

Machine Learning – Homework 3

- Neural Network as a Function Approximator -

ChangYoon Lee

Dankook University, Korea Republic

32183641@gmail.com

Abstract. Multi-layer Perceptron (MLP) model is the basic model in the neural network. The following model consists of the input layer, multiple hidden layers, and an output layer. These layers are fully connected and the weights for each layer, which is the set of perceptron, are updated by the backpropagation algorithm. The MLP model can be used in both regression and classification. By using these concepts, we will present the implemented MLP regressor model, which approximates the function of given x values and y values, and the process of training it by using the given training set. Also, there will be regressions by using MLP regressor with different configurations of the dense layers and the different number of the dense layers, and comparison among them.

Keywords: Multi-Layer Perceptron, MLP, Activation Function, Perceptron Model, Backpropagation

1 Introduction

Neural networks are a subset of machine learning and are at the heart of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another (*IBM Cloud Education, what are neural networks?*).

Node layers, which include an input layer, one or more hidden layers, and an output layer, make up neural networks. Each node, or neuron, is connected to another and has a weight and threshold that go along with it. If the output of any individual node is above the specified threshold value, that node is activated, sending data to the next layer of the network (*IBM Cloud Education, what are neural networks?*). Otherwise, no data is passed along to the next layer of the network (*IBM Cloud Education, what are neural networks?*).

In this paper, we will discuss the basic concepts that are used to implement the neural network model: perceptron model, activation functions, multi-layer perceptron (MLP), MLP regressor, chain rule, and backpropagation. Then, we will discuss how we evaluate the performance of the prediction model by evaluating the error: mean squared error (MSE). After discussing all the concepts, we will present the implemented python code of multiple classes that are used in the MLP regressor, which represents training the model with the training data. In the result, we will show the prediction graph according to the given data set and the coefficients for the predicted model. Also, there will be comparisons between the built-in logistic regression models and the implemented models.

2 Concepts

2.1 Perceptron Model

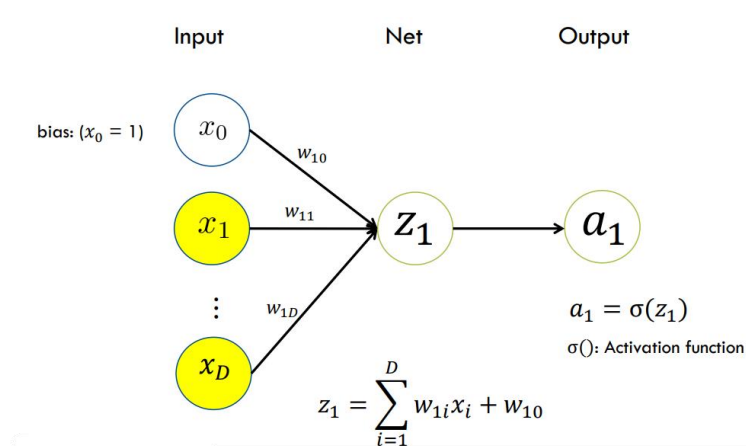


Figure 1 - Perceptron Model (Lee, [Slide] 05. Neural Networks)

The perceptron model is one of the first neural networks that was developed taking into consideration the natural human brain. This system had an input layer that was directly connected to the output layer and was good to classify linearly separable patterns.

The perceptron model consists of the following three main components, which are presented in Figure 1:

- **Input layer:** The input layer is a primary component of the perceptron model which accepts the initial data into the system for further processing.
- **Weight and bias:** The weight parameter represents the strength of the connection between units. Weight is directly proportional to the strength of the associated input neuron in deciding the output. Bias can be considered as the line of intercept in a linear equation.
- **Activation Functions:** Activation functions are the final and important components that help to determine whether the neuron will expire or not. The following functions will be discussed in the later section.

The perceptron model begins with the multiplication of all input values and their weights, then adds these values together to create the weighted sum. Then this weighted sum is applied to the activation function to obtain the desired output.

By accumulating multiple perceptron, we can build a single layer that is used in machine learning algorithms. Also, by accumulating the layer, we can build the multi-layer model, which will be used mainly in our paper.

2.2 Activation Functions






Name	Plot	Equation	Derivative (with respect to x)
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a. Sigmoid or Soft step)		$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$ ^[1]	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$	$f'(x) = 1 - f(x)^2$
Rectified linear unit (ReLU) ^[12]		$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases} = \max\{0, x\} = x \mathbf{1}_{x>0}$	$f'(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases}$

Figure 2 - Activation Functions (Lee, [Slide] 05. Neural Networks)

By calculating the weighted total and then adding bias to it, the activation functions determine whether or not a neuron should be stimulated. The purpose of the activation function is to introduce non-linearity into the output of a neuron. The equation of activation functions and their derivatives that are used in this paper are presented in Figure 2.

The identity function has an equation like a straight line. No matter how many layers the model has, if all the neurons use the identity function, the final activation of the last layer will output the input of the first layer. Therefore, the identity function is only used on the output layer.

The sigmoid function is a function that is plotted in the 'S' shape. It is a non-linear function, where y values are very steep in a certain range of x values. This means that the small changes in x would also bring about a large change in the value of y . Since there are some problems with this function's use in the hidden layer, such as not being zero-centralized, vanishing gradient (a situation where the effect from the first layer converges to zero due to the form of the derivative of activation function), and longer computation due to the exponential function, it is not used in the hidden layer.

The hyperbolic tangent function is a function that works almost better than the sigmoid function. It is also a non-linear function. It is usually used in the hidden layers of a neural network as its values lie between -1 to 1 hence the mean for the hidden layer comes out to be zero or very close to zero. Therefore, it is zero-centralized and does not suffer from the vanishing gradient. However, it still shows a longer computation time due to the exponential function.

ReLU is the abbreviation for rectified linear unit. The most common activation function is this one. If x is positive, it outputs x ; otherwise, it outputs zero. This function is non-linear, and it is less computationally expensive than hyperbolic tangent and sigmoid functions because it involves simpler mathematical operations. At times only a few neurons are activated, making the network sparse and making it efficient and easy for computation.

From the explanation of the activation functions above, we can see that we use the non-linear model in the hidden layers. The reason why is that a neural network without a non-linear activation function is just a linear regression model. Therefore, we may use the non-linear activation function in the hidden layer, so that it does a non-linear transformation to the input and make it capable to learn and perform more complex tasks.

