Lab 1: Backpropagation

Hsiang-Chun Yang
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University
yanghc.cs09g@nctu.edu.tw

1 Introduction

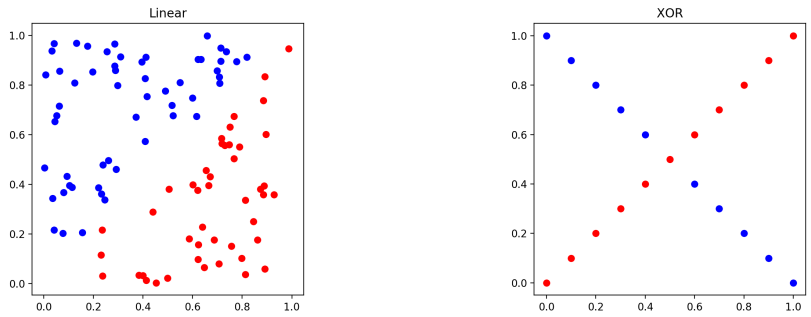
In this lab, we will use `Python`, `Numpy` and `Matplotlib` to

- implement a simple neural network with two hidden layers
- implement backpropagation
- visualize the prediction results and ground truth

1.1 Datasets

Two data generators are given by TA, please check Figure 1 for visualization.

- `generate_linear`
- `generate_XOR_easy`



(a) Linear data (b) XOR data

Figure 1: Datasets

2 Experiment setups

2.1 Sigmoid functions

I use sigmoid function as my activation function σ. The sigmoid function is defined as

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

For the derivative of sigmoid function, we have

$$\sigma'(x) = -\frac{-e^{-x}}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}} \times \frac{e^{-x}}{1+e^{-x}} = \frac{1}{1+e^{-x}} \times \frac{1+e^{-x}-1}{1+e^{-x}} = \sigma(x)(1-\sigma(x))$$

The implementations of sigmoid and its derivative are

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

def derivative_sigmoid(x):
    return sigmoid(x) * (1-sigmoid(x))
```

2.2 Neural network

Each layer is consisted with a weight matrix W and a bias matrix b. The shape of the weight matrix is $M \times N$ where M and N is the number of input features and output features respectively. And the bias matrix is a row vector with N elements.

For each input vector x, output $\hat{y}$ of a neural layer can be written as

$$z = xW + b$$
$$\hat{y} = \sigma(z)$$

By default, in my implementation, there are 4 neurons in each hidden layer. Please refer to `Network` object and `Layer` object in `main.py` for more details.

2.3 Loss function

I use mean square error (MSE) as my loss function $L(\theta)$. Given the outputs $\hat{y}$ of the neural network and the ground truth y, the loss can be written as

$$L(\theta) = MSE(\hat{y} - y) = \frac{1}{N}\Sigma_{i=1}^{N}(\hat{y_i} - y_i)^2 \text{ where } N \text{ is the batch size}$$

The partial derivative of MSE is

$$\frac{\partial MSE(\hat{y}-y)}{\partial y_i} = \frac{2}{N}(\hat{y_i} - y_i)$$

The implementations of MSE and its derivative are

```
def mse_loss(pred_y, y):
    return np.mean((pred_y-y)**2)

