DLP-Lab2

ILE 310553043 You-Pin, Chen

March 21, 2022

1 Introduction

In this assignment, we are asked to implement backpropagation from scratch which means we can not use deep learning framework like Pytorch or Tensorflow. Instead, we are available to apply some mathematical packages like Numpy. Backpropagation is a algorithm of gradient descent which we often make use of updating our model weights. By using backward direction, model weights will update efficiently, specially when the model is complex. When implementing backpropagation, main method is using chain rule to reach our goal, which is a formula that expresses the derivative of the composition of two differentiable functions f and g in terms of the derivatives of f and g. We will exhibit the details about it below. Complete code shows in github.

2 Experiment

2.1 Sigmoid function

Sigmoid function is one of the activation function which are often utilized for mapping the output value to [0, 1] if we hope for normalization value as our output. Through this kind of activation function, we will be able to get continuous variable and it will be helpful in deriving the derivative during backpropagation. But when the number of hidden layers increasing, it will bring about vanishing and exploding gradient problem. That is not our discussion today yet.

$$sigmoid : \sigma(x) = \frac{1}{1+e^{-x}} \tag{2.1.1}$$

and

$$\begin{aligned} \sigma'(x) &= \frac{d(1+e^{-x})^{-1}}{dx} \\ &= -(1+e^{-x})^2 \frac{d}{dx}(1+e^{-x}) \\ &= -(1+e^{-x})(1+e^{-x})(-e^{-x}) \\ &= \sigma(x)(1-\sigma(x)) \end{aligned} \tag{2.1.2}$$

```python
import numpy as np

# sigmoid function
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

# derivative of the sigmoid function
def derivative_sigmoid(x):
    return np.multiply(x, 1.0 - x)
```

2.2 Neural network

Neural network is based on a collection of connected units or nodes called artificial neurons, which loosely model the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a signal to other neurons. Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the input layer), to the last layer (the output layer), possibly after traversing the layers multiple times. In our network structure, we specified two hidden layers and I/O layer which contains different number of neurons that we will discuss later (Fig.1).

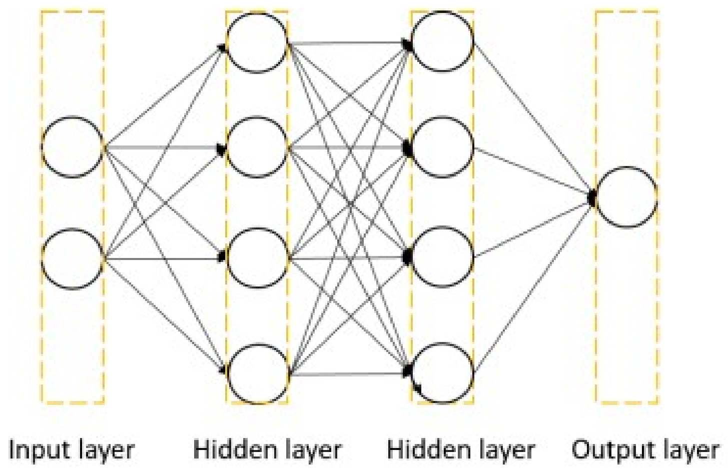


Figure 1: Artificial neural network with 2 hidden layers.

2.2.1 Neural Unit and Layer

The data input X is a vector which dimension also recognized as neural unit. Through Eq.2.2.1, after multiply with weight vector and add some bias, we will get a linear result. Usually, we will pass through a nonlinear activation function before going to next round.

$$Y = \sigma(W^T X + bias) \tag{2.2.1}$$

- ***__init__:***

  1. $layers$: Specify the number of neural units and layers.
  2. $w1$: Initialize the weights between input layer and first hidden layer.
  3. $w2$: Initialize the weights between two hidden layers.
  4. $w3$: Initialize the weights between second layer and output layer.
  5. b : Initialize the bias. Here we do not consider it. so we set 0.0.

```
import numpy as np

class NNet():
    def __init__(self, structure):
        self.layers = structure
        self.w1 = np.random.normal(0.0, 1.0, (self.layers[0], self.layers[1]))
        self.w2 = np.random.normal(0.0, 1.0, (self.layers[1], self.layers[2]))
        self.w3 = np.random.normal(0.0, 1.0, (self.layers[2], self.layers[3]))
        self.b = 0.0
```

2.3 Backpropagation

Backpropagation is a method used in artificial neural networks to calculate a gradient that is needed in the calculation of the weights to be used in the network. Backpropagation is a generalization of the delta rule to multilayered feedforward networks, made possible by using the chain rule to iteratively compute gradients for each layer. The backpropagation learning algorithm can be divided into two parts: propagation and weight update.