3.5 Multi-Layer Perceptron (MLP)

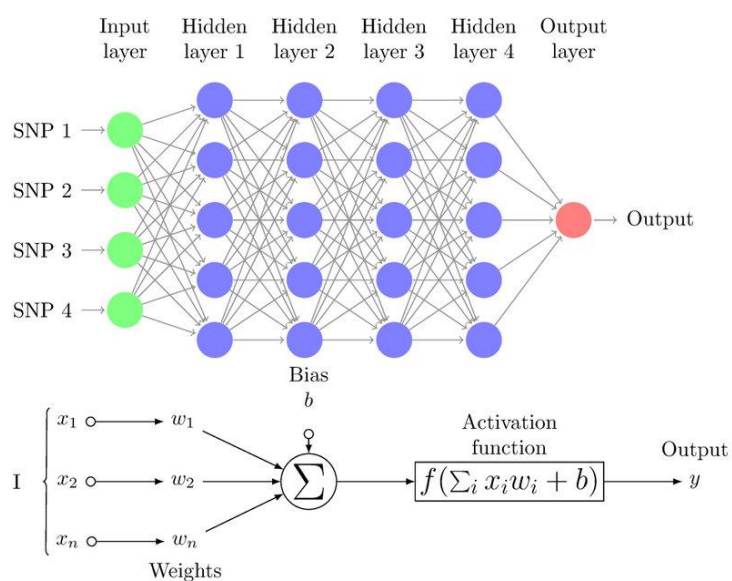


Figure 3 - Multi-Layer Perceptron (Multi-layer perceptron (MLP) diagram with four hidden layers and a ...)

Multi-layer perceptron (MLP) is a supervised learning algorithm that learns a function by training on a dataset, where it consists of multiple layers. It can learn a non-linear function approximator for either classification or regression by using the activation functions that we discussed in the previous section. One or more non-linear layers known as the hidden layers may exist between the input layer and the output layer.

A group of neurons that represent the input features make up the input layer. A weighted linear summation followed by a non-linear activation function, such as the hyperbolic tangent function, modifies the data from the preceding layer in each neuron in the buried layer. The values from the final hidden layer are sent to the output layer, where they are converted into output values.

The ability to train non-linear models and to learn models in real time are two benefits of multi-layer perceptron (MLP) models. However, the disadvantages of the multi-layer perceptron (MLP) model are MLP with hidden layers has a non-convex loss function were three exits more than one local minimum, MLP requires tuning several hyperparameters such as the number of hidden neurons, layer, and iterations, and MLP is sensitive to feature scaling.

3.6 Regressor with Multi-Layer Perceptron (MLP)

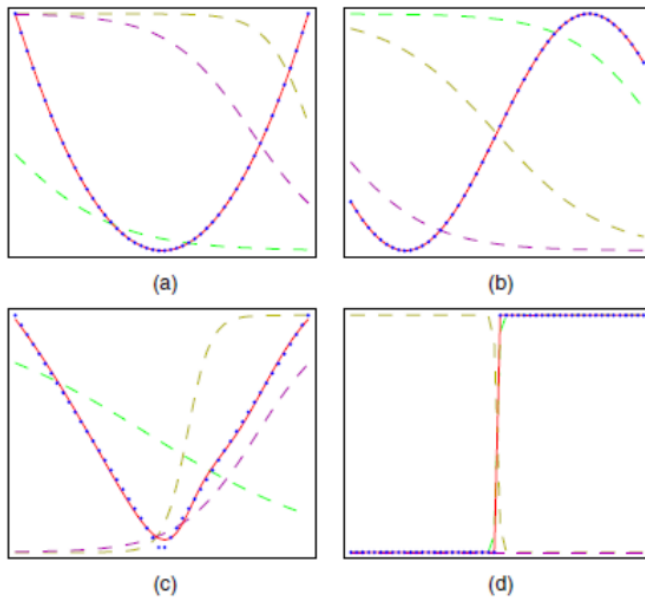


Figure 4 - MLP Regression on Various Functions (Lee, [Slide] 05. Neural Networks)

Regressor with multi-layer perceptron (MLP) that trains using backpropagation, which is the concept that will be discussed in the later section, with no activation function in the output layer, which can also be seen as using the identity function as activation function. Therefore, it uses the square error as the loss function, and the output is a set of continuous values. Figure 4 shows an example of the regression done by the MLP models.

3.7 Backpropagation

Now, let's focus on training the multi-layer perceptron (MLP) model. The basic method to update the weights is using the gradient descent algorithm with forward propagation, which forwards the input variable from the input layer to the output layer and updates the weights for each layer one by one. Since the forward propagation is sequentially processed and the output of the operation comes out at the end of the layer (output layer), it cannot regulate the weights of the layers.

Therefore, we use the error backpropagation algorithm to overcome the following limitation. We reused the previous calculations for computing current gradients. To process the following operation, we use the chain rule, which is presented in Figure 5.

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:

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

Figure 5 - Chain Rule

From the output layer, we perform the backpropagation by using the chain rule. By computing the loss of the output and using it to the previous layer with the chain rule, we can get the error in the previous layers, and process the gradient descent algorithm to update the weights.

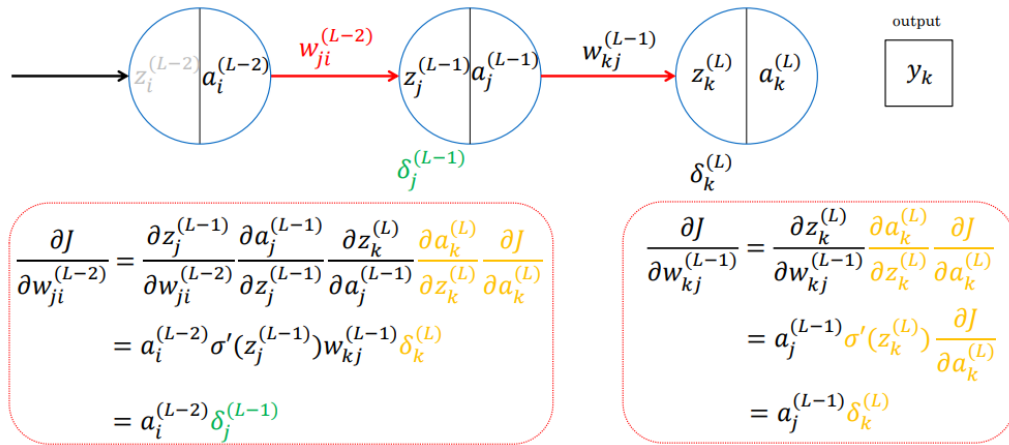


Figure 6 - Back Propagation in the MLP Model with Two Hidden Layers (Lee, [Slide] 05. Neural Networks)

Figure 6 shows the example of backpropagation in the MLP model with two single hidden layers. The left rectangular in Figure 6 shows the backpropagation operation from L hidden layer to the previous L-1 hidden layer. By applying the chain rule, we can get the corresponding error in the L-1 hidden layer. Also, the right rectangular in Figure 6 shows the backpropagation operation from the L-1 layer to the L-2 layer. Same as the previous operation, by applying the chain rule, we can get the corresponding error in the L-2 hidden layer within the error in the L-1 hidden layer. By performing the following operation on the input layer, we can update the weights in the layers by using the errors in each layer.

