Simple Implementation of a Function Approximator as a Multiple Layer Perceptron

: Second Assignment, Basic and Application of Machine Learning

Cha, Ho hyeon

(Department of Mobile System Engineering, Dankook University)

**Abstract**: MLP is the conceptual structure, which consists of perceptron and that of relationship. This logical structure can catch complicated non-linear relationships of unformatted data, even conventional machine learning method cannot grab the features without preprocessing raw data. The perceptron and that of the weight consist of a layer, then each layer is logically connected with adjacent layers. In logically, even MLP can represent as a graph, the actual implementation is grounded on matrices and vector operation. To train a MLP network, back propagation is applied to fit the prediction with updating perceptron’s weight. The chain rule is the main key of the error propagation node by node.

1. **Introduction of the background**

Perceptron imitates Neuron as a model and an algorithm. Perceptron consists of input(x\_i), Summation input(z), and activity function(a).

원, 도표, 그림, 라인이(가) 표시된 사진

자동 생성된 설명.

Figure 1.1 Structure of Perceptron

For each input, there is a corresponding weight, like the linear regression in the previous polynomial model, acts as a parameter of the model. Adding every multiply of input and weight at Z, the corresponding value is input to the activation function. The activation function indicates how active the node is, and to obtain an appropriate activation function, the following rules must be followed. # Need to adding Reference

1. It has a large derivative value
2. It have to be easy for computing the derivative
3. It have to be zero center.

Above rules make operation of MLP easy at back propagation and gives information to next node without any information loss.

The single perceptron structure has an activation function value for each coordinate in a space having each element of the input vector as an axis. If the activation function is a step function having 0 and 1 as definitions, it is possible to observe an obvious solution space that distinguishes the active area within the corresponding space. This forms a linear relationship. So, it has a limitation in nonlinear problems such as XOR.

블랙, 어둠이(가) 표시된 사진

자동 생성된 설명

Figure 1. 2 Structure of Multiple Layer Perceptron[[1]](#footnote-2)

MLP is a structure that can solve this nonlinearity problem. MLP stands for multilayer perceptron and consist of an input layer, an output layer, and hidden layers between them. By connecting with adjacent layers, this can catch a non-linear feature of input.

Returning to the XOR problem, we can solve the problem with MLP. 도표, 스크린샷, 라인, 텍스트이(가) 표시된 사진

자동 생성된 설명

Figure 1. 4 MLP, description of Perceptron activation by positional case

In the diagram above, each dividing line refers a perceptron’s activation decision boundary. In such a MLP structure, each perceptron gives its activation value to next layer perceptron by multiplying a weight. The weight contributes to the representation of non-linear relationships in the MLP structure.

So, the weight is a main key of MLP structure to represent predictions appropriately. Prior to linear regression of polynomial model, it was easy to update weight. However, MLP is the composite function of perceptron and its weight, lot of computation is required to update each weight.

Back propagation is a method of reducing the amount of computation by recycling previous calculated values. To understand why a value is recycled, we must remind concept of chain rule.

In Leibniz notation, if and are both differentiable functions, then.

.[[2]](#footnote-3)

The perceptron is composed of a function called the sum of sums and an activation function with the sum of sums as a function of belonging, and MLP is composed by connecting each perceptron in layers. Therefore, if the composite function is inversely differentiated for each layer, a lot of input computing resources may reduce to calcultae the differential value.

Calculates the cost function.

Output layer: Diffrentiate Cost function over activation function

Output layer: Differentiate Cost function over activation function

Output layer: Gradient operation of Cost function against weighted matrix

Gradient operation for the (L-3) th weight matrix according to above operation

In this situation, let’s define new notation , represents the error derivative used in the back-propagation algorithm. This means the differentiation of the output value of the corresponding perceptron with respect to the amount of change in the loss function. So, we can represent like this :

Example of Caculation of (L-3) th layer

Therefore, the main idea of implementing the back-propagation algorithm is to calculate delta according to the Chain rule and store it in each perceptron. After storing it in each perceptron, the weight update is then calculated with the delta value.

**2. Design**

**2-a. Main Idea**

To implement the MLP, it is essential to consider how each data element should be structured. Since an MLP is conceptual structure, how it manifests as a logical data structure varies depending on the user’s implementation. Thanks to Object-oriented programming language, we can easily address the MLP structure by using class.

So, we can think about how to construct the logical units. The entire MLP structure can be created as a class, and each layer, node also. First, implementing class for a single perceptron has a limitation that every behavior has to be encoded by graph searching. This means the complexity of code will increase, even still cannot use vector and matrix. So, implementing class of layer is the best way for low code complexity.

