Deep Learning Lab1. Backpropagation Report

ID: 412551030   Name: 黃羿寧

1. Introduction

Deep learning has been widely used in several fields such as image recognition, medical diagnosis and natural language processing. To deeply understand the steps of neural network training, it's crucial to delve into the core components which make up a neural network: the architecture, the activation functions, the loss function, and to train the neural network using iteratively weight updates. In this report, the experimental setups including the provided train/test data, transformation functions, neural network applications, and backpropagation will be thoroughly examined. Then, the result of the experiment will be listed detailed to demonstrate the practice of Numpy-based neural network training. Furthermore, in the following section, I will describe the influences of hyperparameters such as learning rates, the numbers of hidden units and the activation functions on training outcomes.

2. Experimental setups

A. Training/Test datasets

In this Lab, the two sample datasets were provided by the DLP course, which are both used for binary classification (Figure 1A). In order to test the accuracy of the model prediction, the training and testing datasets remain identical in this practice.

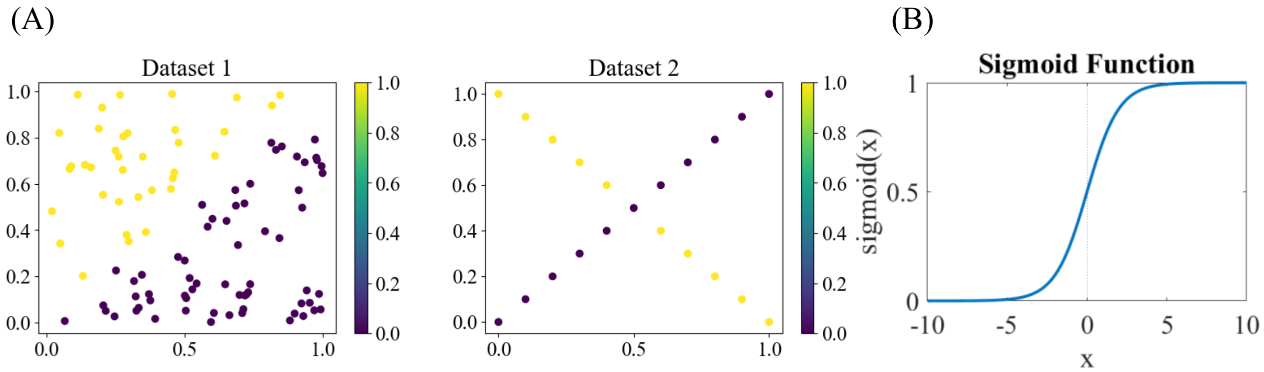

Figure 1. (A) Sample datasets applied in this practice. (B) An example of using sigmoid functions

B. Sigmoid functions

Additionally, to deal with the XOR problem, sigmoid functions were adopted as activation functions in several layers in this Lab. Sigmoid function was a non-linear transformation function, represented by the equation $\sigma(x) = \frac{1}{1+e^{-x}}$. It maps the input values to a value between 0 and 1. To illustrate the sigmoid function (Figure 1B), I generated x values ranging from -10 to 10 and then nonlinearly transformed them into a range between 0 and 1.

C. Neural network

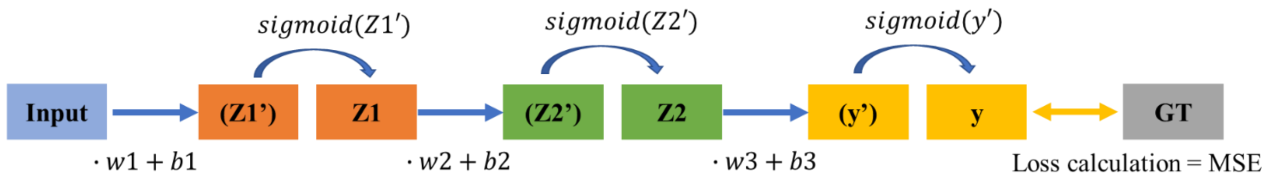

Figure 2. The architecture of the practice neural network.

Note that each square represents a bunch of nodes. The blue square derives the input layer orange and green nodes are the hidden layers, the yellow one is the output layer, and the grey one is the ground truth (GT).