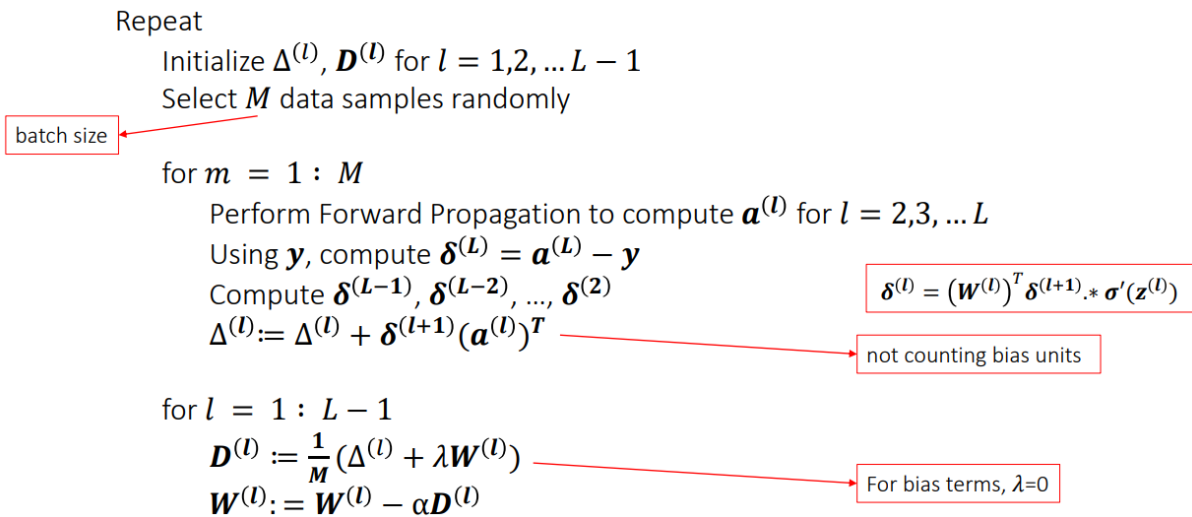


Figure 7 - Back Propagation Algorithm (Lee, [Slide] 05. Neural Networks)

When we generalize the following back propagation operation in the form of an algorithm, we can use the pseudo-code that is presented in Figure 7. By implementing the code in the form of an algorithm, which is represented in Figure 7, we can successfully process the backpropagation operation.

3 Implementation

3.1 Representation of Given Data Set

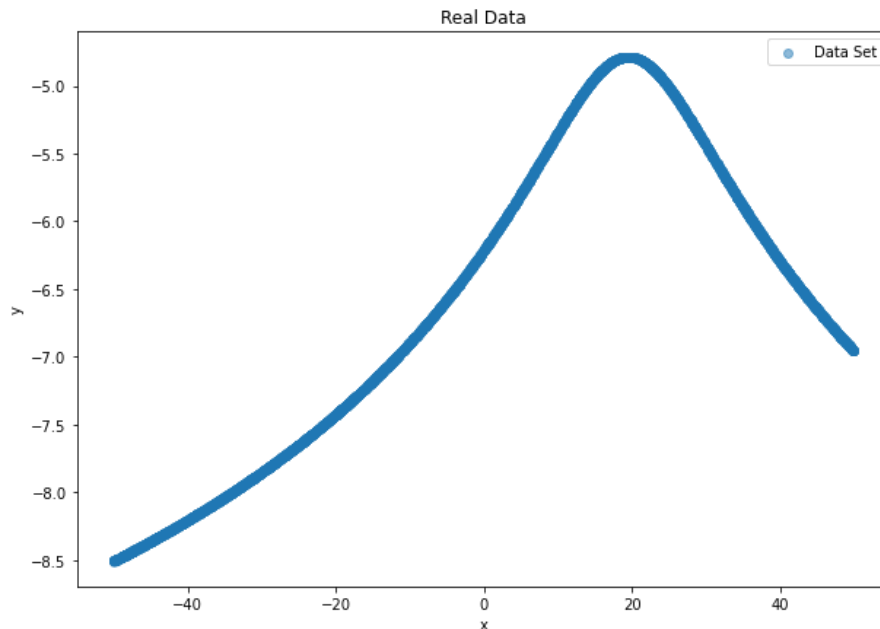


Figure 8 - Visualization of the Given Data Set

The main purpose of this project is to code a neural network that works as a function approximator. The approximated function should be able to describe the given data. The given dataset has two sets of values: one is x value, and the other one is y value. Figure 8 presents a visualization of the given data set.

3.2 Define Activation Functions

```
1 import numpy as np
2
3
4 # Sigmoid
5 class Sigmoid:
6     def forward(self, x):
7         return 1.0 / (1.0 + np.exp(-x))
8
9     def derivate(self, x):
10        return (1 - self.forward(x)) * self.forward(x)
11
12
13 # Relu
14 class Relu:
15     def forward(self, x):
16         return np.where(x > 0, x, 0.0)
17
18     def derivate(self, x):
19         return np.where(x > 0, 1.0, 0.0)
```

Figure 9 - Implementation of Sigmoid and ReLU Activation Functions

```

22 # Hyperbolic Tangent
23 class Tanh:
24     def forward(self, x):
25         return np.tanh(x)
26
27     def derivate(self, x):
28         return 1 - np.tanh(x) ** 2
29
30
31 # Identity
32 class Identity:
33     def forward(self, x):
34         return x
35
36     def derivate(self, x):
37         return 1

```

Figure 10 - Implementation of Hyperbolic Tangent and Identity Activation Functions

Figures 9 and 10 present the implemented code of the activation functions, such as sigmoid, ReLU, hyperbolic tangent, and identity functions. We implemented not only the forward function (returns the output according to the function) but also the derivative of the following functions. These derivatives will be used in both updating weights and errors in the gradient descent algorithm that is applied to each layer in the multi-layer perceptron (MLP) regressor model.

