Deep Learning and Practice Lab1: Back-propagation

Zhi-Yi Chin
joycenerd.cs09@nycu.edu.tw

September 17, 2021

1 Introduction

In this lab, we will need to understand and implement simple neural networks with forwarding pass and backward propagation using only two hidden layers. The simple neural network we are going to implement is a feedforward neural network.

A feedforward neural network is an artificial neural network wherein connections between the nodes do not form a cycle. As such, it is different from its descendant: recurrent neural networks. The feedforward neural network was the first and most straighforward type of artificial neural network devised. In this network, the information moves in only one direction – forward – from the input nodes, hidden nodes (if any), and output nodes. There are no cycles or loops in the network.

2 Experiment Setups

2.1 Sigmoid functions

We use the sigmoid function as our activation function. To start with, let us take a look at the sigmoid function:

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{1}$$

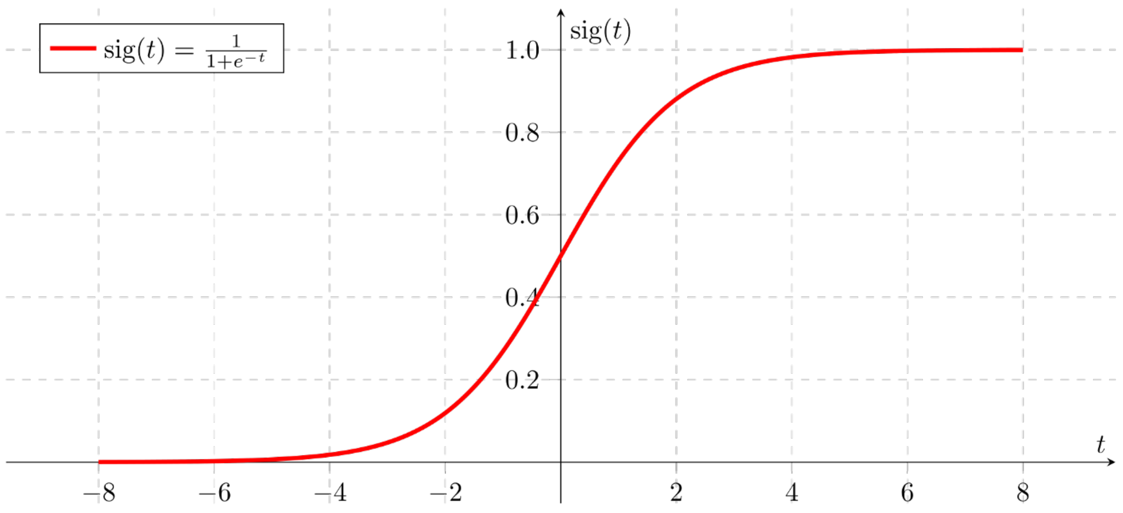


Figure 1: Sigmoid function

Figure 1 shows that the given number n, the sigmoid function would map that number between 0 and 1. As the value on n gets larger, the value of the sigmoid function gets closer and closer to 1, and as n gets smaller, the value of the sigmoid function is getting closer and closer to 0.

Let us start deriving the sigmoid function:

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

The main reason we use sigmoid as our activation function is because of our data. Our data only has two categories we name as 0 and 1. If we use sigmoid as our activation function, we can guarantee that our output will be a value between 0 and 1. If the number is over 0.5, we will classify that data point as 1, and if the number is less than 0.5, we will classify it as 0.

Below if the code for sigmoid and derivation of sigmoid:

```
def sigmoid(x):
    # Sigmoid function.
    return 1 / (1 + np.exp(-x))
```

Listing 1: Sigmoid function

```python
def der_sigmoid(y):
    # First derivative of Sigmoid function.
    return y * (1 - y)
```

Listing 2: Derivative of the sigmoid function

2.2 Neural network

Our neural network structure only has two hidden layers, so this is a simple feedforward network. Our input layer size is two because each input data point has two coordinates: x and y coordinates. We define the first and second hidden layer sizes; in the experiment result, we will show that different hidden layer sizes did affect the accuracy. We have weight and bias for each layer, and after multiplying the weights and adding the bias, we will use go into the activation function, which is the sigmoid function. The output layer size is one which is only a value between 0 and 1. This is known as the forward pass. Figure 2 shows our simple model structure.