2.3.1 Propagation

To obtain the best solution, we need to propagation forward through the network to generate the output value. Once we acquire the prediction, we can calculate the loss between ground truth through the loss function like Mean Square Error (Eq.2.3.1).

$$MSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{2.3.1}$$

```python
def MSE(y, y_pred):
    '''
    y: grouth true
    y_pred: pred
    '''
    square_list = []
    for i in range(y.size):
        s = (y[i] - y_pred[i]) * (y[i] - y_pred[i])
        square_list.append(s)

    total = 0.0
    for j in range(len(square_list)):
        total += square_list[j]

    return total / (len(square_list))



def derivative_MSE(y, y_pred):
    '''
    Derivative of MSE
    '''
    return -2 * (y - y_pred) / y.shape[0]
```

Since our objective is minimize cost in loss function $L(\theta)$, we use gradient descent to update the model weights, propagation of the output activations back through the network. Nonetheless, we find out that it is difficult to calculate the derivative of loss function, so we use chain rule to reach our goal.

$$x \xrightarrow{w_1} z_1 \xrightarrow{\sigma(z_1)} a_1 \xrightarrow{w_2} z_2 \xrightarrow{\sigma(z_2)} a_2 \xrightarrow{w_3} z_3 \xrightarrow{\sigma(z_3)} \hat{y} \Longleftrightarrow L(\theta) \tag{2.3.2}$$

and

$$\frac{\delta L(\theta)}{\delta w_3} = \frac{\delta L(\theta)}{\delta \hat{y}} \cdot \frac{\delta \hat{y}}{\delta z_3} \cdot \frac{\delta z_3}{\delta w_3} \tag{2.3.3}$$

$$\frac{\delta L(\theta)}{\delta w_2} = \frac{\delta L(\theta)}{\delta \hat{y}} \cdot \frac{\delta \hat{y}}{\delta z_3} \cdot \frac{\delta z_3}{\delta a_2} \cdot \frac{\delta a_2}{\delta z_2} \cdot \frac{\delta z_2}{\delta w_2} \tag{2.3.4}$$

$$\frac{\delta L(\theta)}{\delta w_1} = \frac{\delta L(\theta)}{\delta \hat{y}} \cdot \frac{\delta \hat{y}}{\delta z_3} \cdot \frac{\delta z_3}{\delta a_2} \cdot \frac{\delta a_2}{\delta z_2} \cdot \frac{\delta z_2}{\delta a_1} \cdot \frac{\delta a_1}{\delta z_1} \cdot \frac{\delta z_1}{\delta w_1} \tag{2.3.5}$$

2.3.2 Weight update

At the start, we set our weights $w1$, $w2$, $w3$ randomly. Now, because of chain rule, we have $\frac{\delta L(\theta)}{\delta w}$ easily and are able to make use of updating our weights w. Hyperparameter ***learning rate*** (η) decide how far we are going to update.

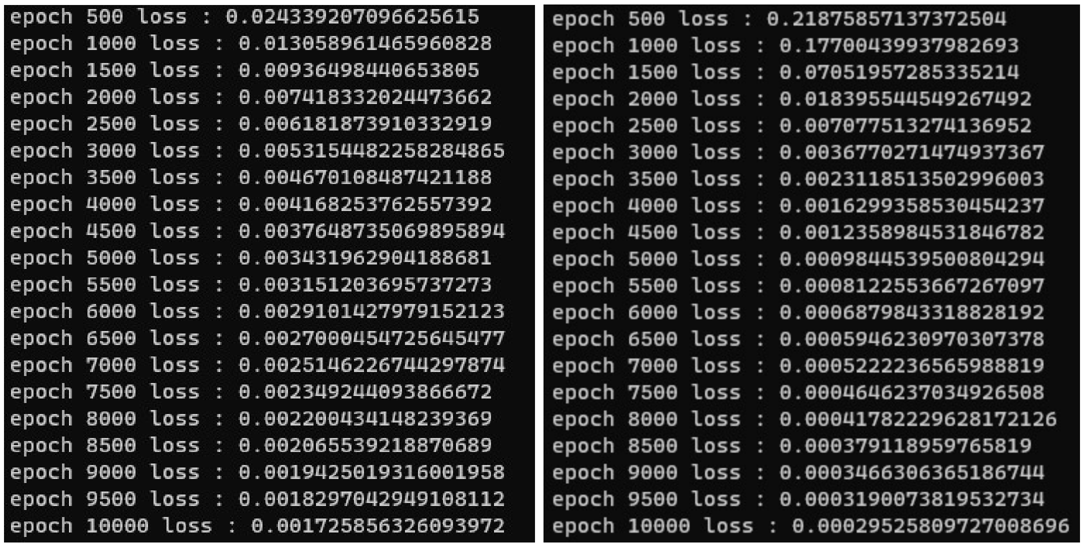
$$W^(t+1) = W^(t) - \eta * \frac{\delta L(\theta)}{\delta w} \tag{2.3.6}$$

