         if func == 'Relu':
131             return Relu
132         elif func == 'Leaky_Relu':
133             return Leaky_Relu
134         elif func == 'Sigmoid':
135             return Sigmoid
136         else:
137             return Identity
138     def Relu(self, x):
139         return np.maximum(x, 0)
140     def Leaky_Relu(self, x):
141         return np.maximum(0.2*x, x)
142     def Sigmoid(self, x):
143         return np.where(x>0, 1/(1+np.exp(-1*x)), np.exp(x)/(1+np.exp(x)))
144     def Identity(self, x):
145         return x
```

다음은 활성화 함수 및 각 활성화 함수의 미분 값들이다. 이때 주의해야 할 사항으로는 sigmoid 활성화 함수를 사용할 시 x값이 -1000과 같이 작은 음수가 들어오면 e^1000과 같이 큰 값을 연산하게 되고 오버플로우 에러를 발생시켜 INF와의 연산에 의해 가중치들이 NAN으로 변하게 된다. 그렇기에 x의 범위에 따라 x가 음수일 경우 분모와 분자에 e^x을 곱해주어 위와 같이 연산을 한다면 오버플로우 에러를 방지할 수 있다.

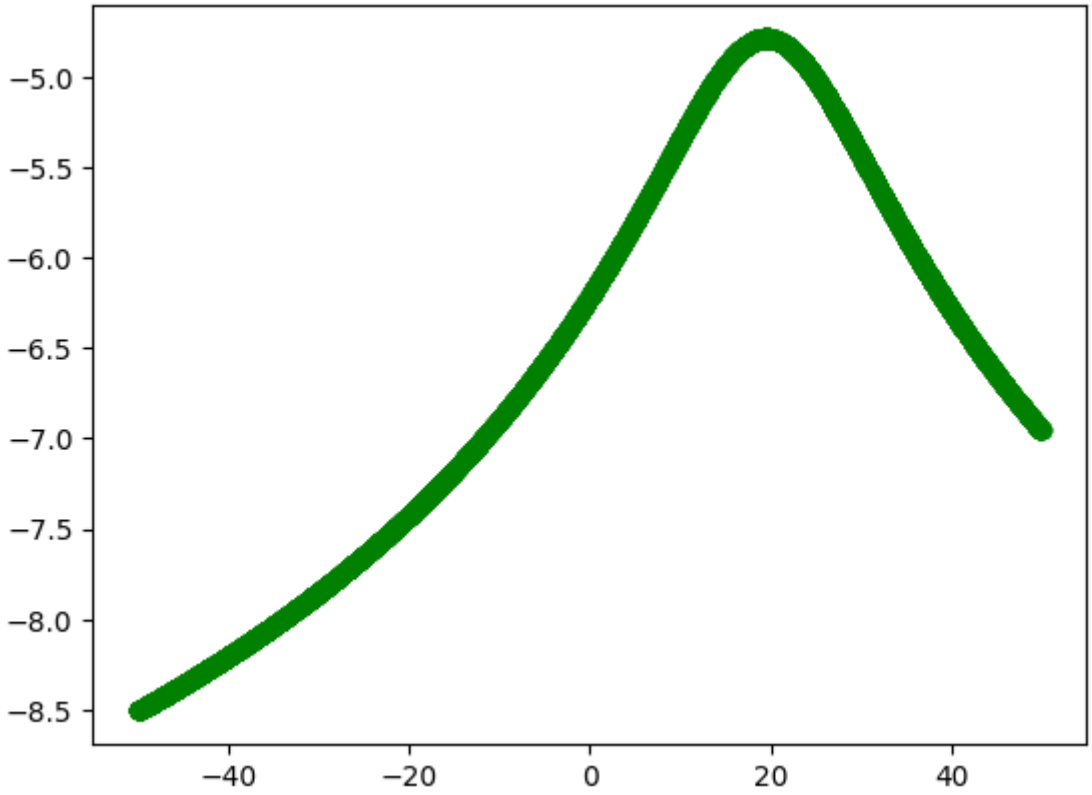
마지막 단계는 모델 생성, 학습 및 예측이다.

```
148    nn = FCNN([1, 4, 4, 1], ["Identity", "Leaky_Relu","Leaky_Relu", "Identity"], 10000000, 0.5*(10**-7))
149
150    xs = np.c_[xs]
151    ys = np.c_[ys]
152
153    nn.training(xs,ys)
154    nn.plot_validerr()
155    nn.predict_and_plot(xs,ys)
```

위와 같이 인공 신경망 객체를 생성하여 사전에 저장해둔 csv파일의 x, y값들을 입력해 학습시킨 후 필요한 결과값들을 그래프로 출력한다.