Class MLP\_network is wrapper class for layer because class Layer is simple data container as a vector. So, every behavior of MLP is implemented on wrapper class. The program oriented to configure network flexibly. To initialize MLP network, the node configuration is needed.

The program is applied mini-batch method, also batch available. However, batch method is not recommended to avoid trapping in local minima.

**2-b. Program Description and Develop Environment**

IDE

* Visual Studio : Python 3.10.3
* Data Spell 23.2.1 : Python 3.9.13

Package

* Pandas : 1.4.4
* Numpy : 1.26.0

1. **Class MLP\_network**

To read the entire code, Please Refer the Appendix 1.

node\_configuration : information about how the nodes configure in this network, represent with list

Ex) [1, 2, 3, 2, 1] : the first number is input layer nodes. The integer indicates the number of nodes in each layer in order

activation\_config : information about which layer has the which activation function

Ex) [‘identity’, ‘sigmoid’, ‘tanh’, ‘tanh’, ‘identity’]

bias\_list : a list containing the bias values for each neural network layer

weight\_list : A list of weight matrices connecting each pair of adjacent layers in the network

self\_layers : A list of layers within the neural network, each comprising neurons and activation functions

stack\_delta\_list : A list used to store the gradients for each layer’s weights during backpropagation

The class is responsible for propagating data on the network. Forward propagation perform using weights, backpropagation perform to calculate errors in the reverse direction, and finally, data is accumulated and updated for each batch with error value. To perform such weight operation, we need to consider about weight and vector values. This is the forward propagation of the code.

도표, 라인, 원, 폰트이(가) 표시된 사진

자동 생성된 설명

Conversely, in the case of back-propagation, is calculated and accumulated in the temporary variable stack\_delta\_list.

The process of calculating is as follows.

**Case 1** : Find the of the output layer

Y represents actual label of the input data

**Case 2** : Find the of the hidden layer

represents the activation function.

Above operation can be calculated with vector operations at once.

Finally weight update is as follows

**B. Class Layer**

To read the entire code, Please Refer the Appendix 1.

num\_node : the integer of node in this layer object

z\_vector : the vector of each perceptron about weighted summation

a\_vector : the vector of each perceptron about activation function for weighted summation

delta\_vector : the vector of each perceptron’s error gradient

\_activ\_info : the string of information about layer’s activation function[[3]](#footnote-4)

Class layer contain node information, differentiation information, and several vector information, including z\_vector for weighted summation, a\_vector for activation function return, delta\_vector and batch\_delta\_vector. This information use on forward propagation and back propagation to update each prediction and weight.

When a layer object is created, it is necessary to configure how many nodes will be created and which activation function to use in that layer. In this implementation, the identity function, the logistic sigmoid , and the tanh can used as the activation function

The activation\_diff returns a differenetial value according to the activation function information of the layer.

**3. Evaluation Results**

**3-a Evaluation Setup**

With the prepared MLP program, we just configured MLP\_network as follows.

mlp = MLP\_network([1,3,8,5,2,1], ['identity','sigmoid','tanh','tanh','identity','identity'])

Using the code below, 80% of the dataset is used as a training dataset and the remaining 20% is used for validation.

base\_data\_set = pd.read\_csv('hw2\_data.csv')

training\_data\_set = base\_data\_set.sample(frac=1).reset\_index(drop=True)

spliter = int(len(base\_data\_set) \* 0.8)

training\_data\_set = base\_data\_set[:spliter]

validation\_data\_set = base\_data\_set[spliter:]

**3-b Training / Validation Error**

for i in range(30):

if i != 0:

mlp.batch\_iteration(training\_data\_set, 5,0.1,0)

else :

mlp.batch\_iteration(training\_data\_set, 5,0.1,0, True)

When the MLP\_network object call batch\_iteration function at first time, the reset option must be set to True to generate the weight matrix and bias vector of the MLP network.

for i in range(30):

if i != 0:

mlp.batch\_iteration(training\_data\_set, 5,0.1,0)

else :

mlp.batch\_iteration(training\_data\_set, 5,0.1,0, True)

**Validation Result :** 0.00667241

**3-c Final Model and Parameter**

**Layer Activation Func.**

-1 : 'identity'

-2 : 'sigmoid'

-3 : 'tanh'

-4 : 'tanh'

-5 : 'identity'

-6 : 'identity'