- ***forward:***
  1. x : Input data. In this assignment, we have Linear and XOR datasets.
  2. $z1$: Output of $multiply(x, w1) + bias$.
  3. $a1$: Output of $\sigma(z1)$.
  4. $z2$: Output of $multiply(a1, w2) + bias$.
  5. $a2$: Output of $\sigma(z2)$.
  6. $z3$: Output of $multiply(a2, w3) + bias$.
  7. y_pred : Final prediction. Output of $\sigma(z3)$.
- ***backprop:***
  1. d_L_w1 : Propagation by chain rule shows in Eq.2.3.5
  2. d_L_w2 : Propagation by chain rule shows in Eq.2.3.4
  3. d_L_w3 : Propagation by chain rule shows in Eq.2.3.3

```python
import numpy as np

def forward(self, x):
    self.x = x
    self.z1 = (self.x @ self.w1) + self.b
    self.a1 = sigmoid(self.z1)
    self.z2 = (self.a1 @ self.w2) + self.b
    self.a2 = sigmoid(self.z2)
    self.z3 = (self.a2 @ self.w3) + self.b
    self.y_pred = sigmoid(self.z3)
    return self.y_pred


def backprop(self, y, y_pred):
    '''
    Backpropagation
    '''
    d_loss = derivative_MSE(y, y_pred)
    d_y_pred = derivative_sigmoid(self.y_pred)
    d_a2 = derivative_sigmoid(self.a2)
    d_a1 = derivative_sigmoid(self.a1)

    self.d_L_w3 = self.a2.T @ (d_y_pred * d_loss)
    self.d_L_w2 = self.a1.T @ (d_a2 * ((d_y_pred * d_loss) @ self.w3.T))
    self.d_L_w1 = self.x.T @ (d_a1 * ((d_a2 * ((d_y_pred * d_loss) @ self.w3.T)) @ self.w2.T))


def optimize(self, lr=1):
    '''
    Optimize weights by using SGD optimizer
    '''
    self.w1 = self.w1 - lr * self.d_L_w1
    self.w2 = self.w2 - lr * self.d_L_w2
    self.w3 = self.w3 - lr * self.d_L_w3
```

3 Results

3.1 Screenshot and comparison figure



```
epoch 500 loss : 0.024339207096625615
epoch 1000 loss : 0.013058961465960828
epoch 1500 loss : 0.00936498440653805
epoch 2000 loss : 0.007418332024473662
epoch 2500 loss : 0.006181873910332919
epoch 3000 loss : 0.0053154482258284865
epoch 3500 loss : 0.004670108487421188
epoch 4000 loss : 0.004168253762557392
epoch 4500 loss : 0.0037648735069895894
epoch 5000 loss : 0.003431962904188681
epoch 5500 loss : 0.003151203695737273
epoch 6000 loss : 0.0029101427979152123
epoch 6500 loss : 0.0027000454725645477
epoch 7000 loss : 0.0025146226744297874
epoch 7500 loss : 0.002349244093866672
epoch 8000 loss : 0.002200434148239369
epoch 8500 loss : 0.002065539218870689
epoch 9000 loss : 0.0019425019316001958
epoch 9500 loss : 0.0018297042949108112
epoch 10000 loss : 0.001725856326093972
```