The architecture of the neural network in Lab 1 consisted of one input layer, two hidden layers and one output layer (Figure 2). The size of the input layer was 2, the size of hidden layers was 3, and that of the output layer was set to 1 for requested binary classification. Each hidden layer first performs linear transformation, defined by $Y = XW + b$, where X is the input, W is the weight matrix and the b derives the bias vector. Following this step, a sigmoid activation function is adopted. Initially, the weight (w) for each layer were randomly generated, the bias (b) were set at zero, and they were subsequently updated using gradient of the loss function through standard gradient descent in the backpropagation step to minimize the loss after several iterations. For the loss function, the mean squared error (MSE) is utilized to deal with regression problems in this lab, defined as $MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2$ mathematically.

```
class Lab1_NN:
    def __init__(self, seed, input_size, L1_size, L2_size, Output_size):

        # Initial weights (rand variable)
        np.random.seed(seed)
        self.w1 = np.random.rand(input_size, L1_size)
        self.w2 = np.random.rand(L1_size, L2_size)
        self.w3 = np.random.rand(L2_size, Output_size)

        # Initial biases (rand variable)
        self.b1 = np.zeros((1, L1_size))
        self.b2 = np.zeros((1, L2_size))
        self.b3 = np.zeros((1, Output_size))

    def feedforward(self, Input):
        # First hidden layer
        self.z1 = sigmoid(np.dot(Input, self.w1)+self.b1)
        # Second hidden layer
        self.z2 = sigmoid(np.dot(self.z1, self.w2)+self.b2)
        # Output layer
        self.y = sigmoid(np.dot(self.z2, self.w3)+self.b3)
        return self.y
```

Figure 3. The code of initialization and feedforward processing. The weight (e.g., w1, w2, w3) are randomly generated through np.random.rand()

D. Backpropagation

The backpropagation fundamentally utilizes the chain rule of calculus to compute the gradients of the loss function with respect to weight/bias in the neural network. In this practice, I will introduce the simplified calculation adopted in the code step by step.

w3 & b3 case (output layer)

1. Using chain rule to calculate the derivative of loss with respect to w3

$$\frac{dL(\theta)}{d\,w3_{i,j}} = \sum_{j=1}^{n} \frac{\mathbf{dL(\theta)}}{\mathbf{d\,y_j}} \frac{\mathbf{dy_j}}{\mathbf{dy'_j}} \frac{\mathbf{dy'_j}}{\mathbf{dw3_i}},\ i = 1,\ 2, \dots m$$

In matrix notation (**Jocobian-gradient product**),

$$\nabla w3^{L(\theta)} = \left(\frac{d\,y\prime}{d\,w3}\right)^T \nabla y'^{L(\theta)}$$

$$= (\boldsymbol{Z2})^T\ (\boldsymbol{Loss\_derivative} \times \boldsymbol{sigmoid\_derivative(y)})$$

- Note that the $\boldsymbol{Loss\_derivative}$ is approximately equals to $-1 \times (GT - y)$
- The $\nabla b3^{L(\theta)}$has the similar calculation, except that it replaces the $(\boldsymbol{Z2})^T$ with 1

2. The weight update follows the equation of:

$$w3_1 = w3_0 - \nabla w3_0^{L(\theta)} \times \lambda$$

3. The bias update follows the equation of:

$$b3_1 = b3_0 - \nabla b3_0^{L(\theta)} \times \lambda$$

```
def backprop(self, Input, GT, learning_rate):
    # Propagation and Wights Update
    # W3 & b3
    loss_derivative = (-1)*(GT-self.y)
    Delta_3 = loss_derivative * sigmoid_derivative(self.y)
    self.w3 -= self.z2.T.dot(Delta_3)*learning_rate
    self.b3 -= np.sum(Delta_3, axis=0, keepdims=True)*learning_rate
```

w2 & b2 case (hidden layer)

1. Using chain rule to calculate the derivative of loss with respect to w2

$$\frac{dL(\theta)}{d\,w2_{i,j}} = \sum_{j=1}^{n} \frac{\mathbf{dL(\theta)}}{\mathbf{d\,y_j}} \frac{\mathbf{dy_j}}{\mathbf{dy'_j}} \frac{\mathbf{dy'_j}}{\mathbf{d\,Z2_j}} \frac{\mathbf{dZ2_j}}{\mathbf{dZ2'_j}} \frac{\mathbf{dZ2'_j}}{\mathbf{dw2_i}},\ i = 1,\ 2, \dots m$$

In matrix notation (**Jocobian-gradient product**),

$$\nabla w2^{L(\theta)} = \left(\frac{d\,Z2\prime}{d\,w2}\right)^T \nabla z2'^{L(\theta)}$$

$$= (\boldsymbol{Z1})^T\ (\boldsymbol{Loss\_derivative} \times \boldsymbol{sigmoid\_derivative(y)} \cdot \boldsymbol{w3}^T \times \boldsymbol{sigmoid\_derivative(Z2)})$$

$$= (\boldsymbol{Z1})^T\ (\boldsymbol{Delta\_3} \cdot \boldsymbol{w3}^T \times \boldsymbol{sigmoid\_derivative(Z2)})$$

- Note the $\boldsymbol{Loss\_derivative} \times \boldsymbol{sigmoid\_derivative(y)}$ is kept as $\boldsymbol{Delta\_3}$ variable in the code.