3.3 Define Initializer

```

1 import numpy as np
2
3
4 # Randomly generates the weights and bias
5 class Rand:
6     def initialize(self, layer):
7         w = np.random.randn(layer.input_dim, layer.output_dim)
8         b = np.random.randn(1, layer.output_dim)
9         dw = np.zeros([layer.input_dim, layer.output_dim])
10        db = np.zeros([1, layer.output_dim])
11        return w, b, dw, db

```

Figure 11 - Implementation of Initializer that Randomly Initializes the Weights in the Layer

Figure 11 presents the implemented code of the initializer that randomly initializes the weights in the layer. Since we can either initialize the weights in the layer into zeros or randomly generate real numbers, we choose the second method as an initialize the weights in the layer for this project. One consideration with this decision is that since the weights are generated randomly in the initial phase, we do not know whether the model starts from the area close to the optimal point of error function or not. Therefore, it can easily converge to the local minimum of the error function if it initially gets the bad random weights. To mitigate the following situation, we should implement the proper optimizer, such as the Adam optimizer.

3.4 Define Layer

```
1 import numpy as np
2
3
4 # Dense layer
5 class Dense:
6     def __init__(self, units=None, activation=None, input_dim=None, initializer=Rand(), initialize=False):
7         self.w = None # weight
8         self.b = None # bias
9         self.z = None # input
10        self.a = None # output
11        self.d = None # derivative
12        self.dw = None # derivative of weight
13        self.db = None # derivative of bias
14
15        self.output_dim = units # number of output
16        self.input_dim = input_dim # number of input
17        self.activation = activation # activation function
18        self.initializer = initializer # initializer of the layer
19
20        if initialize:
21            self.reset_layer()
22
23
24 # Initialize the weights and bias of layer
25 def reset_layer(self):
26     w, b, dw, db = self.initializer.initialize(self)
27     self.w = w
28     self.b = b
29     self.dw = dw
30     self.db = db
31
32
33 # Forward operation
34 def forward(self, x, update=True):
35     z = np.matmul(x, self.w) + self.b
36     a = self.activation.forward(z)
37     if update:
38         self.z = z
39         self.a = a
40     return a
41
42
43 # Update the derivative of weight and bias for the layer
44 def update_delta(self, next_layer):
45     d = np.matmul(next_layer.d, next_layer.w.T) + self.activation.derivate(self.z)
46     self.d = d
47
48
49 # Update the weight and bias for the layer
50 def update_gradients(self, a_in):
51     d_out = self.d
52     self.db = d_out.sum(axis=0).reshape([1, -1])
53     self.dw = np.matmul(a_in.T, d_out)
```

Figure 12 - Implementation of the Dense Layer

Figure 12 presents the implemented code of the dense layer that is used in the multi-layer perceptron (MLP) model. When the dense layer is initialized, it sets the weights in the random real numbers and derivatives of the weights to zero, by calling the initializer. We implemented the forward function that performs the dot product of input and weights, then add the bias to it, and generate the corresponding output of the layer. If the updated argument is true, it updates the value of input and output. The updated argument is true when it performs forward propagation. When the layer performs backpropagation, the update is false. There is also a function that updates the delta value, which represents the error in the layer, and a function that updates the derivative of weights, which is used to update the weights of the layer.

3.5 Define Loss

```
1 import numpy as np
2
3
4 # Loss function -> mean squared error (MSE)
5 class MSE:
6     def forward(self, actual, prediction):
7         return 0.5 * ((prediction - actual) ** 2)
8
9     def derivate(self, actual, prediction):
10        return prediction - actual
```

Figure 13 – Implementation of the Loss Function for Mean Squared Error

Figure 13 presents the implemented code of the loss function for mean squared error that is used in the backpropagation to update the weights in the layers. The following function is used to get the loss from the output layer to the input layer and each loss in each layer will be used to update the weights of each layer. The forward function is used to get the result from the MSE function, and the derivative function is used to get the derivative value of the MSE result.

3.6 Define Optimizer

```
1 import numpy as np
2
3
4 # General optimizer for gradient descent algorithm
5 class GradientDescent:
6     def __init__(self, learning_rate=0.001):
7         self.learning_rate = learning_rate
8
9     # Initialize the parameters
10    def initialize_parameters(self, layers):
11        return layers
12
13    # Update weights of the layer
14    def update_weights(self, layers):
15        for i in range(len(layers)):
16            layers[i].w = layers[i].w - self.learning_rate * layers[i].dw
17            layers[i].b = layers[i].b - self.learning_rate * layers[i].db
18        return layers
```

Figure 14 - Implementation of the Gradient Descent Optimizer

Figure 14 presents the implemented code of the gradient descent optimizer that is used to optimize the weights in the layers. Since the general gradient descent algorithm does not add additional parameters to the weights, in the initialization function, it just returns the layers themselves as an output. In the weight update function, it updates the weights in each layer by using the weights and the derivatives of the weights and returns the layers themselves.

