Summer 2023 DLP Lab1

312552017 董家鳴

1. Introduction (20%)

In this lab, I implemented fully-connected neural networks with forwarding pass and back-propagation using two hidden layers from scratch. For the input data, we need to classify two kinds of datasets, linear data, and XOR data to tell the label of them.

Since the output is either 0 or 1, there will be two ways to classify it, logistic regression and classification. I will mainly provide the result of logistic regression in this report which is required in this lab, and provide the result of classification in the "Discussion" section.

2. Experiment setups (30%)

A. Sigmoid functions

The sigmoid function will be used as the activation in the output layer, and the function definition is denoted below

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

And since we will do the back-propagation, we need the derivative of the sigmoid function,

$$\begin{aligned}
\frac{d}{dx}\sigma(x) &= \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] \\
&= \frac{d}{dx}\left(1+e^{-x}\right)^{-1} \\
&= -\left(1+e^{-x}\right)^{-2}\left(-e^{-x}\right) \\
&= \frac{e^{-x}}{\left(1+e^{-x}\right)^2} \\
&= \frac{1}{1+e^{-x}} \cdot \frac{e^{-x}}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \cdot \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) \\
&= \frac{1}{1+e^{-x}} \cdot \left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \sigma(x) \cdot (1-\sigma(x))
\end{aligned}$$

B. Neural network

For each layer, I've defined input, z (output of linear combination), weights, biases, activation function, a (output of activation function), gradient, and partial cost function over z.

The next thing needed to do is to create layers with the value of weights and biases, their dimension of input and output, activation, and the gradient. Note that we're going to do the matrix multiplication, so the number of neurons should be the input dimension of weights and biases. Figure 1. shows the initialization of a layer.

```
def add_dense(self, units, input_dim="", activation=""):
    """
    Initialize a new layer with weights and biases including its dimension and value
    """

    # input dimension
    if self.input_dim == 0 and input_dim == "":
        raise Exception("input_dim is required")
    elif self.input_dim == 0:
        self.input_dim = input_dim

    # init layer
    layer = "W" + str(len(self.layers) + 1)
    self.layers.append(layer)

    self.init_dense(self.input_dim, units, activation)
    self.input_dim = units
def init_dense(self, input_dim, output_dim, activation):
    '''
    Randomly assign weights and biases to the newest layer,
    assign activation function,
    and initalize the gradient and momentum
    '''
    # newest layer
    layer = self.layers[-1]
    # random between 0 and 1
    np.random.seed(0)
    self.weights[layer] = (np.random.rand(output_dim, input_dim))
    self.biases[layer] =  (np.random.rand(output_dim, 1))
```

Figure 1: Initialize a layer

After setting up all the layers, the last thing to do is forward the input and get the output of layers, i.e., a and z mentioned above.

C. Backpropagation (Refer to [1])

First, we need to know that the gradient is denoted in Figure 2. The right-hand part of it is already calculated in the forward part. So in this part, we need to calculate the partial cost function over z in each layer. To calculate the partial cost over z, we need to use the derivative of the activation function, the loss function, and weights in each layer. By following the equation inside the green box in Figure 2., we can get all the partial cost over z in each layer.