def derivative_mse_loss(pred_y, y):
    return 2 * (pred_y-y) / y.shape[0]
```

2.4 Backpropagation

To update the weight matrices of the network, we need to compute $\frac{\partial C}{\partial W}$ where C is the cost (loss) between $\hat{y}$ and y. Since it is hard to compute $\frac{\partial C}{\partial W}$ directly, we will use chain rule to solve it.

$$\frac{\partial C}{\partial W} = \frac{\partial z}{\partial W}\frac{\partial \hat{y}}{\partial z}\frac{\partial C}{\partial \hat{y}}$$

For the implementation of this section, please refer to `Network` object and `Layer` object in `main.py`.

2.4.1 Forward gradient

Forward gradient can be easily computed when doing forward propagation.

$$\frac{\partial z}{\partial W} = \frac{\partial (xW+b)}{\partial W} = x$$

2.4.2 Backward gradient

$$\text{Backward gradient} = \frac{\partial \hat{y}}{\partial z}\frac{\partial C}{\partial \hat{y}}$$

Knowing that $\hat{y} = \sigma(x)$. The first part of backward gradient can be computed by calculate the derivative of sigmoid function.

$$\frac{\partial \hat{y}}{\partial z} = \frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1 - \sigma(z))$$

For the second part, considering two cases: output layer and hidden layer. For the output layer, cost C is computed by loss function $L(\theta) = MSE(\hat{y} - y)$

$$\frac{\partial C}{\partial \hat{y}} = \frac{\partial MSE(\hat{y}-y)}{\partial \hat{y}} = \frac{2}{N}(\hat{y} - y) \text{ where } N \text{ is the batch size}$$

For the hidden layer, we can divide $\frac{\partial C}{\partial \hat{y}}$ into two parts.

$$\frac{\partial C}{\partial \hat{y}^l} = \frac{\partial C}{\partial z^{l+1}}(\frac{\partial z^{l+1}}{\partial \hat{y}^l})^T \text{ where } l \text{ is the layer number}$$

Since we are computing gradients backward (from deep to shallow), we can assume that $\frac{\partial C}{\partial z^{l+1}}$ is already known. And $\frac{\partial z^{l+1}}{\partial \hat{y}^l}$ can be computed by

$$\frac{\partial z^{l+1}}{\partial \hat{y}^l} = \frac{\partial \hat{y} W^{l+1}}{\partial \hat{y}} = W^{l+1}$$

To sum up, backward gradient will have these two forms

$$\frac{\partial C}{\partial z} = \begin{cases} \sigma'(z) \cdot \frac{2}{N}(\hat{y} - y), \text{ for output layer} \\ \sigma'(z) \cdot \frac{\partial C}{\partial z^{next\ layer}}(W^{next\ layer})^T, \text{ for hidden layer} \end{cases}$$

2.4.3 Weight update

Once we get forward gradient and backward gradient, the gradient of the weight $\frac{\partial W}{\partial W}$ can be computed by multiplying these two together. Then, subtract a ratio α of the gradient from the weight matrix to update it. The α here is so-called learning rate and is set to 1.0 by default.

$$W = W - \alpha \frac{\partial C}{\partial W}$$

3 Results

Given the default configurations

- 4 neurons in each hidden layer
- learning rate α is set to 1.0
- batch size is equal to the total number of input data
- training will be stop if the loss is lower than 0.005 or the number of epochs reaches 100000

The following results and figures are based on these configurations.

3.1 Results and comparison figures

In the comparison figure, the misclassified points will be marked by a black circle.

3.1.1 Linear data

```
Datset: linear data
------------------
| Start training |
------------------
[Epoch:   500] [Loss: 0.031287]
[Epoch:  1000] [Loss: 0.015211]
[Epoch:  1500] [Loss: 0.010655]
[Epoch:  2000] [Loss: 0.008394]
[Epoch:  2500] [Loss: 0.006968]
[Epoch:  3000] [Loss: 0.005952]
[Epoch:  3500] [Loss: 0.005174]
[Epoch:  3631] [Loss: 0.005000]
-----------------
| Start testing |
-----------------
Prediction:
[[5.17958352e-01]
 [2.79324129e-04]
 [9.97638420e-01]
 [1.95324498e-02]
 [9.67728077e-01]
 [9.98983934e-01]
 [9.95732407e-01]
 [9.99059770e-01]
 [2.15356691e-04]
 [9.98527043e-01]
 [9.69576096e-01]
 [9.99146215e-01]
 [8.34554991e-04]
 [9.99144300e-01]
 [9.99175566e-01]
 [9.59226388e-01]
 [9.98193254e-01]
 [9.99178024e-01]
 [3.23139023e-04]
 [1.13468769e-03]
 [6.41648050e-03]
 [4.15448663e-03]
 [9.99355342e-01]
 [2.90084265e-01]
 [2.91839958e-03]
 [9.99027665e-01]
 [2.13766921e-03]
 [2.94857569e-04]
 [9.75474396e-01]
 [6.31164240e-04]
 [9.98693800e-01]
 [8.81154202e-04]
```