**Validation Error**

Error : 0.00667241

원, 도표, 라인, 대칭이(가) 표시된 사진

자동 생성된 설명

Figure 3. 1 structure of model

|  |
| --- |
| **-0.0626** |
| **-0.3334** |
| **-0.2859** |

|  |  |  |
| --- | --- | --- |
| **1.03040987** | **1.55830559** | **2.85109778** |
| 3.2555728 | 0.29564105 | -1.08821851 |
| -1.37401311 | -0.66947426 | -0.88086979 |
| 1.84051141 | -1.01754509 | 0.4297664 |
| -0.92879879 | 0.3692871 | 1.0636455 |
| 0.07715822 | -1.33588418 | -1.74722716 |

**Weight 0**

**Weight 1**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **-2.15139853** | **1.94750005** | **-1.05733621** | **0.52980642** | **-0.23737413** | **-1.0184331** | **-0.85363257** | **-0.85504241** |
| 2.28129609 | 2.11232905 | 0.08438671 | 0.54566598 | -1.32141386 | 0.29976349 | 0.16346127 | -2.19611427 |
| -0.35517054 | -0.11276794 | 0.07820863 | 0.77730904 | 0.18551326 | 0.42609964 | -0.38942077 | 1.0104366 |
| -0.12239945 | 1.50855472 | 0.40969918 | -0.51007583 | 0.56447497 | 0.50382641 | -0.96885717 | 1.05273143 |
| -2.69726464 | 0.01077343 | 0.06308951 | -0.04455208 | -0.24313317 | 0.7508244 | -1.91751802 | -0.63176971 |

**Weight 2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **-0.50125905** | **0.37796811** | **0.19167706** | **0.17398634** | **2.50655449** |
| -0.28956216 | 0.33905885 | -0.29938257 | -0.10651709 | 1.34568317 |

**Weight 3**

|  |  |
| --- | --- |
| **2.24649081** | **0.69128164** |

**Weight 4**

**3-d Final result.**

**Training Option**

Batch size : 5

Learning rate : 0.1

Regularization Bias : 0

Epoch : 30

Iteration : 800

**Validation Error**

Error : 0.00667241

텍스트, 그래프, 도표, 스크린샷이(가) 표시된 사진

자동 생성된 설명

**Reference**

Rcassani, ‘MLP Example,’ Github repository, Last modified November 26, 2023, Accessed November 26, 2023, <https://github.com/rcassani/mlp-example>

Stewart. James, Caculus Early Transcendentals ed. 7th ver. Yonsei University Department of Mathematics, p. 173.

Appendix 1.

class MLP\_network:

def \_\_init\_\_(self, node\_config: list, activation\_config: list):

# node\_config에는 Layer에 몇 개의 노드를 가질 것인지 나타낸다.

# Ex) [1, 2, 3, 3, 3, 1] : 5개의 층에 각 층별로 노드가 1개, 2개, 3개, ... 1개씩 존재한다.

self.node\_configuration = node\_config

self.activation\_config = activation\_config

# 각 노드에 하나씩 존재하도록 한다.

# 별도의 업데이트 과정을 거치도록 한다.

self.bias\_list = list()

# 3차원 큐브의 형태로 구성하고 있는 weight 정보

# 한 Layer가 다른 layer로 전사될 때 하나의 Matrix가 사용된다.

# 처음 학습을 시작할 때 별도로 weight\_list를 초기화하는 과정을 거치도록 한다.

self.weight\_list = list()

# layer를 담도록 하는 list 객체

# MLP 선언과 동시에 layer 정보가 담겨야 한다.

self.layers = []

self.stack\_delta\_list = []

return

def \_configure\_network(self):

for index in range(len(self.node\_configuration)):

node\_info = self.node\_configuration[index]

activation\_info = self.activation\_config[index]

self.layers.append(Layer(node\_info,activation\_info))

self.layers[0].is\_first = True

self.layers[-1].is\_final = True

for i in range(len(self.node\_configuration)-1):

# MLP에 초기 무직위 weight 정보 구성

self.weight\_list.append(2 \* (np.random.rand(self.node\_configuration[i+1], self.node\_configuration[i])) - 1)

self.bias\_list.append(2\*np.random.rand(self.node\_configuration[i+1]) - 1)

self.stack\_delta\_list.append(np.zeros((self.node\_configuration[i+1], self.node\_configuration[i])))

def \_refresh\_stack\_delta\_list(self):

for i in range(len(self.node\_configuration)-1):

self.stack\_delta\_list[i] = np.zeros((self.node\_configuration[i+1], self.node\_configuration[i]))

def forward\_propagation(self):

for i in range(len(self.node\_configuration)-1):