2. The update of weight and bias was identical to w3 and w3.

```
# W2 & b2
Delta_3_w3_T = Delta_3.dot(self.w3.T)
Delta_2 = Delta_3_w3_T*sigmoid_derivative(self.z2)
self.w2 -= self.z1.T.dot(Delta_2)*learning_rate
self.b2 -= np.sum(Delta_2, axis=0, keepdims=True)*learning_rate
```

w1 & b1 follows the similar process, since they are also a hidden layer

```
# W1 & b1
Delta_2_w2_T = Delta_2.dot(self.w2.T)
Delta_1 = Delta_2_w2_T*sigmoid_derivative(self.z1)
self.w1 -= Input.T.dot(Delta_1)*learning_rate
self.b1 -= np.sum(Delta_1, axis=0, keepdims=True)*learning_rate
```

3. Results

A. Screenshot and comparison figure

In dataset 1, I use learning rate of 0.001 and epoch iteration amount of 100000, and in dataset 2, I change the learning rate to 0.01 and change the epoch iteration amount to 200000. The screen shot of the training shows the epoch # and its loss. As we can see the loss of two dataset gradually decrease through the training process (Figure 4).

```
Dataset 1                                      Dataset 2
-------------------------------------          -------------------------------------
Epoch: 0, Loss: 0.287652                       Epoch: 0, Loss: 0.290255
Epoch: 5000, Loss: 0.248781                    Epoch: 10000, Loss: 0.249381
Epoch: 10000, Loss: 0.247613                   Epoch: 20000, Loss: 0.249301
Epoch: 15000, Loss: 0.237742                   Epoch: 30000, Loss: 0.248946
Epoch: 20000, Loss: 0.092736                   Epoch: 40000, Loss: 0.244773
Epoch: 25000, Loss: 0.033301                   Epoch: 50000, Loss: 0.165840
Epoch: 30000, Loss: 0.022445                   Epoch: 60000, Loss: 0.064409
Epoch: 35000, Loss: 0.017806                   Epoch: 70000, Loss: 0.046330
Epoch: 40000, Loss: 0.015083                   Epoch: 80000, Loss: 0.044203
Epoch: 45000, Loss: 0.013233                   Epoch: 90000, Loss: 0.043737
Epoch: 50000, Loss: 0.011870                   Epoch: 100000, Loss: 0.043567
Epoch: 55000, Loss: 0.010813                   Epoch: 110000, Loss: 0.043484
Epoch: 60000, Loss: 0.009965                   Epoch: 120000, Loss: 0.043437
Epoch: 65000, Loss: 0.009266                   Epoch: 130000, Loss: 0.043407
Epoch: 70000, Loss: 0.008680                   Epoch: 140000, Loss: 0.043387
Epoch: 75000, Loss: 0.008180                   Epoch: 150000, Loss: 0.043372
Epoch: 80000, Loss: 0.007747                   Epoch: 160000, Loss: 0.043361
Epoch: 85000, Loss: 0.007369                   Epoch: 170000, Loss: 0.043352
Epoch: 90000, Loss: 0.007034                   Epoch: 180000, Loss: 0.043345
Epoch: 95000, Loss: 0.006735                   Epoch: 190000, Loss: 0.043340
```

Figure 4. The screenshot of the training output of two datasets

The comparative figure of ground truth and the predictive value is shown below. The model predicts well in two datasets after training (Figure 5 & Figure 6).

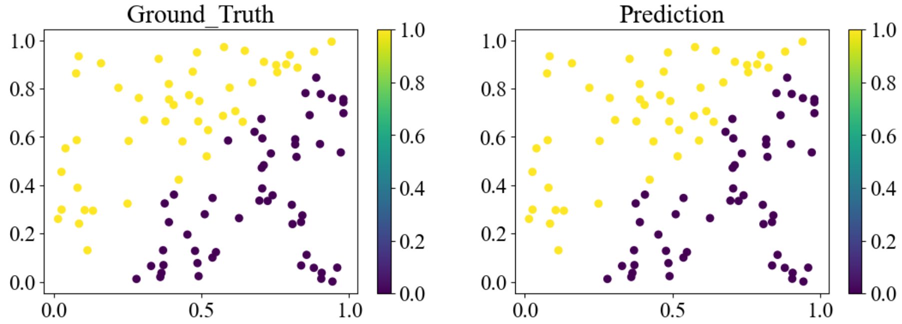

Figure 5. The comparative figure of dataset 1.

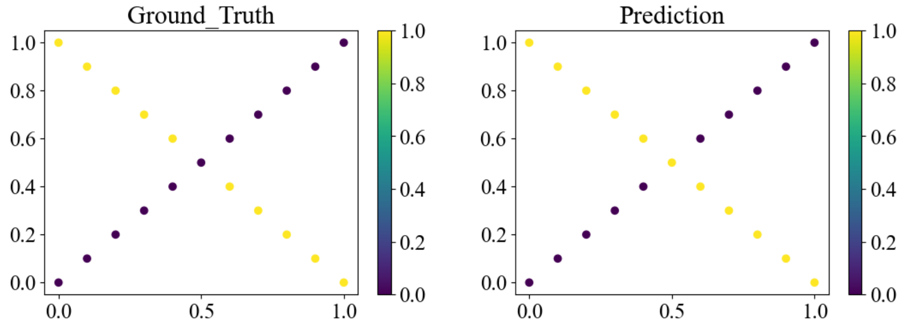

Figure 6. The comparative figure of dataset 2.