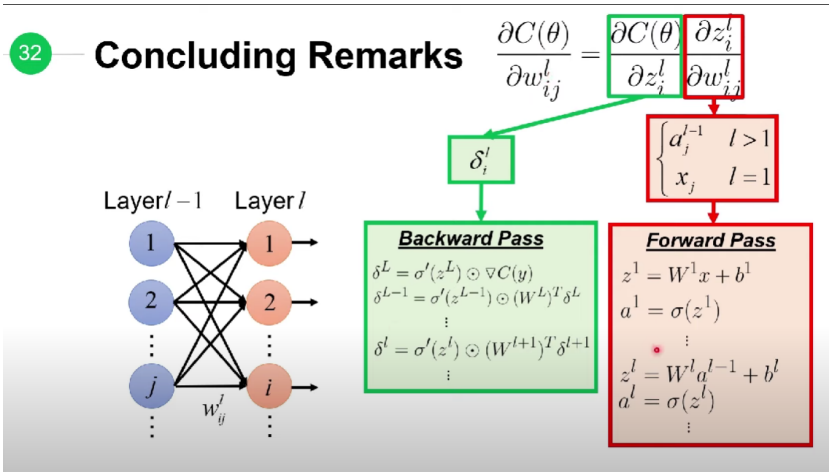

Figure 2: Initialize a layer

3. Results of your testing (20%)

B. Linear dataset, accuracy 100% The following Figure 3 and Figure 4 shows the result of the prediction

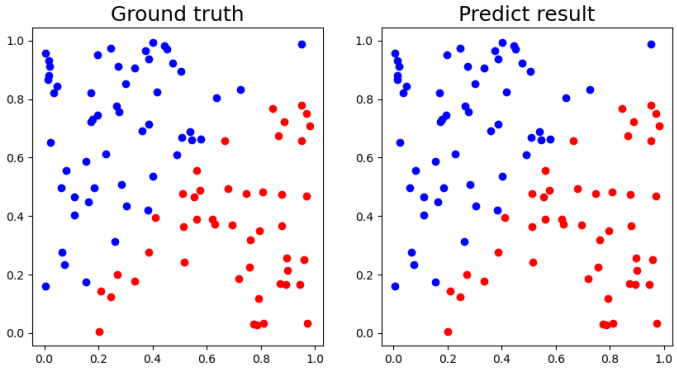

Figure 3: A. Screenshot and comparison figure of linear dataset

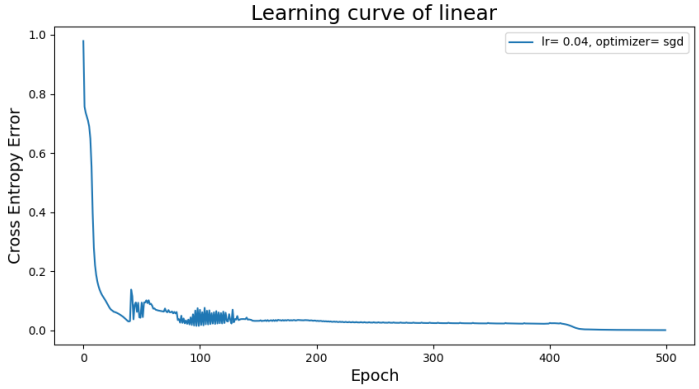

Figure 4: C. Learning curve of linear dataset

B. XOR dataset, accuracy 100% The following Figure 5 and Figure 6 shows the result of XOR prediction

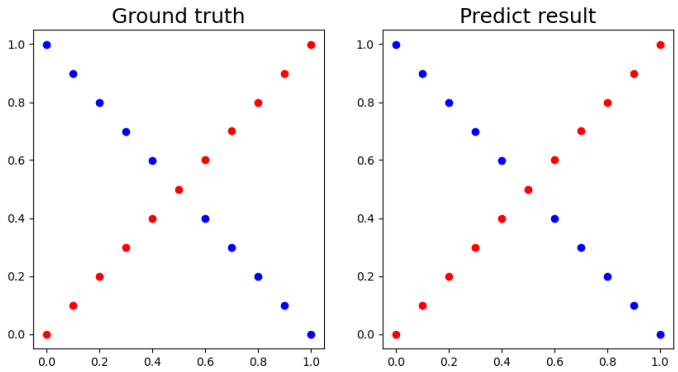

Figure 5: A. Screenshot and comparison figure of XOR dataset

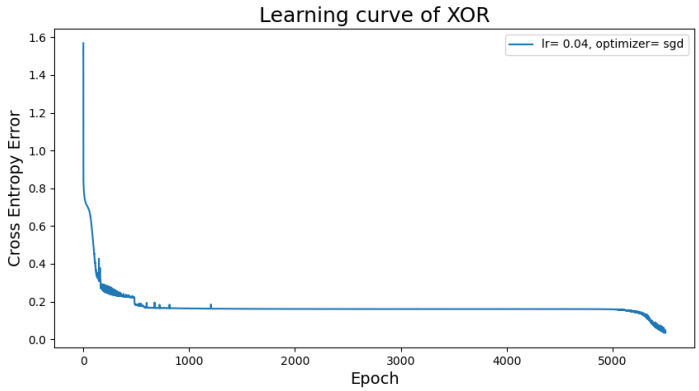

Figure 6: C. Learning curve of XOR dataset

D. Anything you want to present First, let's compare the learning curve of linear and XOR. Since we have more training data on linear, it only takes about 500 epochs to reach 100% accuracy compared to XOR under the same LR and optimizer. Second, since the optimizer is SGD, you can notice that the learning curve is oscillating in Figure 4 and Figure 6 at some epochs.