Result

결과에 앞서 모델 설계를 설명하자면 모델 레이어는 [1, 4, 4, 1], hidden layer의 활성화 함수는 전부 Leaky_Relu로 구성하였다. 그에 대한 이유는 다음과 같다. 우선 csv파일에 저장된 x값과 y값들에 대한 그래프는 다음과 같다.



해당 그래프는 3~4차 방정식으로 표현될 수 있을 것이라 판단했으며, 다항식으로 표현 가능한 모델은 복잡도가 크지 않아도 될 것이라 판단했다. 이를 검증하기 위해 [1, 4, 1], [1, 20, 1]과 같이 hidden layer가 1개인 모델을 학습시켜 보았지만 해당 모델은 curve fitting이 적절히 이루어지지 못하고 각진 모양으로 수렴했다. 이후 hidden layer의 개수를 늘려 [1, 4, 4, 1], [1, 8, 8, 1], [1, 20, 20, 1]과 같이 뉴런의 개수를 늘려가며 검증했지만 주어진 데이터가 단순하여 [1, 4, 4, 1]과 같은 모델로도 mse가 0.0006에 수렴하며 적절하게 학습되었다. 이 외에도 hidden layer의 개수를 더욱더 늘려 복잡도를 늘려보았지만 오히려 local minimum에 빠져 학습이 정상적으로 이루어지지 않는 모습 또한 보였다. 