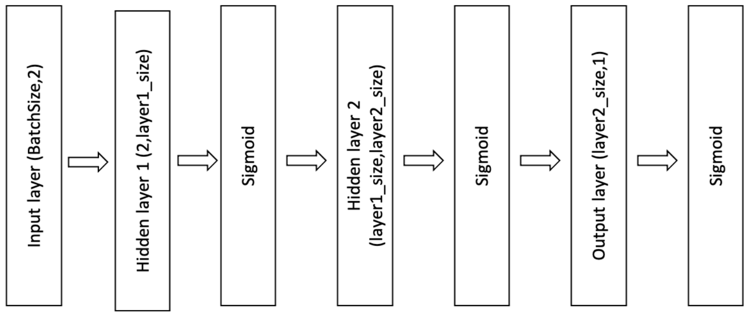

Input layer (BatchSize,2)
Hidden layer 1 (2,layer1_size)
Sigmoid
Hidden layer 2 (layer1_size,layer2_size)
Sigmoid
Output layer (layer2_size,1)
Sigmoid

Figure 2: Model architecture

Below is the code for Model initialization and forward pass:

```
class Model:
    def __init__(self, hidden_size, epochs, lr):
        """
        Feedforward network with 2 hidden layers
        :parma hidden_size: two hidden layers neurons (layer1,layer2)
        :param epochs: num of training epochs
        """
        self.epochs = epochs

        # Model parameters initialization
        input_size = 2
        output_size = 1
        self.lr = lr
        self.initial_lr=lr
        self.momentum = 0.9
        (layer1,layer2)=hidden_size
        self.layer1=layer1
        self.layer2=layer2

        self.w1 = np.random.randn(input_size, layer1)
        self.w2 = np.random.randn(layer1, layer2)
        self.w3 = np.random.randn(layer2, output_size)
        self.b1 = np.zeros((1, layer1))
        self.b2 = np.zeros((1, layer2))
        self.b3 = np.zeros((1, output_size))

        self.v_w1 = np.zeros((input_size, layer1) )
        self.v_w2 = np.zeros((layer1, layer2))
        self.v_w3 = np.zeros((layer2, output_size))
        self.v_b1 = np.zeros((1, layer1))
        self.v_b2 = np.zeros((1, layer2))
        self.v_b3 = np.zeros((1, output_size))


    def forward(self, inputs):
        # Forward pass
        self.input = inputs
        self.a1    = sigmoid(np.dot(self.input, self.w1)+self.b1)
        self.a2    = sigmoid(np.dot(self.a1, self.w2)+self.b2)
        output     = sigmoid(np.dot(self.a2, self.w3)+self.b3)

        return output
```

Listing 3: Derivative of the sigmoid function

2.3 Backpropagation

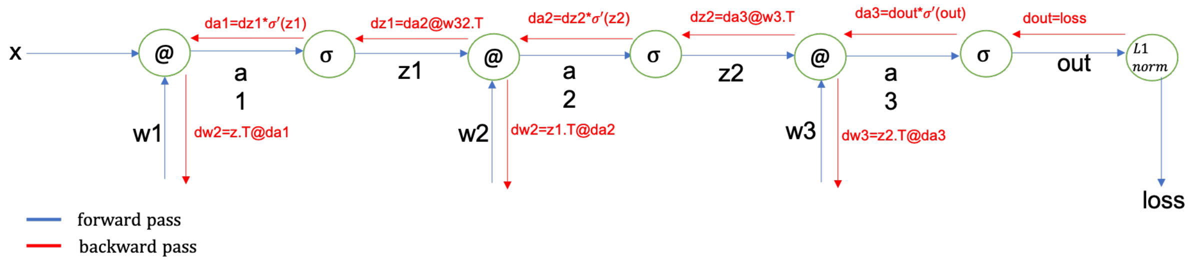


Figure 3: Both forward procedure and backward procedure.

Figure 3 shows both the forward pass and backward pass. Because this is a simple feedforward network, so it is easy to show the whole procedure step by step. In the forward pass, we go from input x to output and loss. As for backward pass, we do it in a reverse way. We go from loss back to input x. In the backward pass we are doing SGD. We set the batch size to be 1 because we create the data so we know that our data doesn't contain outliers, so using 1 as our batch size won't affect the end result. Besides learning rate, we also define momentum which we set to 0.9.