```
[2.15952522e-02]
[3.15560213e-04]
[9.95522124e-01]
[2.15839047e-03]
[2.06620812e-04]
[5.21973250e-01]
[9.98808071e-01]
[9.99370579e-01]
[9.99413763e-01]
[2.29635450e-04]
[9.99374513e-01]
[2.14405586e-04]
[5.33915885e-02]
[2.11703023e-04]
[9.77791701e-01]
[6.82456562e-04]
[8.71173950e-04]
[9.99134726e-01]
[3.48836168e-01]
[9.99353486e-01]
[5.89267956e-01]
[6.90718053e-01]
[9.99096405e-01]
[5.19715738e-01]
[3.38785437e-04]
[6.79922308e-01]
[9.99437763e-01]
[5.99878668e-03]
[9.97305841e-01]
[2.49472052e-04]
[2.29560609e-02]
[2.53338964e-04]
[3.21087517e-04]
[9.99423677e-01]
[9.99204963e-01]
[9.98970516e-01]
[9.99186510e-01]
[9.97186886e-01]
[2.37541654e-04]
[9.99165724e-01]
[9.99390389e-01]
[9.82117695e-01]
[2.51614658e-04]
[2.36075721e-04]
[1.79318255e-03]
[9.99359282e-01]
[1.35351624e-01]
[2.75211164e-04]
[2.27968802e-03]
```

```
 [2.27832034e-04]
 [9.99117800e-01]
 [2.75609481e-01]
 [2.53619844e-04]
 [8.61328329e-03]
 [1.72593542e-02]
 [2.83803499e-04]
 [9.99235263e-01]
 [9.98754091e-01]
 [2.76709167e-01]
 [8.51165648e-01]
 [9.98472861e-01]
 [1.11759691e-03]
 [2.27382248e-02]
 [3.78874102e-01]
 [9.99195784e-01]
 [3.26450197e-04]
 [4.26592919e-04]
 [9.98316090e-01]]
Testing loss: 0.018
Acc: 98/100 (98.0%)
```

Acc: 98.0%

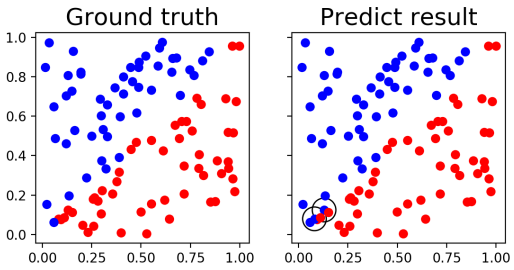


Figure 2: Result of linear data

3.1.2 XOR data

```
Dataset: XOR data
------------------
| Start training |
------------------
[Epoch:   500] [Loss: 0.236510]
[Epoch:  1000] [Loss: 0.141487]
[Epoch:  1500] [Loss: 0.038212]
[Epoch:  2000] [Loss: 0.015391]
```