(a) Linear dataset

```
epoch 500 loss : 0.21875857137372504
epoch 1000 loss : 0.17700439937982693
epoch 1500 loss : 0.07051957285335214
epoch 2000 loss : 0.018395544549267492
epoch 2500 loss : 0.007077513274136952
epoch 3000 loss : 0.0036770271474937367
epoch 3500 loss : 0.0023118513502996003
epoch 4000 loss : 0.0016299358530454237
epoch 4500 loss : 0.0012358984531846782
epoch 5000 loss : 0.0009844539500804294
epoch 5500 loss : 0.0008122553667267097
epoch 6000 loss : 0.0006879843318828192
epoch 6500 loss : 0.0005946230970307378
epoch 7000 loss : 0.0005222236565988819
epoch 7500 loss : 0.0004646237034926508
epoch 8000 loss : 0.00041782229628172126
epoch 8500 loss : 0.000379118959765819
epoch 9000 loss : 0.0003466306365186744
epoch 9500 loss : 0.0003190073819532734
epoch 10000 loss : 0.00029525809727008696
```

(b) XOR dataset

Figure 2: Comparison between two datasets. $Epoch : 10000$, $LR : 1$

3.2 Accuracy

```
'''
Prediction with 100 data from linear dataset.
'''
[[9.99837146e-01]
 [1.46848013e-02]
 [9.99889197e-01]
 [9.99920911e-01]
 [3.64367074e-05]
 [6.08949084e-05]
 [9.99915938e-01]
 [9.16238924e-01]
 [9.99925544e-01]
 [9.99257167e-01]
 [1.21186746e-04]
 [9.99912502e-01]
 [9.99922847e-01]
 [9.99925469e-01]
 [7.79059007e-04]
 [9.99922511e-01]
 [9.99593406e-01]
 [9.99924085e-01]
 [8.19549821e-01]
 [6.08234908e-05]
 [9.98635912e-01]
 [3.56678898e-05]
 [8.08136308e-01]
 [2.77496372e-03]
 [9.93098125e-01]
 [3.27959963e-04]
 [3.37161348e-05]
```

```
'''
Prediction with 100 data from XOR dataset.
'''
[[0.02133344]
 [0.98981418]
 [0.01878599]
 [0.98987787]
 [0.0171231 ]
 [0.98998373]
 [0.01611708]
 [0.98991977]
 [0.01545536]
 [0.97265167]
 [0.01492339]
 [0.01442891]
 [0.95319625]
 [0.01395138]
 [0.9938795 ]
 [0.013497  ]
 [0.99571768]
 [0.01307661]
 [0.99611034]
 [0.01269857]
 [0.99623912]]
Epoch:  10000
Learning Rate: 1
Test Accuracy (%): 100.00%
```

```
[3.05824419e-02]
[4.30046518e-05]
[7.85935019e-05]
[2.78732509e-02]
[3.80546679e-05]
[9.99846311e-01]
[3.46738676e-05]
[3.45233892e-05]
[3.80909676e-05]
[4.44833815e-05]
[3.49209942e-05]
[1.59835254e-04]
[1.26116289e-04]
[9.99341964e-01]
[9.99924388e-01]
[5.31410386e-05]
[4.15208369e-05]
[9.99915603e-01]
[9.99915562e-01]
[9.99923853e-01]
[7.99553531e-04]
[3.69436906e-05]
[3.40155521e-05]
[4.24924201e-05]
[9.99890766e-01]
[9.99913690e-01]
[3.01404707e-01]
[9.99925106e-01]
[9.99918665e-01]
[4.28071239e-05]
[4.17967016e-05]
[5.27407466e-05]
[1.06620887e-04]
[7.66934694e-05]
[5.07320271e-05]
[9.99859596e-01]
[9.99907842e-01]
[4.50827246e-05]
[1.07079452e-04]
[9.99830630e-01]
[9.49758962e-01]
[9.98743782e-01]
[7.87953587e-05]
[3.75858555e-05]
[9.99920973e-01]
[9.91661116e-04]
[9.99925547e-01]
[9.99907172e-01]
[9.99925066e-01]
[9.99921092e-01]
[9.99925107e-01]
[6.94158121e-05]
[4.85103473e-05]
[6.73661410e-05]
[9.99785555e-01]
[4.93956516e-04]
[9.99915864e-01]
[3.70187831e-05]
[3.98676694e-05]
[1.19595414e-04]
[3.67407671e-05]
[9.99923724e-01]
```

```
 [3.52245026e-05]
 [9.99921208e-01]
 [4.80583135e-05]
 [4.96710417e-05]
 [9.95460039e-01]
 [4.18493944e-05]
 [1.08841375e-04]
 [9.99925269e-01]
 [6.53389576e-05]
 [9.66457188e-01]
 [9.99917786e-01]]
Epoch: 10000
Learning Rate: 1
Test Accuracy (%): 100.00%
```