4. Discussion

A. Try different learning rates I've tried three different learning rates in this part. A larger learning rate means a larger distance for each gradient update, while a smaller learning rate means a smaller distance. When the learning rate is set too large, the distance between updates is too large, so the local minimum cannot be reached. Therefore, the loss cannot be reduced as shown in Figures 7 and 8.

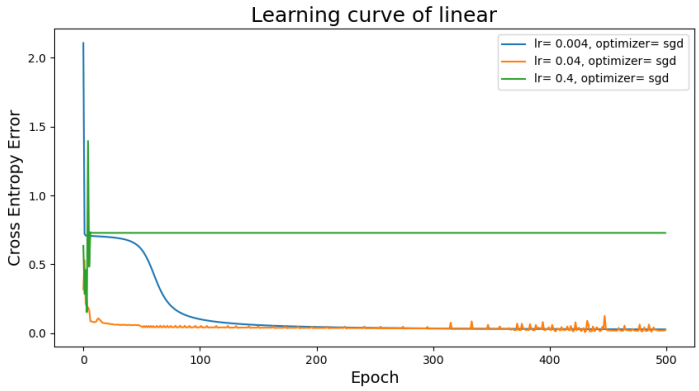

Figure 7: Different LR of linear dataset

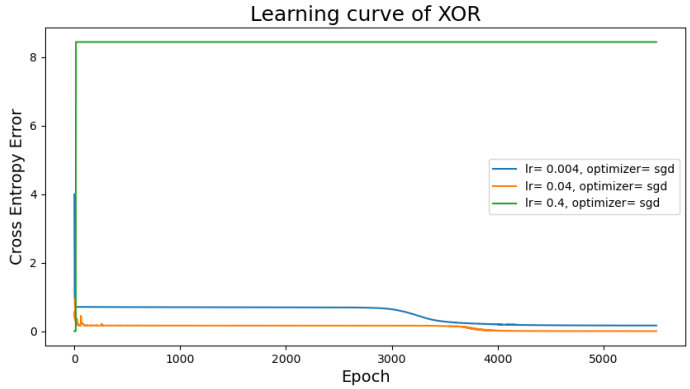

Figure 8: Different LR of XOR dataset

B. Try different numbers of hidden units I've tried three different units in this part. From Figures 9 and 10, it seems that the more neurons there are, the harder the model can be trained. In my opinion, as the depth (number of layers or units) of neural networks increases, gradients can start to vanish (become very small), making the optimization process difficult, and thus the loss will not converge.

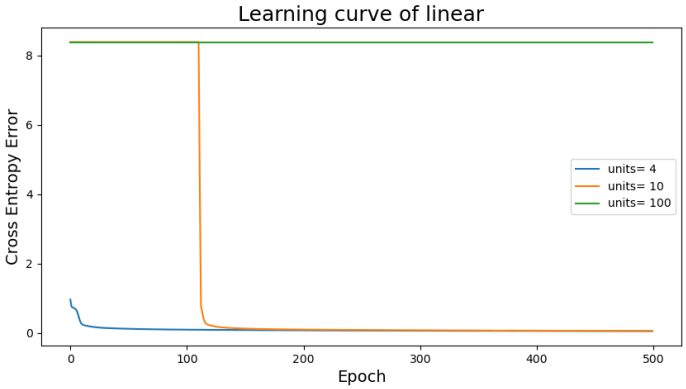

Figure 9: Different units of linear dataset

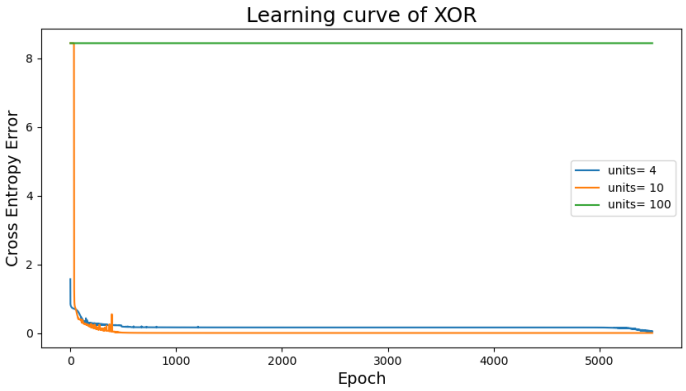

Figure 10: Different units of XOR dataset

C. Try without activation functions Without the activation function, the model can't nonlinearly differentiate the nonlinear data and the loss will be stuck in one place.