tempInput : np.ndarray = self.layers[i].a\_vector

tempWeight : np.ndarray = self.weight\_list[i]

tempOutput : np.ndarray = np.zeros((1,self.node\_configuration[i]))

tempOutput : np.ndarray = tempWeight @ tempInput

self.layers[i+1].z\_vector = tempOutput + self.bias\_list[i]

self.layers[i+1].activation\_function()

return

def backpropagation(self, y):

self.layers[-1].delta\_vector = self.layers[-1].a\_vector - y

for i in range(len(self.node\_configuration)-2, 0, -1):

temp\_weight\_holder = self.weight\_list[i].transpose()

self.layers[i].delta\_vector = temp\_weight\_holder @ self.layers[i+1].delta\_vector \* self.layers[i].activation\_diff()

temp = np.outer(self.layers[i+1].delta\_vector, self.layers[i].a\_vector)

self.stack\_delta\_list[i] = self.stack\_delta\_list[i] + temp

return

def batch\_iteration(self, data\_set : pd.DataFrame, batch\_size : int,learning\_rate : float, bias\_lambda : float, reset = False):

if reset == True :

self.\_configure\_network()

for batch in np.array\_split(data\_set.sample(frac = 1), len(data\_set) // batch\_size):

for index, row in batch.iterrows():

self.set\_value(row['x'])

self.forward\_propagation()

error = 0.5 \* (self.layers[-1].a\_vector - row['y'])\*\*2

print('X = %20s :: Y = %20s :: PREDICT = %20s :: ERROR = %20.10s'%(row["x"],row["y"],self.layers[-1].a\_vector,float(error)))

self.backpropagation(row['y'])

self.weight\_update(batch\_size, learning\_rate, bias\_lambda)

for i in self.layers:

i.delta\_vector = None

i.batch\_delta\_vector = 0

self.\_refresh\_stack\_delta\_list()

return

def weight\_update(self, batch\_size : int,learning\_rate : float, bias\_lambda: float ):

for i in range(len(self.weight\_list)):

D = (1/batch\_size)\*(self.stack\_delta\_list[i] + bias\_lambda \* self.weight\_list[i])

self.weight\_list[i] = self.weight\_list[i] - learning\_rate \* D

def validation(self, validation\_data\_set):

total\_error = 0

total\_samples = len(validation\_data\_set)

for index, row in validation\_data\_set.iterrows():

self.set\_value(row['x'])

self.forward\_propagation()

error = self.layers[-1].a\_vector - row['y']

total\_error += error \*\* 2

mse = total\_error / total\_samples

return mse

def set\_value(self,x):

self.layers[0].a\_vector = np.array([x])

def just\_prediction(self, input\_data):

self.set\_value(input\_data)

self.forward\_propagation()

return self.layers[-1].a\_vector[0]

Appendix 2.

class Layer:

def \_\_init\_\_(self, num\_nodes, diff: str, is\_first:bool = False, is\_final:bool=False):

self.num\_node = num\_nodes

self.z\_vector : np.ndarray = None

self.a\_vector : np.ndarray = None

self.delta\_vector : np.ndarray = None

self.\_diff\_info = diff

self.is\_final = False

self.is\_first = False

def activation\_function(self):

if self.\_diff\_info == 'sigmoid':

self.\_sigmoid()

elif self.\_diff\_info == 'identity':

self.\_identity()

elif self.\_diff\_info == 'tanh':

self.\_tanh()

def \_sigmoid(self):

self.a\_vector = 1 / (1 + np.exp(-self.z\_vector))

def \_identity(self):

self.a\_vector = self.z\_vector

def \_tanh(self):

self.a\_vector = np.tanh(self.z\_vector)

def activation\_diff(self):

if self.\_diff\_info == 'sigmoid':

return (1 / (1 + np.exp(-self.z\_vector))) \* (1 - (1 / (1 + np.exp(-self.z\_vector))))

elif self.\_diff\_info == 'identity':

return 1

elif self.\_diff\_info == 'tanh':

return 1 - np.tanh(self.z\_vector) \*\* 2

1. Rcassani, ‘MLP Example,’ Github repository, Last modified November 26, 2023, Accessed November 26, 2023, <https://github.com/rcassani/mlp-example> [↑](#footnote-ref-2)
2. Stewart. James, Caculus Early Transcendentals ed. 7th ver. Yonsei University Department of Mathematics, p. 173. [↑](#footnote-ref-3)
3. In the repository, the actual member variable labeled as \_diff, this is error. So, just revised in this paper [↑](#footnote-ref-4)