B. Show the accuracy of your prediction

The screenshot of the accuracy in model prediction is shown below (Figure 7), note that the full screenshot of the dataset 1 is in the *appendix*. The accuracy of the model prediction in dataset 1 is 100%, while that in dataset 2 is 95.2%, which remains a miss in the trial of (0.5, 0.5).

Dataset 1 (partial)

```
Num: 82 | GT: 1 | pred: 0.999811
Num: 83 | GT: 1 | pred: 0.999823
Num: 84 | GT: 0 | pred: 0.000685
Num: 85 | GT: 1 | pred: 0.999752
Num: 86 | GT: 0 | pred: 0.144930
Num: 87 | GT: 1 | pred: 0.992058
Num: 88 | GT: 1 | pred: 0.999614
Num: 89 | GT: 1 | pred: 0.999831
Num: 90 | GT: 0 | pred: 0.000408
Num: 91 | GT: 1 | pred: 0.965054
Num: 92 | GT: 0 | pred: 0.274463
Num: 93 | GT: 0 | pred: 0.000441
Num: 94 | GT: 0 | pred: 0.000531
Num: 95 | GT: 0 | pred: 0.000467
Num: 96 | GT: 0 | pred: 0.000421
Num: 97 | GT: 1 | pred: 0.890518
Num: 98 | GT: 0 | pred: 0.000456
Num: 99 | GT: 0 | pred: 0.000853
Loss: 0.003256 | Accuracy: 1.0
```

Dataset 2 (full)

```
Num: 00 | GT: 0 | pred: 0.000130
Num: 01 | GT: 1 | pred: 0.908201
Num: 02 | GT: 0 | pred: 0.000110
Num: 03 | GT: 1 | pred: 0.908201
Num: 04 | GT: 0 | pred: 0.000093
Num: 05 | GT: 1 | pred: 0.908201
Num: 06 | GT: 0 | pred: 0.000139
Num: 07 | GT: 1 | pred: 0.908201
Num: 08 | GT: 0 | pred: 0.020493
Num: 09 | GT: 1 | pred: 0.908201
Num: 10 | GT: 0 | pred: 0.908201
Num: 11 | GT: 0 | pred: 0.022758
Num: 12 | GT: 1 | pred: 0.908201
Num: 13 | GT: 0 | pred: 0.000111
Num: 14 | GT: 1 | pred: 0.908201
Num: 15 | GT: 0 | pred: 0.000033
Num: 16 | GT: 1 | pred: 0.908201
Num: 17 | GT: 0 | pred: 0.000024
Num: 18 | GT: 1 | pred: 0.908201
Num: 19 | GT: 0 | pred: 0.000022
Num: 20 | GT: 1 | pred: 0.908201
Loss: 0.043335 | Accuracy: 0.95238
```

Figure 7. The screenshot of the testing

C. Learning curve (loss, epoch curve)

The learning curve of two models is shown below. Interestingly, in the 29622 iterations of the dataset 1 reaches the 100% accuracy with the loss of 0.032 (Figure 8). The ceiling results in accuracy shows that the training after that might not be necessary in this practice. Similarly, the model training in dataset 2 also reached the ceil of 95.2% accuracy in the 58171th iteration (Figure 9). Hence, I will consider using early stopping to reduce the computational costs in the future.

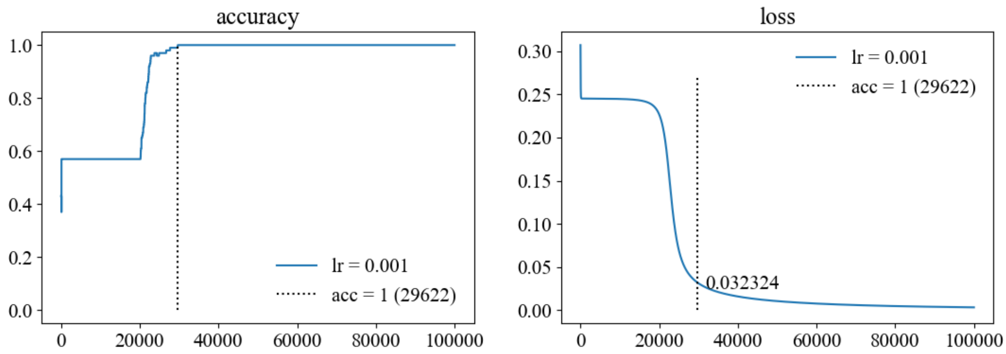

Figure 8. The learning curve of dataset 1 (accuracy & loss)