The code for the backward pass is as below:

```python
class Model():
    def backward(self):
        # backward pass
        dout    = self.error
        grad_a3    = np.multiply(dout, der_sigmoid(self.output))
        grad_w3 = np.dot(self.a2.T, grad_a3)
        grad_b3 = np.sum(grad_a3, axis=0)

        grad_z2    = np.dot(grad_a3, self.w3.T)
        grad_a2    = np.multiply(grad_z2, der_sigmoid(self.a2))
        grad_w2 = np.dot(self.a1.T, grad_a2)
        grad_b2 = np.sum(grad_a2, axis=0)

        grad_z1    = np.dot(grad_a2, self.w2.T)
        grad_a1    = np.multiply(grad_z1, der_sigmoid(self.a1))
        grad_w1 = np.dot(self.input.T, grad_a1)
        grad_b1 = np.sum(grad_a1, axis=0)


        self.v_w1 = self.momentum * self.v_w1 + self.lr * grad_w1
        self.v_w2 = self.momentum * self.v_w2 + self.lr * grad_w2
        self.v_w3 = self.momentum * self.v_w3 + self.lr * grad_w3
        self.v_b1 = self.momentum * self.v_b1 + self.lr * grad_b1
        self.v_b2 = self.momentum * self.v_b2 + self.lr * grad_b2
        self.v_b3 = self.momentum * self.v_b3 + self.lr * grad_b3

        self.w1 -= self.v_w1
        self.w2 -= self.v_w2
        self.w3 -= self.v_w3
        self.b1 -= self.v_b1
        self.b2 -= self.v_b2
        self.b3 -= self.v_b3
```

Listing 4: Backward pass

3 Experimental Results

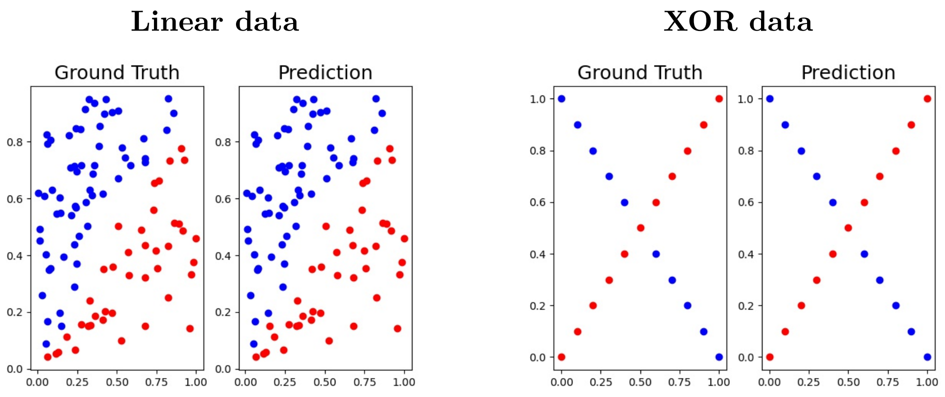


Table 1: In order to get the perfect results here for linear data we set the learning rate to 0.01, and for xor data, we set the learning rate to 0.01. For both data our model hidden size if (512,32).

lr	loss	acc
0.1	**0.0762**	**0.98**
0.01	0.3861	0.88
0.001	0.4995	0.6

Table 2: Linear data with different learning rates. The hidden size here is fix to (5,10).

hidden_size	loss	acc
(5, 10)	0.3861	0.88
(64, 128)	0.1379	0.98
(512, 32)	**0.123**	**0.99**

Table 3: Linear data with different hidden layer size. The learning rate is fixed to 0.01.

lr	loss	acc
0.1	**0.4879**	**0.6667**
0.01	0.4993	0.5714
0.001	0.4962	0.5238

Table 4: XOR data with different learning rates. The hidden size is fixed to (5, 10).

hidden_size	**loss**	**acc**
(5, 10)	0.4879	0.6667
(64, 128)	0.2141	1
(512, 32)	**0.2049**	**1**

Table 5: XOR data with different hidden layer size. The learning rate is fixed to 0.1.