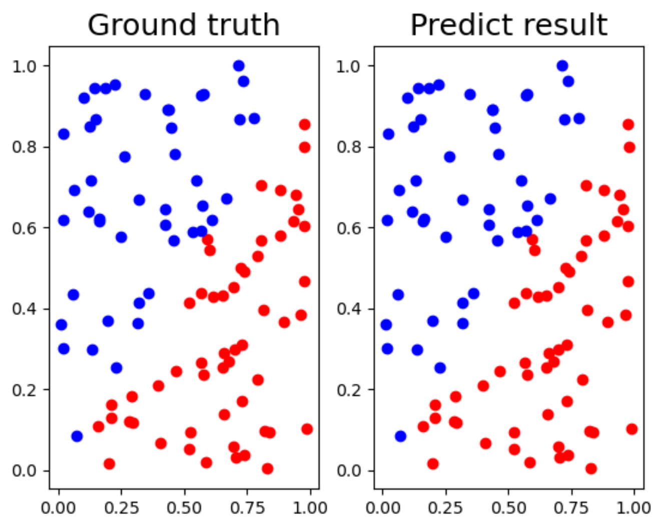


Figure 3: Result from linear dataset. $Epoch: 10000$, $LR: 1$, $Acc: 100\%$

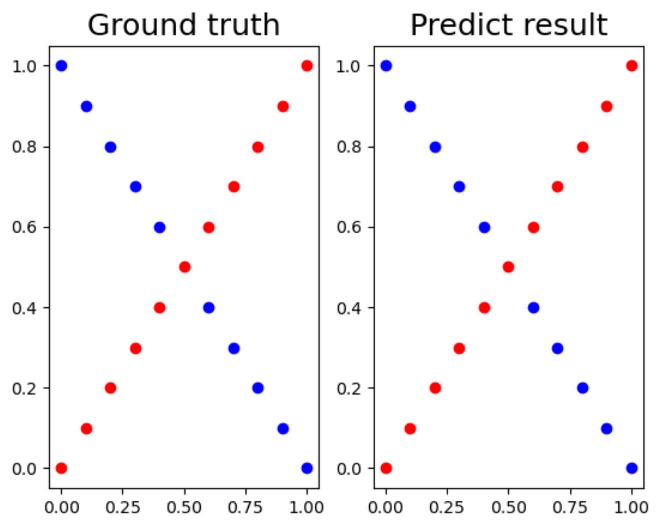


Figure 4: Result from XOR dataset. $Epoch: 10000$, $LR: 1$, $Acc: 100\%$

3.3 Learning curve

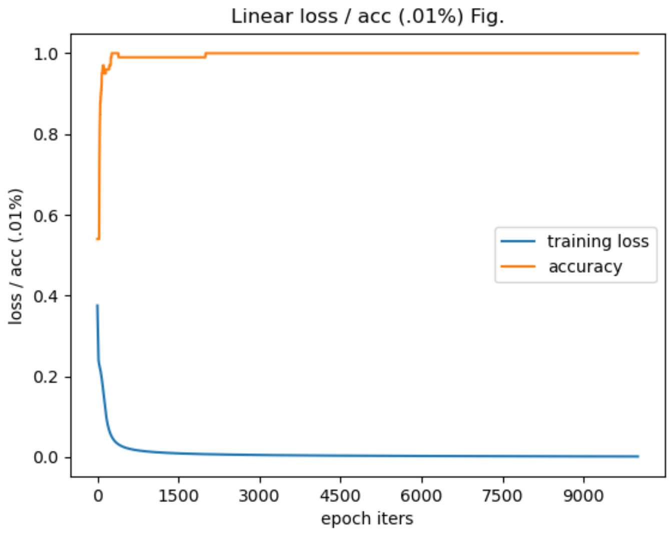


Figure 5: Loss / Acc.(.01%) from linear dataset. $Epoch : 10000$, $LR : 1$

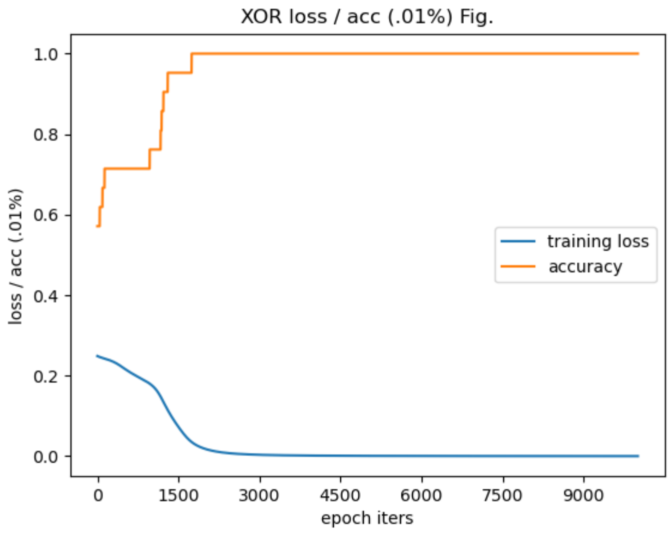


Figure 6: Loss / Acc.(.01%) from XOR dataset. $Epoch : 10000$, $LR : 1$

4 Discussion

4.1 Different learning rates

In previous experiments, our hyperparameters were fixed by setting epoch to 10000, learning rate to 1.0. The reason to do so is that we found out this combination will lead us to the best result. We reset learning rate to the value that bigger than 1.0, they all came out 100% accuracy. Since we cut down the value of learning rate to $[0.001, 0.01]$, the accuracy, shows in Fig.7, decrease