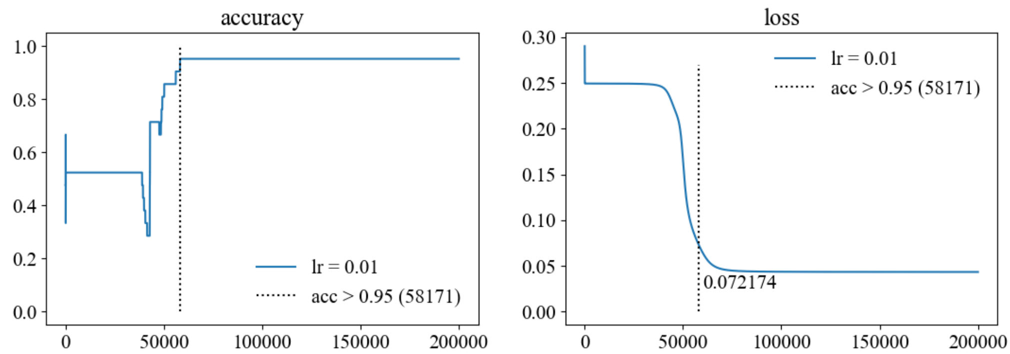

Figure 9. The learning curve of dataset 2 (accuracy & loss)

4. Discussion

A. Try different learning rates

The observation that different learning rates (0.001, 0.005, and 0.01) significantly influence the speed of learning within dataset 1 (Figure 10) and the training effectiveness within dataset 2 (Figure 11). In dataset 1, note that a learning rate of 0.01 achieves the fastest convergence to the performance ceiling. However, in dataset 2, using 200000 iterations to update the model seems not enough for a relative smaller learning rates (i.e., 0.001) to learn.

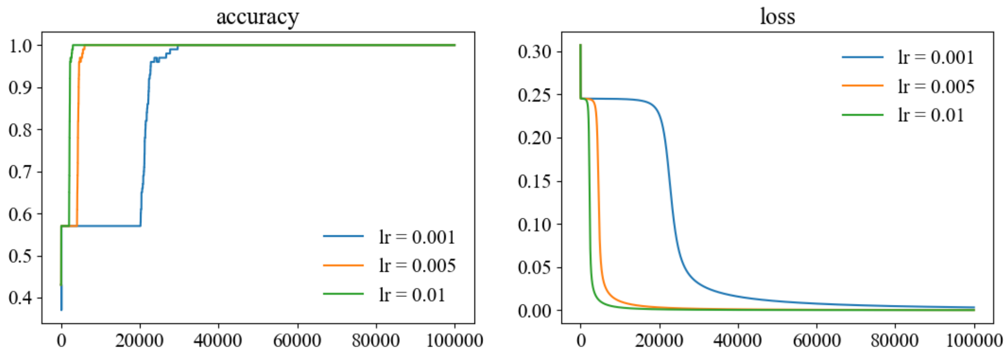

Figure 10. The influences of learning rates on the learning process in dataset 1

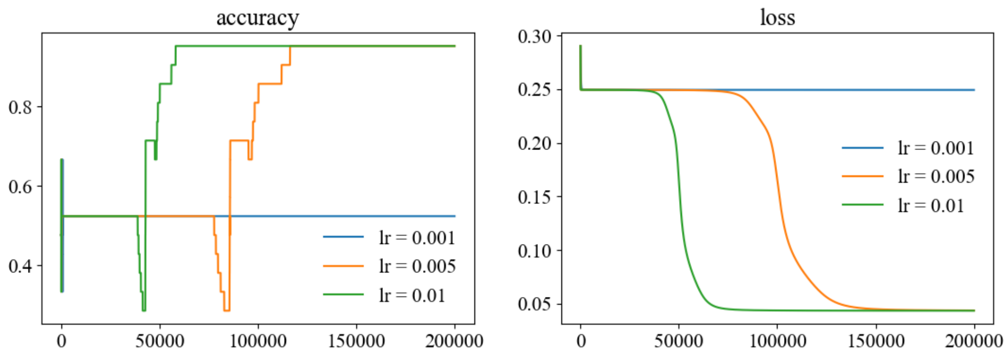

Figure 11. The influences of learning rates on the learning process in dataset 2

B. Try different numbers of hidden units

The number of hidden units was also examined in this practice. With the same hyperparameters applied in the main test, I tried on 4 kinds of combinations (I-H-H-O), which is 2-3-3-1 (original one), 2-2-2-1, 2-4-2-1, and 2-1-2-1. In dataset 1, the result shows that the combination did not influence the accuracy of the prediction, but it influenced the speed of training (Figure 12). The combination of 2-4-2-1 achieved fastest convergence to the performance ceiling. In contrast, the combination of 2-1-2-1 did not available to learn the pattern in dataset 2, and the 2-3-3-1 is the best combination (Figure 13).