활성화 함수는 모델을 선형 함수의 합으로 표현하고자 하는 의미로 Relu를 택했지만 0이하의 값에 대하여 전부 0으로 설정하는 Relu의 특성상 정보가 지나치게 삭제된다고 판단하여 Leaky_Relu로 설정하게 되었다.

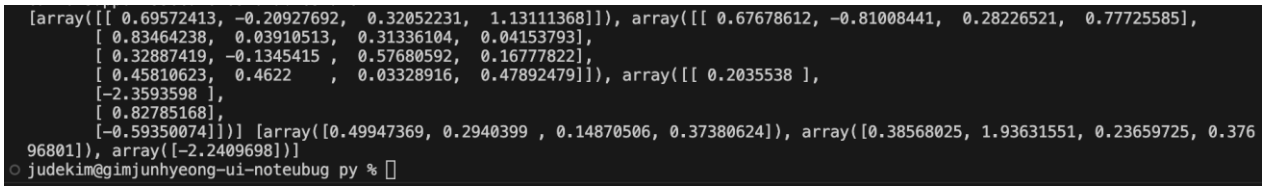
결과는 다음과 같다.

Setting:

Layer: [1, 4, 4, 1], activation function: Leaky_Relu
epoch: 1000000, learning_rate: 0.5*(10^-7)

Result:

weight and bias:



weight:

w1: [0.6957413, -0.20927692, 0.32052231, 1.13111368]

w2: [[0.67678612, -0.81008441, 0.28226521, 0.77725585],
[0.83464238, 0.03910513, 0.31336104, 0.04153793],
[0.32887419, -0.1345415, 0.57680592, 0.16777822],
[0.45810623, 0.4622, 0.03328916, 0.47892479]]