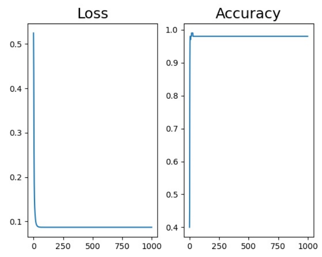
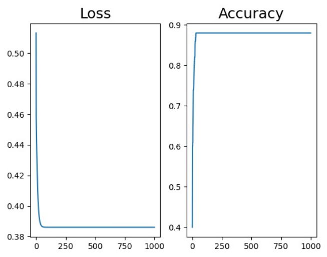
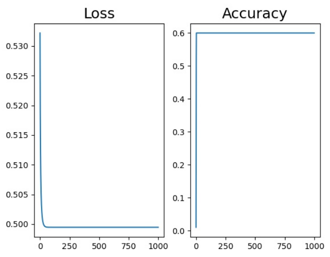
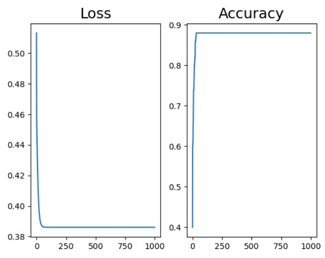
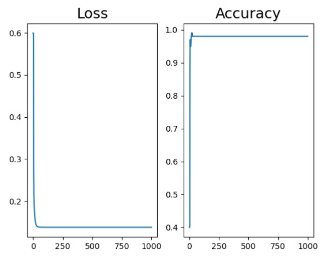
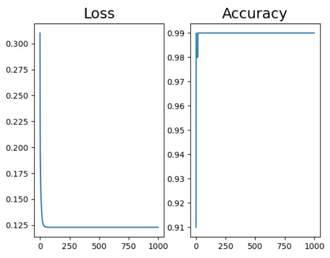
lr	0.1	0.01	0.001
			
hidden_size	(5, 10)	(64, 128)	(512, 32)
			

Table 6: Linear data loss and accuracy curve. The setting has been mentioned in Table 2 and 3.

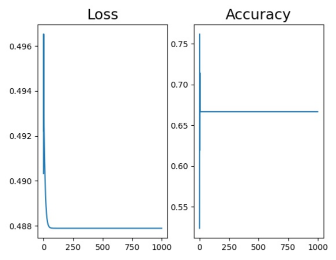
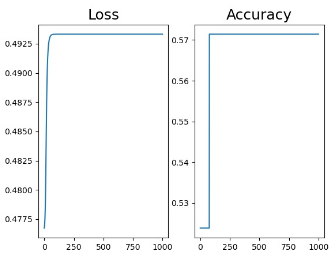
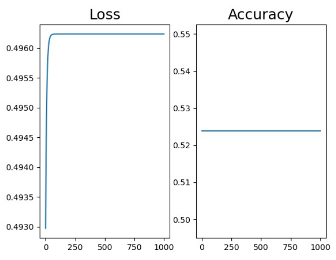
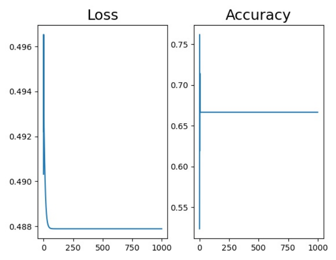
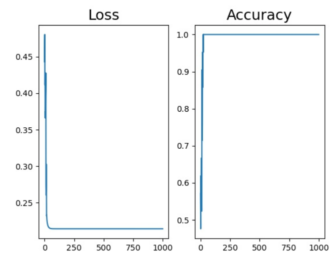
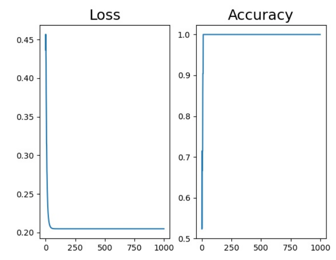
lr	0.1	0.01	0.001
			
hidden_size	(5, 10)	(64, 128)	(512, 32)
			

Table 7: XOR data loss and accuracy curve. The setting has been mentioned in Table 4 and 5.

4 Discussion

When we fixed the hidden layer size to (5,10), we found out that the smaller the learning rate, the worst the results became. Both linear and xor data show the best results when the learning rate is set to 0.1.

We fixed the learning rate when we are trying out different hidden layer sizes. We fixed the learning rate of linear data to 0.01. At first, we set the learning rate to 0.1 since it got the best result when trying out different learning rates. However, the outcome becomes worst when using more hidden units, so we change the learning rate to 0.01. In xor data, we fixed the learning rate to 0.1 when trying out different hidden units. Both linear data and xor data show that when increasing the hidden units, the higher the accuracy. We can find a drastic increase in xor data. When the hidden units are (5, 10), the accuracy is only 66.67%. However, when increase the hidden units to (64, 128), the accuracy can rise to 100%.

We also tried to run our model without the activation function. It will have output values in the first few epochs, but after a few epochs the output become NaN. We think the reason is because when we are doing multiplication and addition the value we accumulate to either very large number or very small number, this will cause the gradient to explode or vanishing. By using activation function we can guarantee our values are always in between 0 or 1, this can prevent gradient exploding.