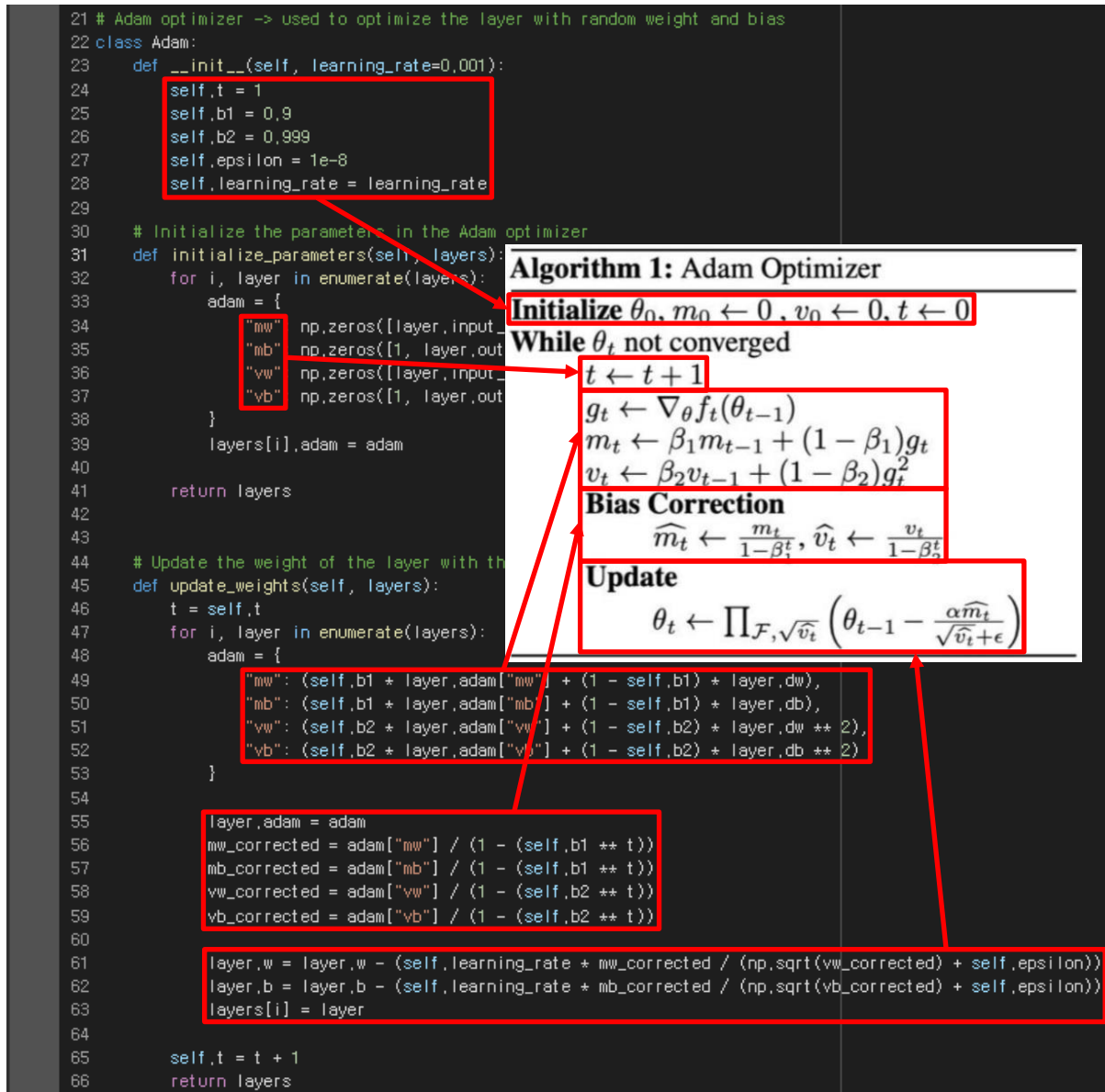


Figure 15 - Implementation of the Adam Optimizer

Figure 15 presents the implemented code of the Adam optimizer that is used to optimize the weights in the layers. The following optimizer follows the operation that is represented in the pseudo-code in Figure 15. As we mentioned previously, since the weights are determined randomly, the starting point in the loss function can be around the area of the local minimum. Therefore, by applying the Adam optimizer, which has the property of per-parameter learning rate and momentum, we can overcome the limitation of randomly generated weights in the initial state of the training. As the Adam optimizer shows the best performance when the parameters b_1 and b_2 are set as 0.9 and 0.999, we set those variables just like the following values and implemented the codes that work the same as the Adam algorithm. By using the following optimizer, we can overcome the problem of local minimum.

3.7 Define Model

```
1 import numpy as np
2
3
4 # Multi-layer perceptron model
5 class MLPRegressor:
6     def __init__(self):
7         self.layers = []      # List of layers
8         self.n_layers = 0     # Number of layers
9         self.trainer = None   # Trainer
10        self.train_log = None  # Log generated from the trainer
11
12
13    # Add the layer to the MLP model
14    def add(self, layer):
15        if len(self.layers) > 0:
16            layer.input_dim = self.layers[-1].output_dim
17
18            layer.reset_layer()
19            self.layers.append(layer)
20            self.n_layers += 1
21
22    # Predict output from the input
23    def predict(self, x):
24        pred = self.forward_prop(x, update=False)
25        return pred
26
27
28    # Train the MLP model
29    def train(self, loss, train_data, optimizer=Adam(), params=None):
30        self.trainer = ModelTrain()
31        self.trainer.train(self, loss, train_data, optimizer, params)
32
33
34    # Forward propagation
35    def forward_prop(self, x, update=True):
36        a = x
37        for layer in self.layers:
38            a = layer.forward(a, update=update)
39        return a
40
41
42    # Back propagation
43    def back_prop(self, x, y, loss):
44        self.update_deltas(loss, y)
45        self.update_gradients(x)
46
47
48    # Update the derivatives of weight and bias in the layers
49    def update_deltas(self, loss, y):
50        for i, layer in enumerate(reversed(self.layers)):
51            if i == 0:
52                d = loss.derivate(y, layer.a) + layer.activation.derivate(layer.z)
53                layer.d = d
54            else:
55                layer_next = self.layers[-i]
56                layer.update_delta(layer_next)
57
58
59    # Update the weight and bias in the layers
60    def update_gradients(self, x):
61        for i, layer in enumerate(self.layers):
62            if i == 0:
63                a_in = x
64            else:
65                prev_layer = self.layers[i - 1]
66                a_in = prev_layer.a
67            layer.update_gradients(a_in)
68
69
70    def reset_layers(self):
71        for layer in self.layers:
72            layer.reset_layer()
```

Figure 16 - Implementation of the Multi-Layer Perceptron Model

Figure 16 presents the implemented code of the multi-layer perceptron (MLP) model that is mainly used for function approximation. When the MLP model is initialized, it generates the layer list, which stores the added layers, and the number of layers, the trainer model, which is set as Adam optimizer, and the training log, which stores the log of the train results. If the add function is called, it initializes the layer, adds the layer to the end of the model layer list, and increases the number of layer variables. If the prediction function is called, the forward propagation operation performed within the updated argument is false. It generates the prediction according to the trained MLP model. If the train function is called, it initializes the trainer and starts to train the model. If the forward propagation function is called, it performs the forwarding within the update argument is true and returns the result of the forwarding operation. If the backpropagation function is called, it operates by updating the loss and the weights of each layer. If the update derivative function is called, it updates the losses in each layer. If the update gradient function is called, it updates the weights in each layer. In the end, if the reset layer function is called, it resets all the weights in each layer.