w3: [[0.2035538],
[-2.3593598],
[0.82785168],
[-0.59350074]]

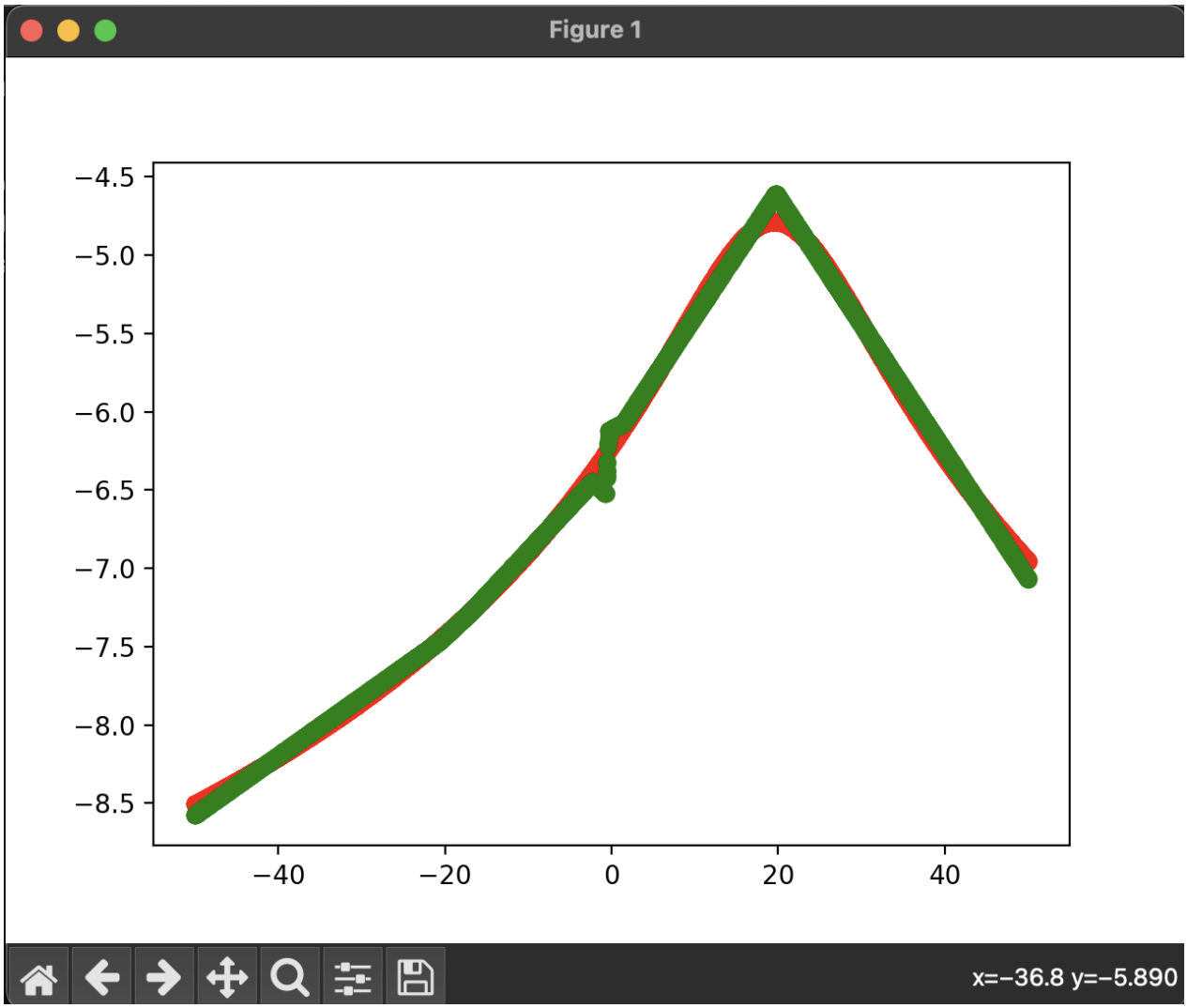
bias:

b1: [0.49947369, 0.2940399, 0.14870506, 0.37380624]

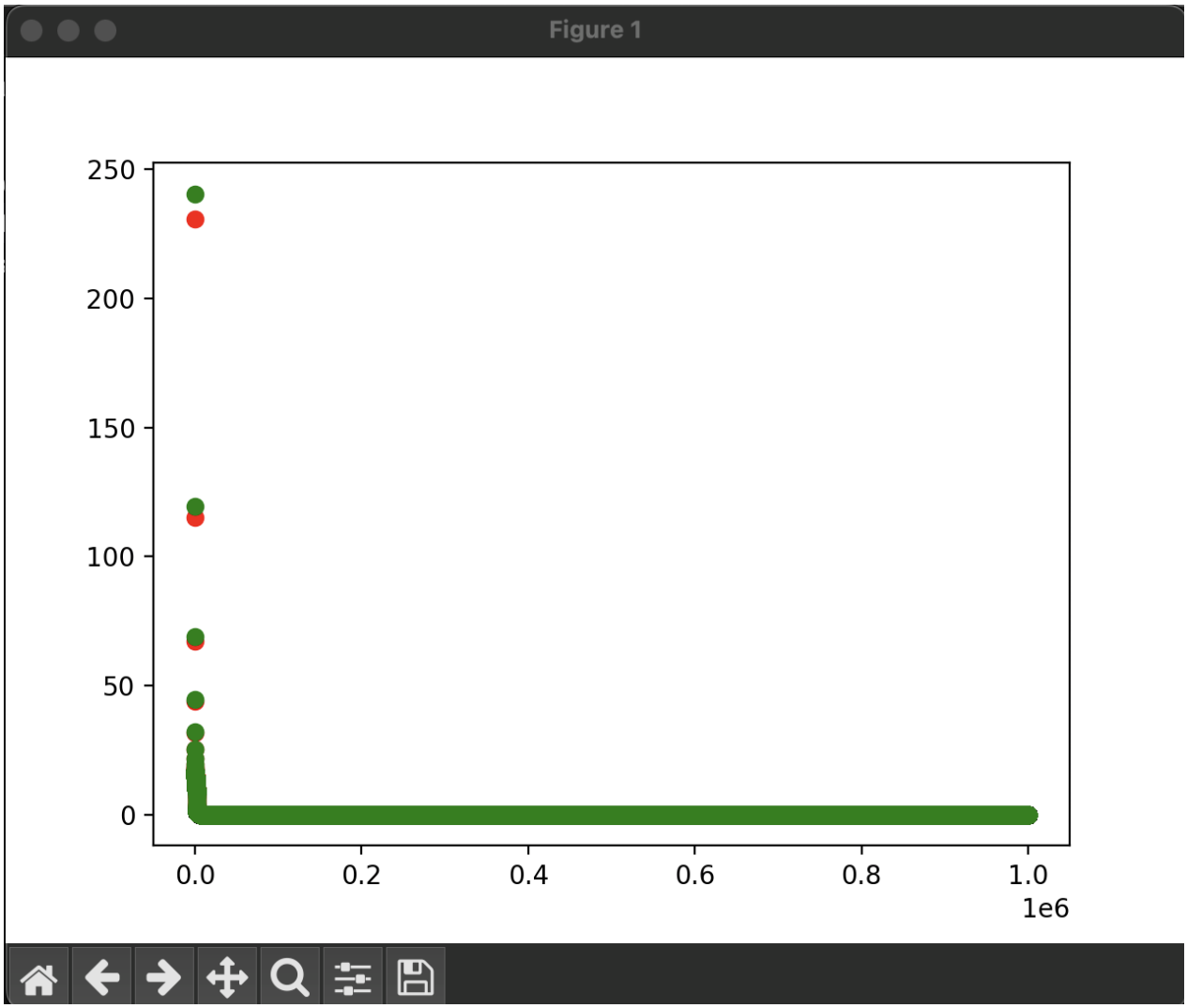
b2: [0.38568025, 1.93631551, 0.23659725, 0.3766801]

b3: [-2.2409698]

model feature:



validation error & training error

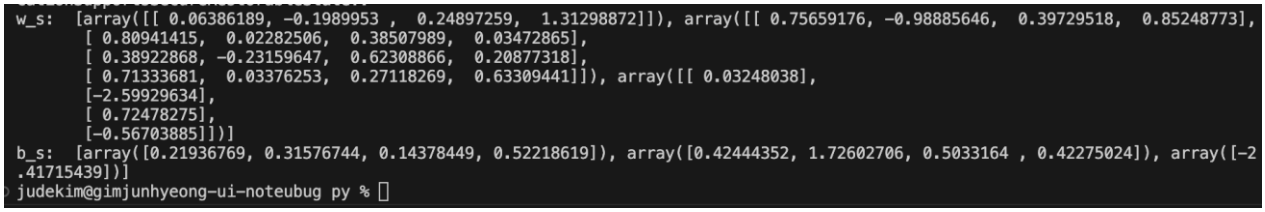


Setting:

Layer: [1, 4, 4, 1], activation function: Leaky_Relu
epoch: 5000000, learning_rate: 0.5*(10^-7)

Result:

weight and bias:



weight:

w1: [0.06386189, -0.1989953, 0.24897259, 1.31298872]