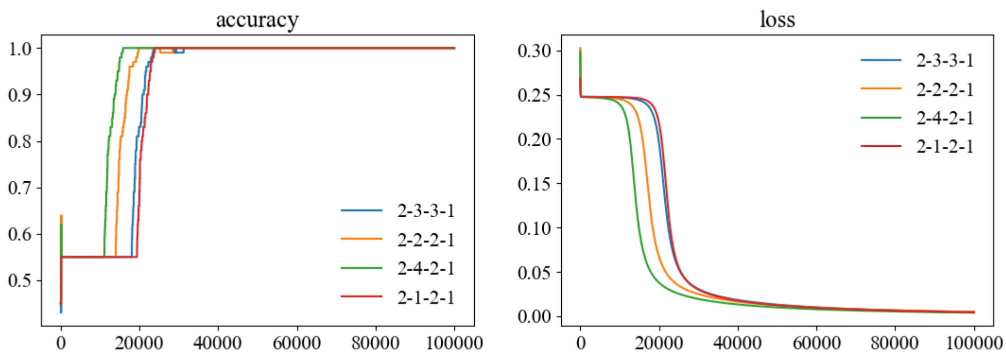

Figure 12. The learning curve of model training using different number of hidden units in dataset 1

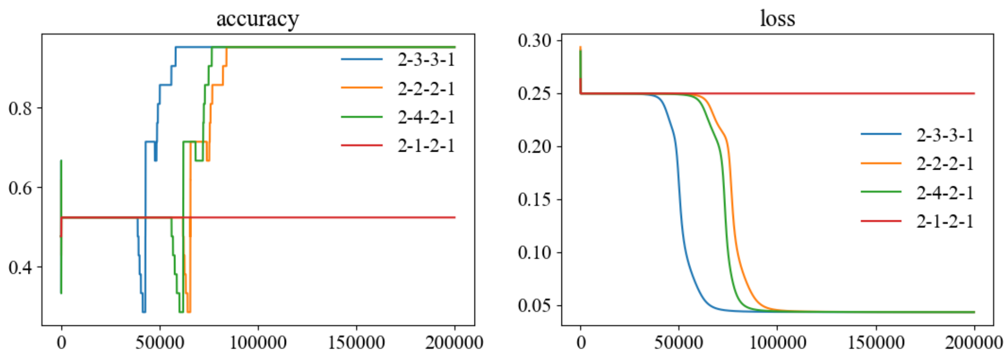

Figure 13. The learning curve of model training using different number of hidden units in dataset 2

C. Try without activation functions

The application of sigmoid function non-linearly transformed the value, so XOR problems are able to classify through model training. Dataset 1, a simple linear regression problem, is not influenced by the withdrawal of sigmoid functions (Figure 14). Instead, the learning process was extremely fast. However, in dataset 2, the XOR problem can not be learned by the model (Figure 15), which in lines with the current understanding.

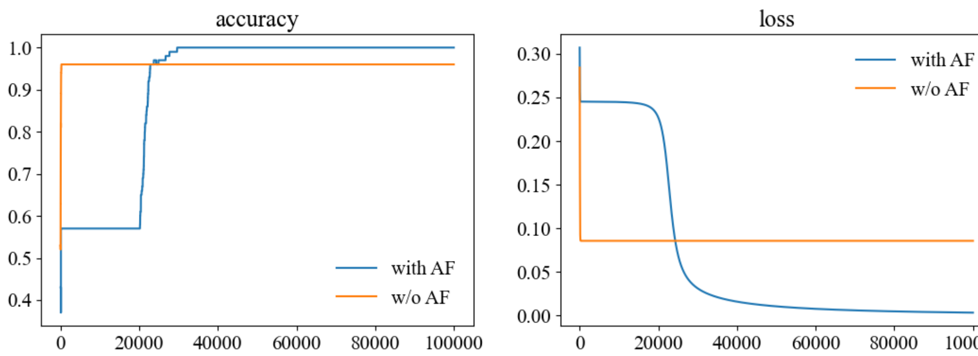

Figure 14. The learning curve of model training with and without activation functions in dataset 1

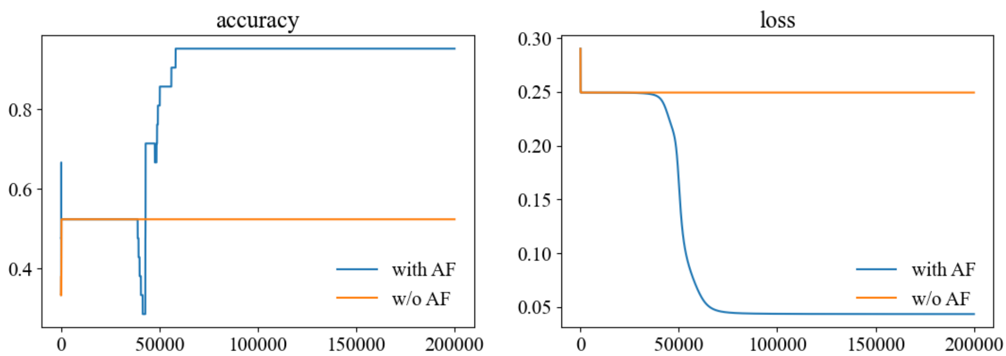

Figure 15. The learning curve of model training with and without activation functions in dataset 2