3.8 Define Batcher

```

1 import numpy as np
2
3
4 # Batcher that generates the batch for the training
5 class Batcher:
6     def __init__(self, data, batch_size, shuffle_on_reset=False):
7         self.data = data
8         self.batch_size = batch_size
9         self.shuffle_on_reset = shuffle_on_reset
10
11         if type(data) == list:
12             self.data_size = data[0].shape[0]
13         else:
14             self.data_size = data.shape[0]
15
16         self.n_batches = int(np.ceil(self.data_size / self.batch_size))
17         self.idx = np.arange(0, self.data_size, dtype=int)
18         if shuffle_on_reset:
19             np.random.shuffle(self.idx)
20         self.current = 0
21
22
23 # Shuffle the index
24 def shuffle(self):
25     np.random.shuffle(self.idx)
26
27
28 # Reset the batcher
29 def reset(self):
30     if self.shuffle_on_reset:
31         self.shuffle()
32     self.current = 0
33
34
35 # Select the next batch for the training
36 def next(self):
37     batch = []
38     i_select = self.idx[(self.current * self.batch_size) : ((self.current + 1) * self.batch_size)]
39     for elem in self.data:
40         batch.append(elem[i_select])
41
42     if self.current < (self.n_batches - 1):
43         self.current = self.current + 1
44     else:
45         self.reset()
46
47     return batch

```

Figure 17 - Implementation of the Batcher

Figure 17 presents the implemented code of the batcher that sets the total number of training examples presented in a single batch. If the batcher is initialized, it initializes the data, batch size, number of batches, index to point each batch, current batch index, and whether to shuffle the indexes in the batch or not. If the shuffle function is called, it shuffles the indexes in the batcher. If the reset function is called, it shuffles the batcher's indexes and sets the current batch index to zero. If the next function is called, we select the next training example from the current batch index and return it. By using the batcher, we can select the training set and control the size of the training set in a single training step.

3.9 Define Trainer

```

1 import numpy as np
2 import pandas as pd
3
4
5 # Trainer parameters
6 default_params = {
7     "verbose": True,
8     "n_epoch": 1000,
9     "batch_size": 200,
10    "print_rate": 100,
11    "learning_rate": 0.5
12 }
13
14 # Model trainer
15 class ModelTrain:
16     def __init__(self, params=None):
17         self.batcher = None # Batcher
18         self.optimizer = None # Optimizer
19         self.train_params = default_params # Training parameters
20
21
22
23     # Train the given model with the optimizer and batcher
24     def train(self, model, loss, train_data, optimizer=Adam(), params=None):
25         self.optimizer = optimizer
26         model.layers = self.optimizer.initialize_parameters(model.layers)
27
28         if self.batcher is None:
29             self.batcher = Batcher(train_data, self.train_params["batch_size"])
30
31         epoch = 1
32         verbose = self.train_params["verbose"]
33
34         train_loss = []
35         model.train_log = []
36         while epoch <= self.train_params["n_epoch"]:
37             self.batcher.reset()
38
39             for batch_i in range(self.batcher.n_batches):
40                 batch = self.batcher.next()
41                 x_batch = batch[0]
42                 y_batch = batch[1]
43
44                 self.train_step(model, x_batch, y_batch, loss)
45
46                 loss_i = self.compute_loss(model.layers[-1].a, y_batch, loss)
47                 train_loss.append(loss_i)
48                 model.train_log.append(np.array([epoch, batch_i, loss_i]))
49
50                 if verbose and (epoch % self.train_params["print_rate"] == 0):
51                     print(f"epoch: {epoch} \t \t train_loss: {np.mean(train_loss)}")
52                 epoch += 1
53
54         model.train_log = np.vstack(model.train_log)
55         model.train_log = pd.DataFrame(model.train_log, columns=["epoch", "iter", "loss"])

```

```

58 # Performs a single training step
59 def train_step(self, model, x, y, loss):
60     _ = model.forward_prop(x)
61     model.back_prop(x, y, loss)
62     model.layers = self.optimizer.update_weights(model.layers)
63
64
65 # Compute the loss from the forward operation in current model
66 def compute_loss(self, actual, prediction, loss):
67     current_loss = loss.forward(actual, prediction)
68     return current_loss.mean()

```

Figure 18 - Implementation of the Trainer

Figure 18 presents the implemented code of the trainer, which trains the given model within the given parameters. If the trainer is initialized, then it defines the batch, optimizer, and the parameters that are used in training the model. If the train function is called, it performs the training in the number of the given epoch. For each epoch, it resets more batches, and the batcher generates the next batch training set. For each training set, it performs the training step, which is the backpropagation and updates the weights in each layer by the generated weights by the given optimizer. Then, it performs the forward propagation, calculates the loss with the mean squared error (MSE) function, and prints out the loss on the screen. If the train step function is called, it first performs the forward propagation to get the output, and from that output, it performs the backpropagation to the input layer, while changing the weights in each layer that the trainer meets. If the compute loss function is called, then it computes the loss between the output and the real data and returns the result of the MSE function.