```
[Epoch:  2500] [Loss: 0.006553]
[Epoch:  2692] [Loss: 0.004995]
-----------------
| Start testing |
-----------------
Prediction:
[[0.01118449]
 [0.99879237]
 [0.01047738]
 [0.99888116]
 [0.01775101]
 [0.99879791]
 [0.04753546]
 [0.99710191]
 [0.10993695]
 [0.84613179]
 [0.14050197]
 [0.1080741 ]
 [0.8289132 ]
 [0.06267566]
 [0.99758629]
 [0.03327374]
 [0.9995199 ]
 [0.01821128]
 [0.99971731]
 [0.01078522]
 [0.99977258]]
Testing loss: 0.005
Acc: 21/21 (100.0%)
```

Acc: 100.0%

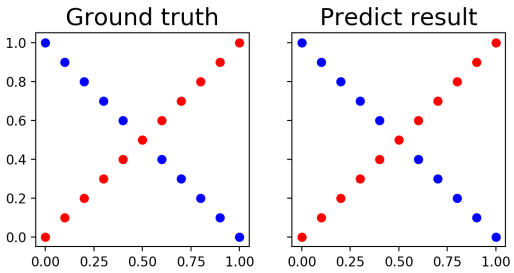


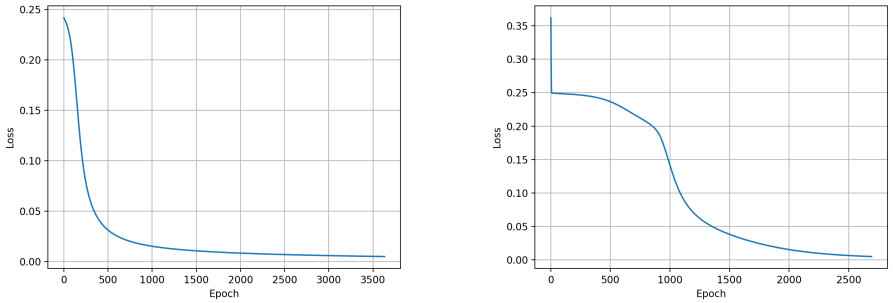
Figure 3: Result of XOR data

3.2 Accuracy

The accuracy of testing data is shown on the figures in section 3.1.

- Linear data: 98/100 (98.0%)
- XOR data: 21/21 (100.0%)

3.3 Learning curve



(a) Linear data (b) XOR data

Figure 4: Learning curves

4 Discussion

4.1 Different learning rates

Given the same network structure (section 3), I had tired 4 different learning rates.

4.1.1 Linear data

Accuracy of different learning rates:

- Learning rate = 1: 100.0%
- Learning rate = 0.5: 100.0%
- Learning rate = 0.1: 100.0%
- Learning rate = 0.05: 100.0%

Compare learning curve between different learning rates: Figure 5(a)

4.1.2 XOR data

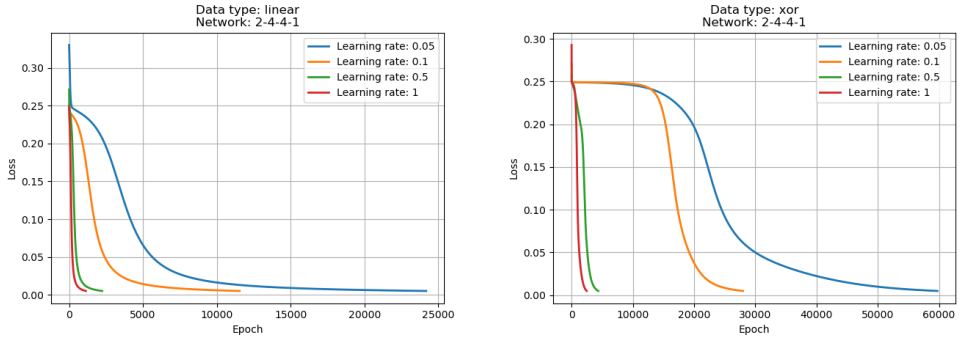
Accuracy of different learning rates:

- Learning rate = 1: 100.0%
- Learning rate = 0.5: 100.0%
- Learning rate = 0.1: 100.0%
- Learning rate = 0.05: 100.0%

Compare learning curve between different learning rates: Figure 5(b)

4.1.3 Observations

It is obvious that the network structure of 4 neurons in each hidden layer is enough to obtain a satisfiable performance (> 90%). The main difference between these 4 learning rates is that the loss decreases faster with a larger learning rate.



(a) Linear data (b) XOR data

Figure 5: Different learning rates

4.2 Different numbers of hidden units

In this section, I fixed the learning rate to 1 and tried 3 different number of hidden units.

4.2.1 Linear data

Accuracy of different numbers of hidden units:

- 2 neurons in each hidden layer: 100.0%
- 4 neurons in each hidden layer: 100.0%
- 8 neurons in each hidden layer: 100.0%

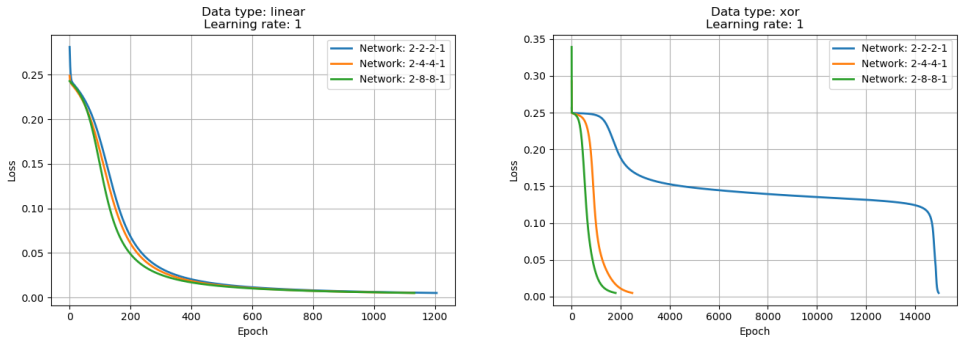
Compare learning curve between different numbers of neurons: Figure 6(a)