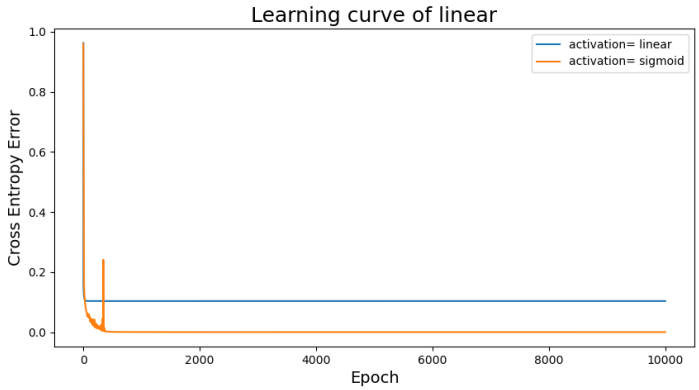

Figure 11: Without activation functions of linear dataset

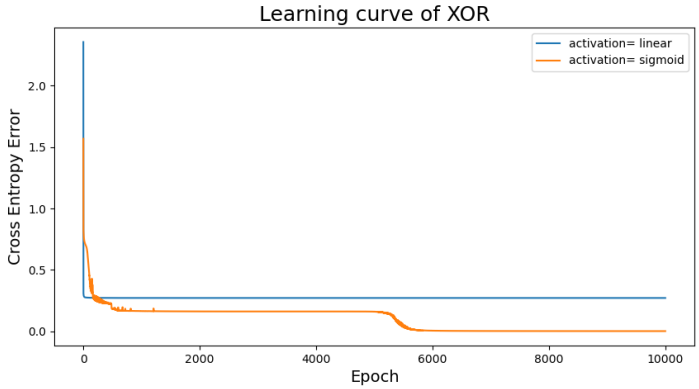

Figure 12: Without activation functions of XOR dataset

D. Anything you want to share I mentioned at the beginning that the classification methods could be logistic regression or classification. Therefore, in addition to the original logistic regression with the sigmoid function for the output layer, I also used the softmax function as an exception. Unlike the sigmoid function, softmax can help us to categorize more than two kinds of data. However, in this lab, since there is only one label, I have to do one-hot encoding by myself and notice that the output dimension of the model has to be changed to 2. From the results, the softmax function performs better on the XOR dataset.

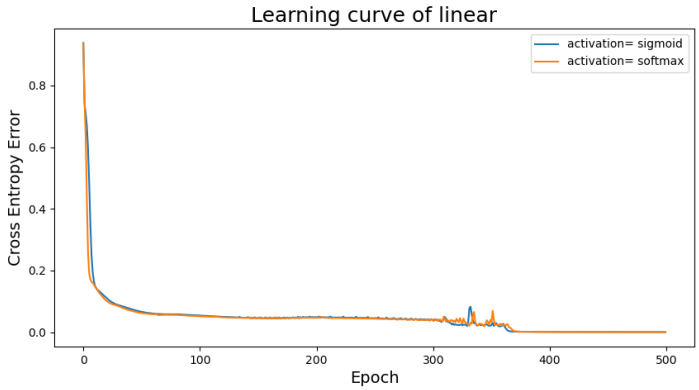

Figure 13: Classification of linear dataset

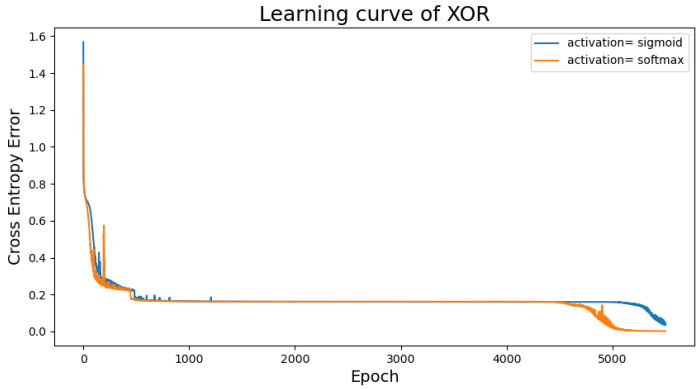

Figure 14: Classification of XOR dataset

5. Extra (10%)

A. Implement different optimizers (2%) To solve the oscillating problem, I implemented the optimizer, momentum. From the Figures below, we can see that the loss can converge faster.