w2: [[0.75659176, -0.98885646, 0.39729518, 0.85248773],
[0.80941415, 0.02282506, 0.38507989, 0.03472865],
[0.38922868, -0.23159647, 0.62308866, 0.20877318],
[0.71333681, 0.03376253, 0.27118269, 0.63309441]]

w3: [[0.03248038],
[-2.59929634],
[0.72478275],
[-0.56703885]]

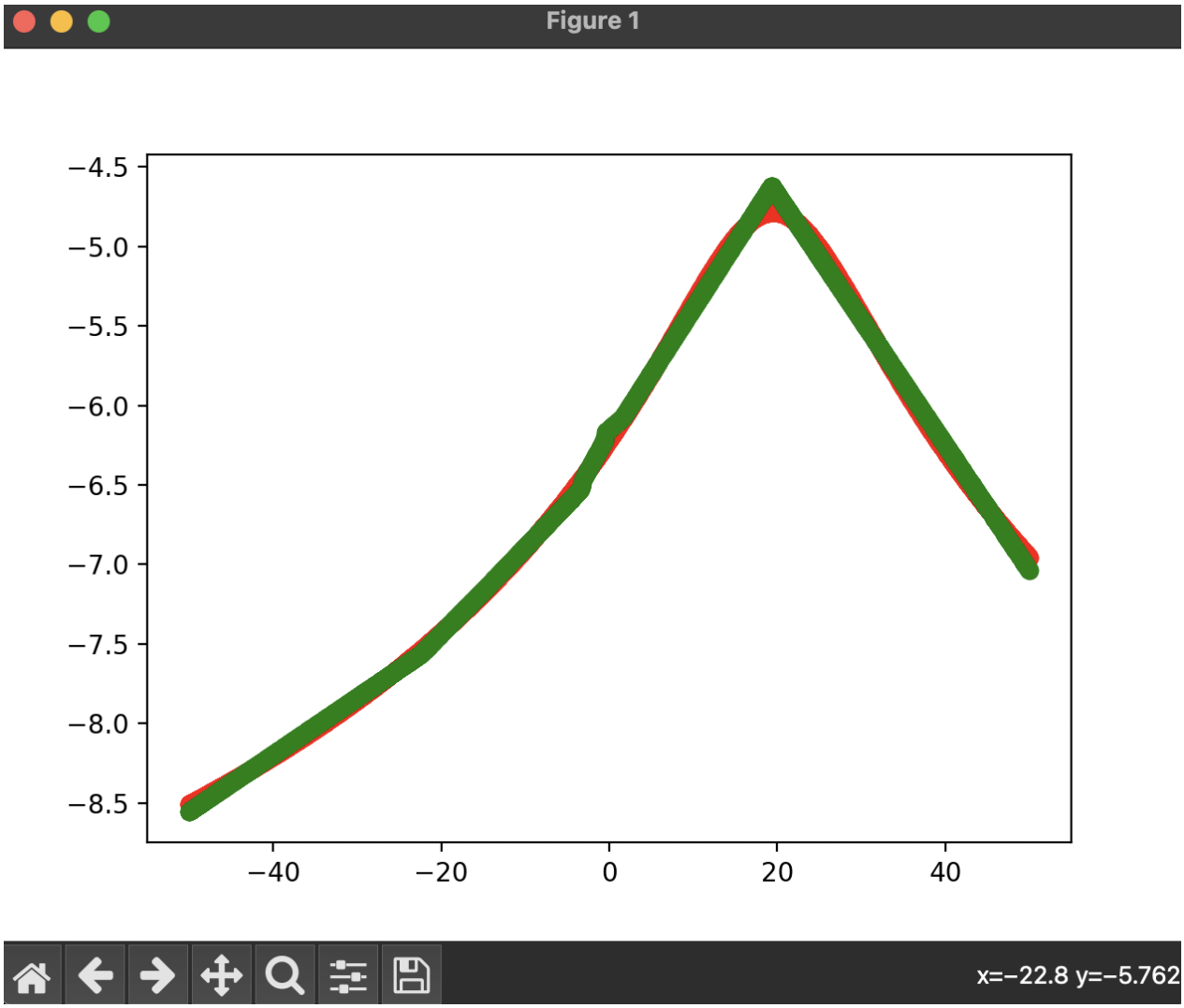
bias:

b1: [0.21936769, 0.31576744, 0.14378449, 0.52218619]

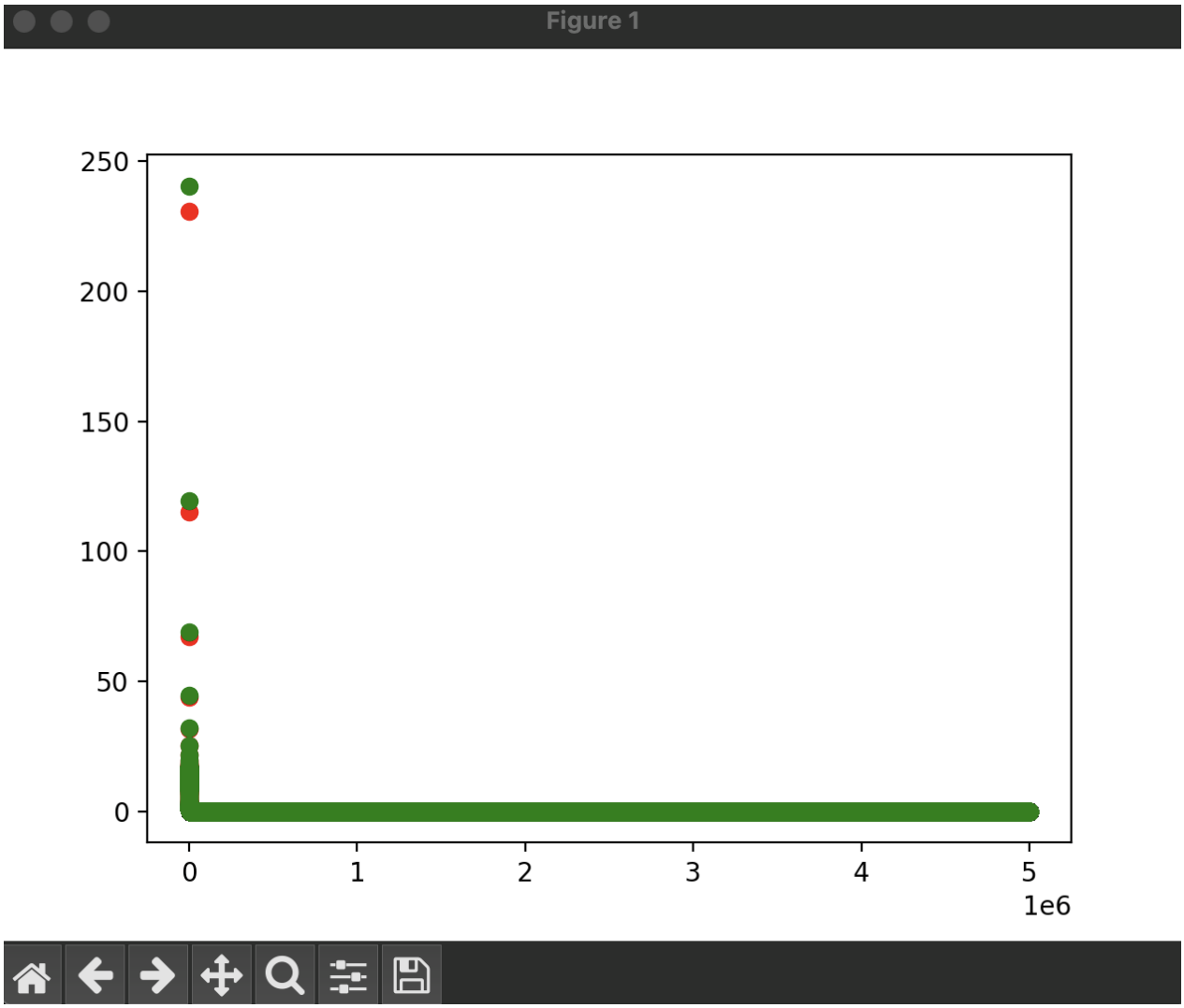
b2: [0.42444352, 1.72602706, 0.5033164, 0.42275024]

b3: [-2.41715439]

model feature:



validation error & training error



Conclusion