instead of 100%. We though that the main reason is on the data, because they all came from same random seed. For this situation, can we called it ”overfitting”?

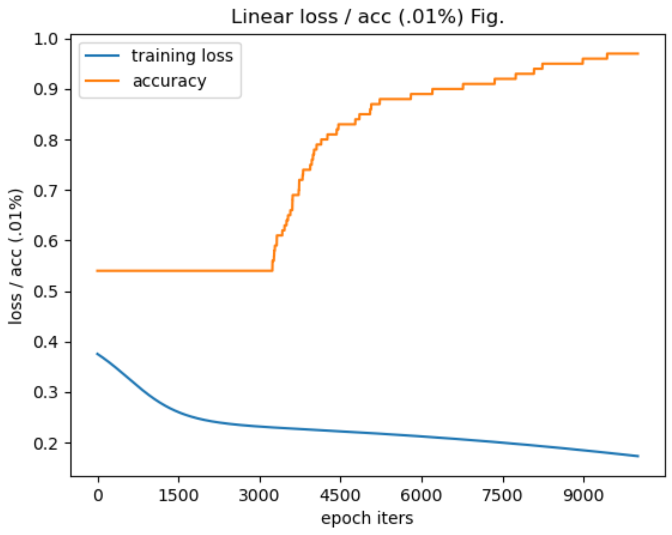


Figure 7: Decrease the learning rate, the model update more slowly.

4.2 Different numbers of hidden units

In general, more neurons will lead the model more powerful. But if the data is not enough, not only model can not learn the complete feature, but also cost lots of time for nothing. So the relationship between data and model complexity is crucial. Previously, we just testing numerous experiments in model structure with $[2, 4, 4, 1]$ neural units in each layer respectively. Continue the last experiment in Fig.7, we converted the number of neuron in hidden layer from 4 to 10. The result shows in Fig.8, by increasing the number of neuron, the model convergence rapidly compare to original structure.