3.10 Main

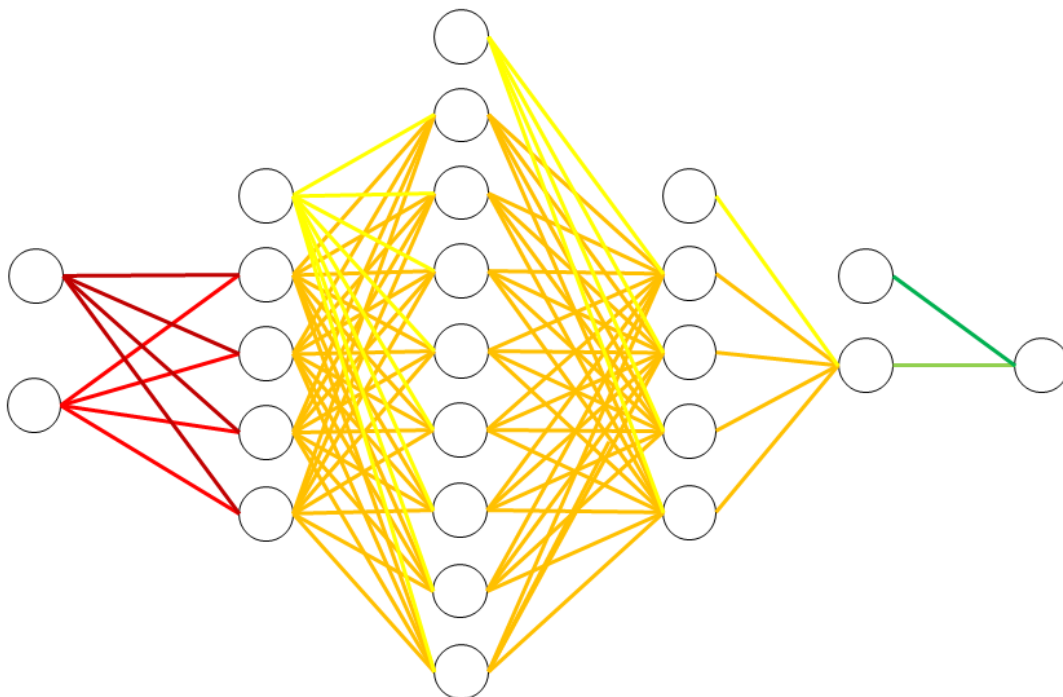


Figure 19 - Representation of the Optimized MLP Regressor Model

```

1 # Reshape the x and y data to fit them into the model
2 # For the henced prediction, we have to normalize the x data
3 x = np.array(data['x']).reshape(5000, 1)
4 y = np.array(data['y']).reshape(5000, 1)
5
6 # Generate the MLP regressor with the dense layers
7 model = MLPRegressor()
8 model.add(Dense(units=4, activation=Sigmoid(), input_dim=x.shape[1]))
9 model.add(Dense(units=8, activation=Sigmoid()))
10 model.add(Dense(units=4, activation=Identity()))
11 model.add(Dense(units=1, activation=Identity()))
12
13 # Select the loss function and perform training
14 loss = MSE()
15 model.train(loss, train_data=[x, y])
16
17 # Plot the real data and predicted result
18 plt.figure(figsize=(10, 7))
19 plt.title("Real Data")
20
21 plt.scatter(x, model.predict(x), color='tab:orange', alpha=0.5, label="Data Set")
22 plt.scatter(x, y, color='tab:blue', alpha=0.5, label="Real Set")
23 plt.xlabel('x')
24 plt.ylabel('y')
25 plt.legend()
26 plt.show()

```

Figure 20 - Implementation of the Main Statement

Figure 20 presents the implemented main code that initializes the multi-layer perceptron (MLP) regressor model, adds multiple dense layers into it, trains the following model by using the loss function as mean squared error (MSE) function, and plots the prediction of the model with a given data set to compare them. By using three dense layers which have the activation function of the sigmoid function and identity function, and the output layer which has the identity function, we can generate the MLP regressor model that can predict almost the same feature of the given data. The MLP regressor model is presented in Figure 19.

In the following implementation, we can change the configurations in the dense layers or the number of the dense layers. However, the implemented MLP regressor model is an optimized version of our implementation. Comparisons with different configurations will be discussed in the evaluation section.

5 Build Environment

The following build environments are required to execute the implemented code for task 1, task 2, and extra implemented prediction model.

Building Environment:

- The environment that can open the ipython notebook: COLAB, Jupyter Notebook
- Each cell in the notebook should be executed sequentially.

```

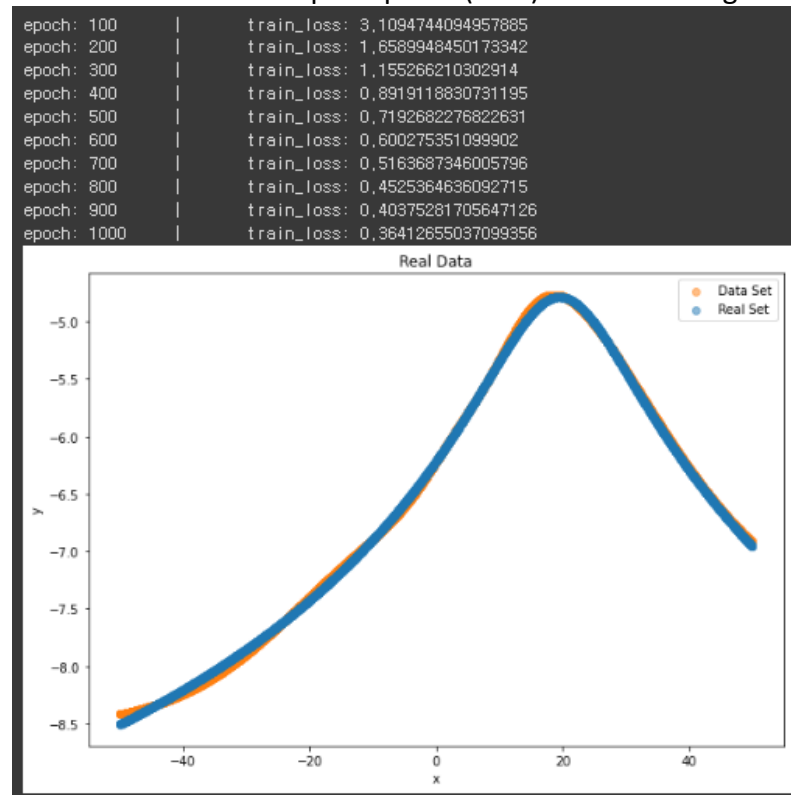
파일 선택 hw3_data.csv
• hw3_data.csv(text/csv) - 131370 bytes, last modified: 2022. 11. 12. - 100% done
Saving hw3_data.csv to hw3_data.csv
User uploaded file "hw3_data.csv" with length 131370 bytes

```

File Upload: Before cell execution, the given data set must be uploaded first.