인공 신경망은 사람 뇌의 뉴런을 모방한 인공지능으로 가중치를 반영한 값의 전달의 특징에 의해 행렬 연산으로 구현할 수 있다. 인공 신경망의 모델은 레이어 계층 및 활성화 함수, 경사 하강법 진행 시 정규화 등으로 이루어져 있는데 해당 과제에서는 정규화는 반영하지 않았으며 hidden layer 2개 및 Leaky_Relu로 구성된 단순한 모델으로 구현했다. 모델이 단순함에 따라 local minimum으로 빠질 위험이 적었으며 epoch을 늘려 적절히 학습된 모델의 validation error는 0.0006에 수렴했다.

이 과제를 수행하기 전 개인적인 흥미에 의해 인공 신경망을 공부한 적이 있었다. 그 경험으로 인해 모델을 생성하기에 앞서 간단한 예측을 통해 적절한 모델이 무엇일지 고민해 보았고 지나친 복잡도는 학습 효율을 해칠 수 있다는 점을 다시 한번 느끼게 되었다. 또한 sigmoid를 사용하지 않고 Leaky_Relu를 통해 학습을 진행했기에 학습률이 조금만 커져도 오버플로우 에러가 자주 발생하여 모델 학습 전 데이터를 스케일링 하는 것 또한 염두에 두었지만 초기 데이터와 비교하는 과정이 불편할 듯하여 학습률을 줄이고 epoch을 늘리는 방향으로 진행했다.