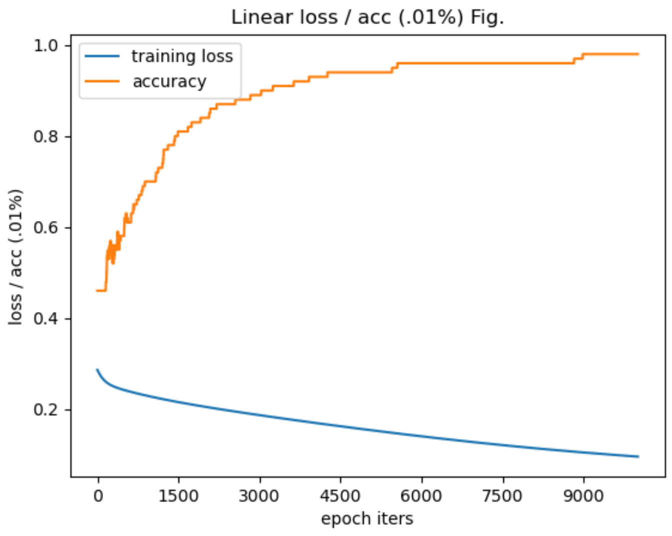


Figure 8: Converge raidly with the number of neuron increasing.

4.3 Without activation functions

In Eq.2.2.1 we realize that if the model function without nonlinear activation, every neuron will only be performing a linear transformation on the inputs using the weights and biases. Although linear transformations make the neural network simpler, but this network would be less powerful and will not be able to learn the complex patterns from the data. It turns out to be a simple linear regression problem (Fig.9).

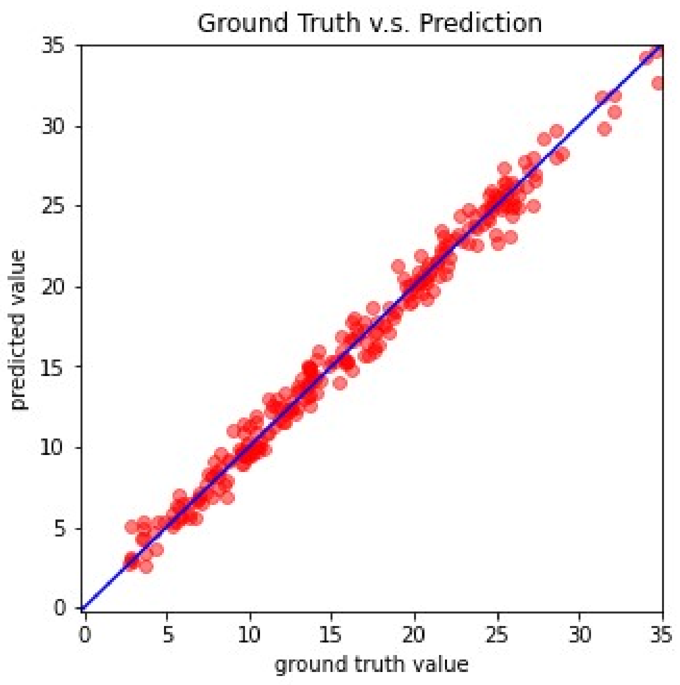


Figure 9: Simple linear regression problem without activation function.

5 Extra

5.1 Activation function

In chapter 4.3, we knew how crucial activation function is. Except for sigmoid function (Eq.2.1.1), there are many other kinds. Rectified Linear Unit (Eq.5.1.1) also known as *ReLU*, is one of the popular activation function. It turns the value to 0 if it is negative. Unlike the derivative of sigmoid function will suffer vanishing gradient problem, The derivative of ReLU is 1 for values larger than zero. Because multiplying 1 by itself several times still gives 1, this basically addresses the vanishing gradient problem.

$$ReLU : f(x) = max(0, x) \tag{5.1.1}$$

```
def ReLu(x):
    return x * (x > 0)


def derivative_ReLu(x):
    if x >= 0:
        return 1
    else:
        return 0
```

Nonetheless, the negative component of the ReLU function cannot be discriminated against because it is 0. As a result, the derivative of negative values are simply set to 0. It is not a big deal with forward propagation, but during backpropagation, the gradient is 0. This problem we called *Dead ReLU*. To solve this problem, one variance of ReLU which called *Leaky ReLU* (Eq.5.1) came out. Leaky ReLU is defined to address this problem. Instead of defining the ReLU function as 0 for negative values of x, we define it as an extremely small linear component of x (Fig.10).

$$LeakyReLU : f(x_i) = \begin{cases} x_i & x_i >= 0 \\ a_i \cdot x_i & x_i < 0 \end{cases}$$

(5.1.2)