4.2.2 XOR data

Accuracy of different numbers of hidden units:

- 2 neurons in each hidden layer: 100.0%
- 4 neurons in each hidden layer: 100.0%
- 8 neurons in each hidden layer: 100.0%

Compare learning curve between different numbers of neurons: Figure 6(b)



(a) Linear data (b) XOR data

Figure 6: Different numbers of neurons

4.2.3 Observations

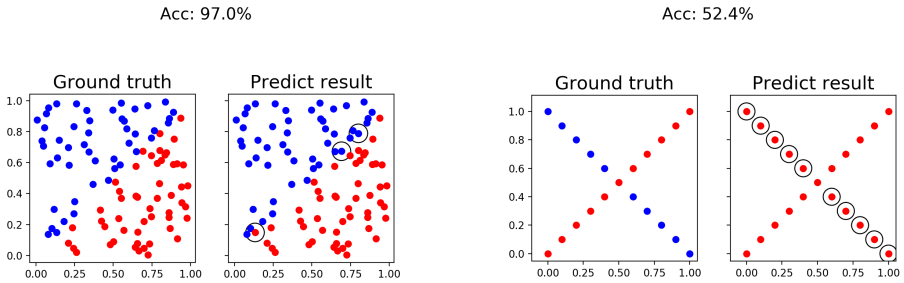
There is no great difference between 3 cases on linear data. Since we can separate two classes with a single straight line. The network with 2 neurons in each hidden layer is enough to solve this problem. On the other hand, XOR data does not have a linear solution. Network needs to learn some complex patterns to solve the classification problem. Although the network with 2 neurons in each hidden layer can solve the problem, we may fasten the training process by increasing the number of neurons.

4.3 Without activation functions

For the section, I use the following network configurations

- remove activation function in each layer
- 4 neurons in each hidden layer
- learning rate is set to 0.001
- training will be stop if the loss is lower than 0.005 or the number of epochs reaches 100000

Figure 7 is the experiment result.



(a) Linear data (b) XOR data

Figure 7: Without activation functions

One important feature about activation function is to provide non-linearity which allows the network to learn complex patterns from the data. Without activation function, the network can only solve the linear regression problem. In Figure 7(b), we can observe that about half of inputs are misclassified. In such case, the network can only produce a linear decision boundary since there are no activation functions inside it.

Another thing is about learning rate. Notice that the learning rate here is much smaller than those in section 3 and section 4.1. With activation function, the output value of each layer will be bounded in a certain range, which can prevent the gradients from overflowing. After removing activation functions from the network, the gradients of shallow layers will be very large. If the learning rate is too large, the weight matrices might overflow during weight update.