6 Results

- Result of the multi-perceptron (MLP) model training and its visualization:



- Weights of each layer in the multi-layer perceptron (MLP) model:

```
Weights of the layer 0 :
[[-0.07154734 -0.25010079 -0.09304556 -0.26467937]]

Weights of the layer 1 :
[[ 0.49243165  0.86349245 -2.29479987  0.19519798  0.02376995 -2.38760376
 -1.73705278  0.99567615]
 [-0.92267985 -1.48479476  2.01044673 -0.57620045 -0.91136153  0.57317234
 -0.17654998 -1.10877551]
 [ 0.65788494 -0.15936802  0.62772905  1.05366754  0.82906435 -0.70789745
  0.99445584 -0.53008999]
 [ 0.86190573 -0.64687632  0.37545412  0.53709955 -1.29792269 -0.59242563
  0.80714257 -1.25389648]]

Weights of the layer 2 :
[[-0.36421011 -1.36749839  1.35051145  0.4128582 ]
 [ 2.15612457  0.04926947 -0.38637827 -1.19293467]
 [-0.44851023  1.26570238  0.4099762  -0.85585974]
 [-1.24891853 -0.39211  0.23257889 -1.1629634 ]
 [-0.8266972  0.77189211  0.67826065 -0.17803908]
 [ 0.39309549  0.18723519  0.58484636 -1.37944659]
 [-0.39853409  2.19786871  0.78792722 -0.05932982]
 [ 0.03455935 -2.55138246 -0.33984153  0.2809833 ]]

Weights of the layer 3 :
[[ 1.23616681]
 [-1.41637154]
 [-1.18215076]
 [ 0.88629956]]

Biases of the layer 0 :
[[ 1.76984172  3.00443938 -1.8229025  0.02456587]]

Biases of the layer 1 :
[[ 0.91839476 -0.60776367  0.22334154  0.2269187  0.65920391 -0.63071193
  0.2848928 -1.10733998]]

Biases of the layer 2 :
[[-0.77502832  0.51260964  1.43389616 -0.38492196]]

Biases of the layer 3 :
[[2.50443795]]
```

7 Evaluation

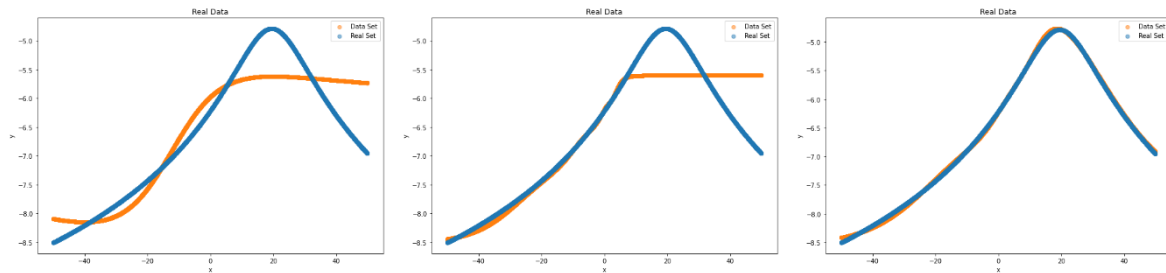


Figure 21 - Results of Predictions due to Different Number of Layers

Figure 21 shows the results of the predictions due to the different number of layers. The first case is the model with a single layer of 4 perceptron with a sigmoid activation function. The second case is the model with two layers of 4 and 8 perceptron with a sigmoid activation function. The last case is the model with three layers of 4, 8, and 4 perceptron that use the sigmoid activation function for the first two layers, and the identity activation function for the third layer. As we can see from the results, as the number of layers increases, the accuracy of the prediction increases.

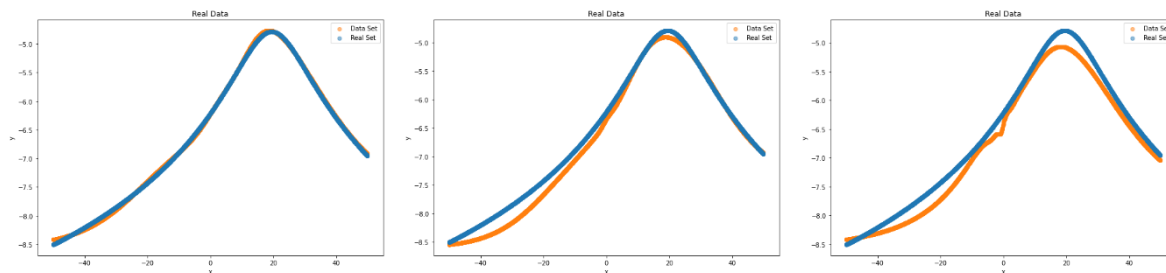


Figure 22 - Results of Predictions due to Different Configurations of Layers

Figure 22 shows the results of the predictions due to the different configurations of the layers. The first case is the model with three layers of 4, 8, and 4 perceptron with a sigmoid activation function for the first two layers and an identity layer for the third layer. The second case is the model with three layers of 8, 16, and 8 perceptron with a sigmoid activation function for the first two layers and an identity layer for the third layer. The last case is the model with three layers of 16, 32, and 16 perceptron with a sigmoid activation function for the first two layers and an identity layer for the third layer. As we can see from the results, as the configurations, which also mean the increment in the number of perceptron, change, the accuracy of prediction decreases.

In short, according to the results that we get from our MLP regressor, as the number of layers increases, and as the configuration gets complex, the prediction rate of the MLP model increases.

8 Conclusion

Multi-layer perceptron model is a neural network model that accumulates multiple layers and fully connects them. It consists of the input layer, multiple hidden layers, and an output layer. Except for the input nodes, each node in the hidden layers uses a non-linear activation function, such as sigmoid, ReLU, and hyperbolic tangent functions. In the case of the output layer, if the model is used for regression, it uses the identity function as an activation function. If the model is used for the classification, it uses non-linear activation functions. MLP model utilizes a supervised learning method called backpropagation for training.

In this paper, we have explained the concepts that are used to implement the multi-layer perceptron (MLP) regression model: perceptron model, activation functions, regressor with MLP, and backpropagation. We presented the neural network model for the function approximator, which is the implementation of MLP regressor by training the model using the backpropagation algorithm. After that, we compared the prediction rate and the plot of the prediction of the models with a different number of layers and different configurations of the layers. As a result, we get the conclusion that the prediction rate of the MLP regression model increases when the number of layers increases, and the configuration gets simpler.

By understanding this paper, we can understand the basic concepts of the multi-layer perceptron (MLP) model and the concepts that are used to implement the following model. Also, we can understand the correlation of the dense layers in the MLP models.

Citations

- [1] IBM Cloud Education. (n.d.). What are neural networks? IBM. Retrieved November 20, 2022, from <https://www.ibm.com/cloud/learn/neural-networks>
- [2] Lee, K. H. (n.d.). [Slide] 05. Neural Networks.
- [3] Multi-layer perceptron (MLP) diagram with four hidden layers and a ... (n.d.). Retrieved November 20, 2022, from https://www.researchgate.net/figure/Multi-Layer-Perceptron-MLP-diagram-with-four-hidden-layers-and-a-collection-of-single_fig1_334609713