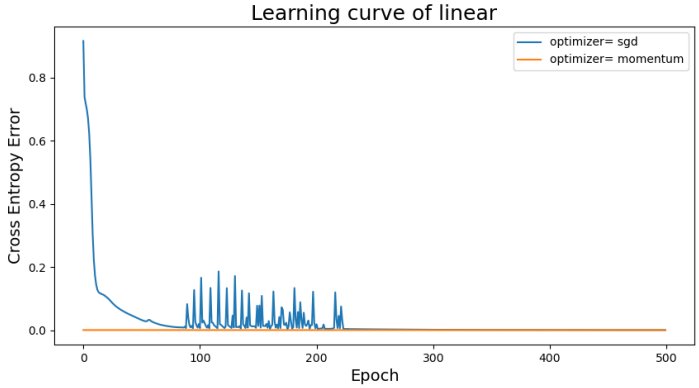

Figure 15: Different optimizers of linear dataset

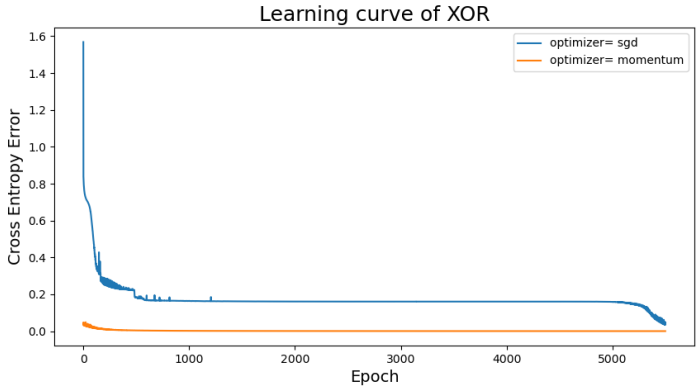

Figure 16: Different optimizers of XOR dataset

B. Implement different activation functions (3%) I have also implemented the activation function tanh, and we can see that tanh performs better in handling XOR data.

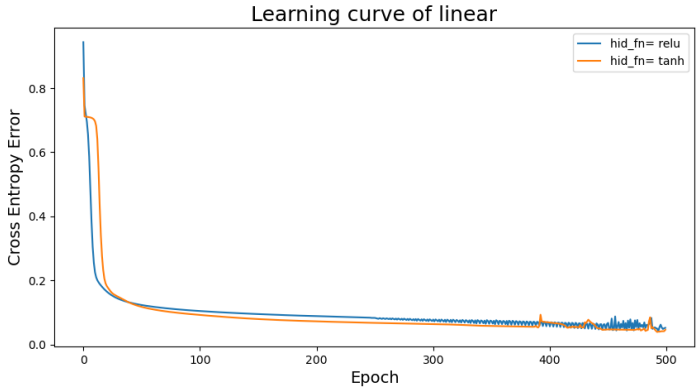

Figure 17: Different activation functions of linear dataset

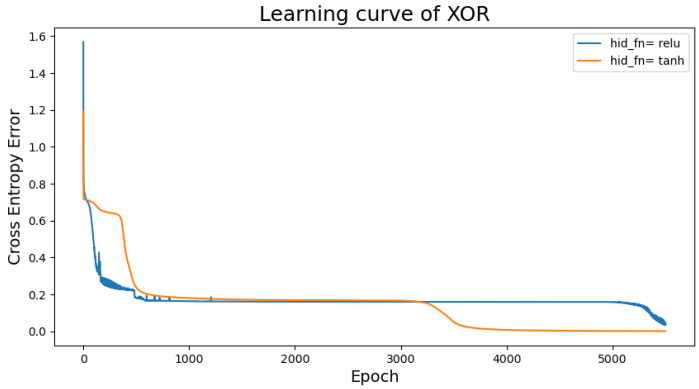

Figure 18: Different activation functions of XOR dataset

References

[1] Yun-Nung Vivian Chen. 台大資訊深度學習之應用 / *ADL 2.5: Backpropagation* 效率地計算大量參數. 2022. URL: https://youtu.be/oYnJ3pmRhjk.