5. Extra

A. ReLU activation functions

As a comparison between saturated methods and one-sided saturation methods, I replaced the sigmoid function with the ReLU function in the practice, basically by replacing the sigmoid with RELU, and sigmoid_derivative with RELU_derivative.

```python
import numpy as np
# Activation function
def sigmoid(x):
    return 1/(1+np.exp(-x))

def RELU(x):
    return(np.maximum(0, x))

# Activation derivative for backprop
def sigmoid_derivative(x):
    return x*(1-x)

def RELU_derivative(x):
    return np.where(x>0, 1, 0)
```

The result shows the ReLU function speeds up the computation compared to the sigmoid function; however, the model applying ReLU did not achieve the 100% accuracy on the two datasets and stop updating remarkably soon (Figure 16 & Figure 17). The underlying cause for this performance discrepancy remains unclear due to the limited knowledge at this moment, but exploring this issue will be an important task for future work

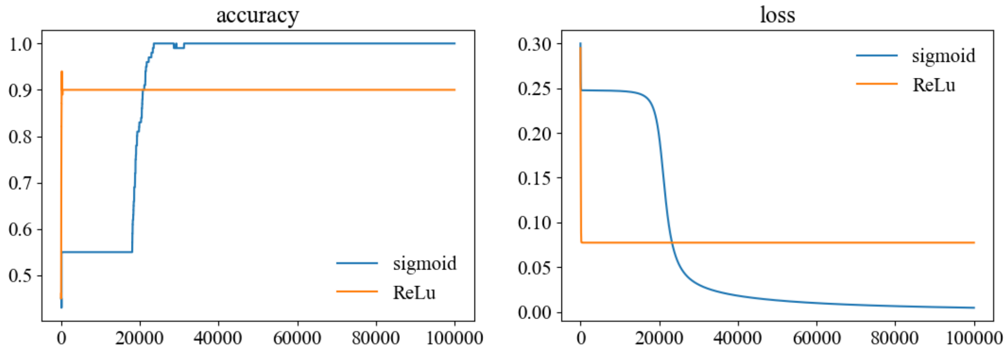

Figure 16. The learning curve of two models using sigmoid functions and ReLU functions in dataset 1

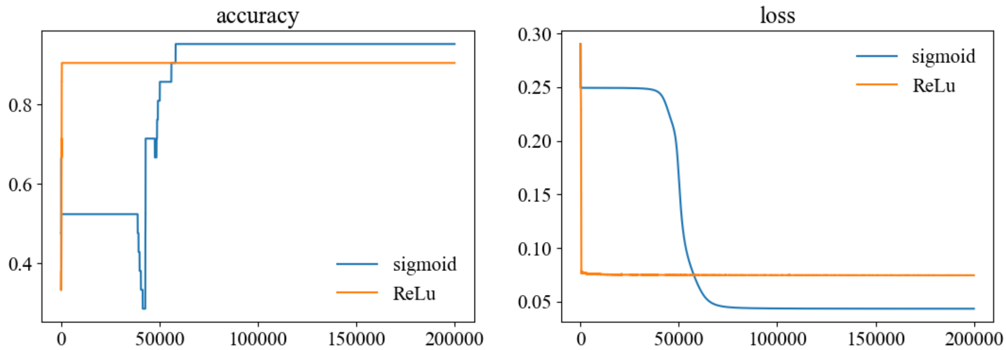

Figure 17. The learning curve of two models using sigmoid functions and ReLU functions in dataset 2