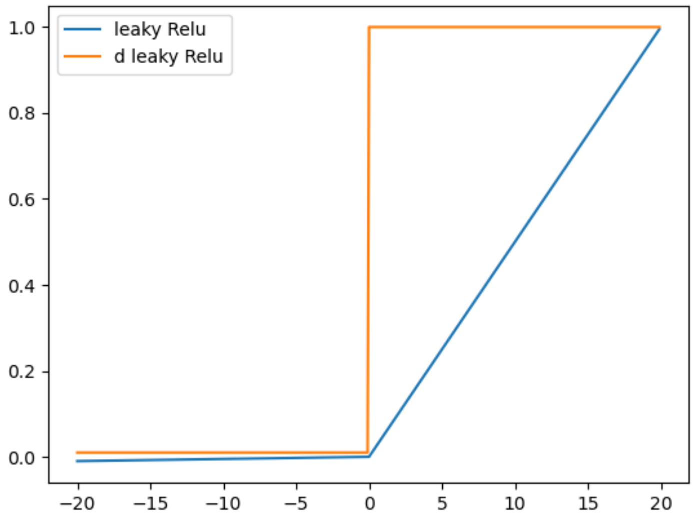


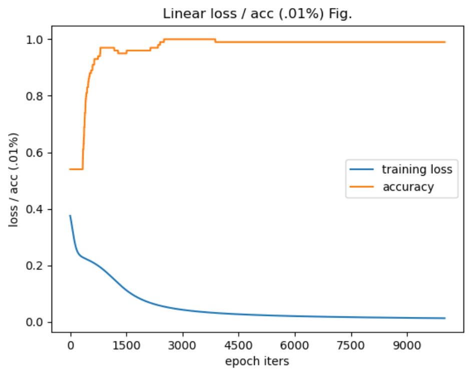
Figure 10: Leaky ReLU and derivative of Leaky ReLU

```
'''
Leaky ReLU
'''

a_i = 0.01

def leaky_Relu(x):
    if x >= 0:
        return x
    else:
        return a_i * x


def derivative_leaky_Relu(x):
    if x >= 0:
        return 1
    else:
        return a_i
```

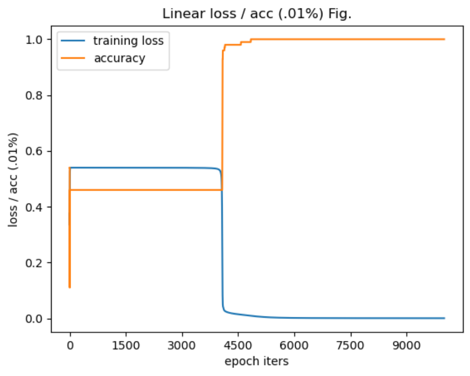
5.2 Optimizer

Backpropagation is the way to optimize weights efficiently. The procedure is calculating gradient and use it to update the weights of previous round. In previous experiments, we express that if we want to optimize the weights, we can through Eq.2.3.6 to achieve. The function is called Stochastic Gradient Decent which is uncomplicated to implement and is widely use when the task are not quite complex. Along with the model structure getting bigger, many different kind of optimizeies are proposed. Momentum (Eq.5.2.1) is a optimizer that the learning speed will be faster in the dimension of the same direction, and the learning speed will be slower when the direction changes. Through β value to control the strength of speed. Usually, we will set β value to 0.9 or 0.99.

$$\begin{aligned} V_t &= \beta V_{t-1} - \eta * \frac{\delta L(\theta)}{\delta w} \\ W &= W + V_t \end{aligned} \tag{5.2.1}$$



(a) SDG, $Loss: 0.01305$, $Acc: 99\%$



(b) Momentum, $Loss: 0.00036$, $Acc: 100\%$

Figure 11: Comparison between two optimizeies. $Epoch: 10000$, $LR: 0.1$

```
lr = 0.01
beta = 0.9

def optimize(self, lr):
    if args.o == 'sgd':
        self.w1 = self.w1 - lr * self.d_L_w1
        self.w2 = self.w2 - lr * self.d_L_w2
        self.w3 = self.w3 - lr * self.d_L_w3
    elif args.o == 'momentum':
        self.v1 = beta * self.v1 - lr * self.d_L_w1
        self.v2 = beta * self.v2 - lr * self.d_L_w2
        self.v3 = beta * self.v3 - lr * self.d_L_w3

        self.w1 = self.w1 + self.v1
        self.w2 = self.w2 + self.v2
        self.w3 = self.w3 + self.v3
```