Appendix

```
Num: 00 | GT: 1 | pred: 0.999764
Num: 01 | GT: 1 | pred: 0.999729
Num: 02 | GT: 1 | pred: 0.999834
Num: 03 | GT: 1 | pred: 0.999857
Num: 04 | GT: 0 | pred: 0.000832
Num: 05 | GT: 0 | pred: 0.031676
Num: 06 | GT: 0 | pred: 0.002538
Num: 07 | GT: 0 | pred: 0.000670
Num: 08 | GT: 1 | pred: 0.999706
Num: 09 | GT: 0 | pred: 0.000526
Num: 10 | GT: 0 | pred: 0.000455
Num: 11 | GT: 1 | pred: 0.999809
Num: 12 | GT: 0 | pred: 0.026200
Num: 13 | GT: 0 | pred: 0.002885
Num: 14 | GT: 0 | pred: 0.000786
Num: 15 | GT: 0 | pred: 0.093021
Num: 16 | GT: 0 | pred: 0.000420
Num: 17 | GT: 1 | pred: 0.999794
Num: 18 | GT: 1 | pred: 0.996814
Num: 19 | GT: 0 | pred: 0.005690
Num: 20 | GT: 1 | pred: 0.999873
Num: 21 | GT: 1 | pred: 0.993654
Num: 22 | GT: 1 | pred: 0.999873
Num: 23 | GT: 0 | pred: 0.011251
Num: 24 | GT: 1 | pred: 0.999870
Num: 25 | GT: 1 | pred: 0.806268
Num: 26 | GT: 0 | pred: 0.002074
Num: 27 | GT: 0 | pred: 0.001059
Num: 28 | GT: 1 | pred: 0.999865
```

```
Num: 29 | GT: 0 | pred: 0.000622
Num: 30 | GT: 0 | pred: 0.003757
Num: 31 | GT: 0 | pred: 0.009842
Num: 32 | GT: 1 | pred: 0.736341
Num: 33 | GT: 0 | pred: 0.000658
Num: 34 | GT: 0 | pred: 0.005625
Num: 35 | GT: 0 | pred: 0.001239
Num: 36 | GT: 1 | pred: 0.999730
Num: 37 | GT: 0 | pred: 0.000489
Num: 38 | GT: 1 | pred: 0.843463
Num: 39 | GT: 0 | pred: 0.000555
Num: 40 | GT: 0 | pred: 0.000509
Num: 41 | GT: 0 | pred: 0.000414
Num: 42 | GT: 1 | pred: 0.999534
Num: 43 | GT: 1 | pred: 0.998492
Num: 44 | GT: 1 | pred: 0.999868
Num: 45 | GT: 1 | pred: 0.999871
Num: 46 | GT: 0 | pred: 0.235264
Num: 47 | GT: 0 | pred: 0.007328
Num: 48 | GT: 0 | pred: 0.002456
Num: 49 | GT: 1 | pred: 0.999832
Num: 50 | GT: 1 | pred: 0.999830
Num: 51 | GT: 0 | pred: 0.000401
Num: 52 | GT: 0 | pred: 0.000448
Num: 53 | GT: 0 | pred: 0.000485
Num: 54 | GT: 0 | pred: 0.000468
Num: 55 | GT: 1 | pred: 0.999875
Num: 56 | GT: 1 | pred: 0.997182
Num: 57 | GT: 0 | pred: 0.001041
```

```
Num: 58 | GT: 1 | pred: 0.999250
Num: 59 | GT: 0 | pred: 0.000737
Num: 60 | GT: 1 | pred: 0.999478
Num: 61 | GT: 1 | pred: 0.999866
Num: 62 | GT: 1 | pred: 0.999874
Num: 63 | GT: 0 | pred: 0.000955
Num: 64 | GT: 1 | pred: 0.997660
Num: 65 | GT: 0 | pred: 0.000438
Num: 66 | GT: 0 | pred: 0.021108
Num: 67 | GT: 0 | pred: 0.000950
Num: 68 | GT: 0 | pred: 0.000792
Num: 69 | GT: 0 | pred: 0.000921
Num: 70 | GT: 0 | pred: 0.085473
Num: 71 | GT: 0 | pred: 0.000485
Num: 72 | GT: 1 | pred: 0.999792
Num: 73 | GT: 0 | pred: 0.000556
Num: 74 | GT: 1 | pred: 0.999765
Num: 75 | GT: 0 | pred: 0.001163
Num: 76 | GT: 0 | pred: 0.001973
Num: 77 | GT: 0 | pred: 0.000754
Num: 78 | GT: 1 | pred: 0.940679
Num: 79 | GT: 1 | pred: 0.948178
Num: 80 | GT: 0 | pred: 0.000468
Num: 81 | GT: 1 | pred: 0.934442
Num: 82 | GT: 1 | pred: 0.999811
Num: 83 | GT: 1 | pred: 0.999823
Num: 84 | GT: 0 | pred: 0.000685
Num: 85 | GT: 1 | pred: 0.999752
Num: 86 | GT: 0 | pred: 0.144930
```

```
Num: 87 | GT: 1 | pred: 0.992058
Num: 88 | GT: 1 | pred: 0.999614
Num: 89 | GT: 1 | pred: 0.999831
Num: 90 | GT: 0 | pred: 0.000408
Num: 91 | GT: 1 | pred: 0.965054
Num: 92 | GT: 0 | pred: 0.274463
Num: 93 | GT: 0 | pred: 0.000441
Num: 94 | GT: 0 | pred: 0.000531
Num: 95 | GT: 0 | pred: 0.000467
Num: 96 | GT: 0 | pred: 0.000421
Num: 97 | GT: 1 | pred: 0.890518
Num: 98 | GT: 0 | pred: 0.000456
Num: 99 | GT: 0 | pred: 0.000853
Loss: 0.003256 | Accuracy: 1.0
```