Deep Learning Lab1 Report

學號: 313552052

姓名: 柳孝融

1. Introduction (5%)

This is a lab aimed at implementing backpropagation without using the existing python framework. In my work, there are three different versions of codes, the first one is a common MLP model with two hidden layers which can result in 96% accuracy, the second one I tried to use another optimizer, Adam, to compare the difference with the first one. Surprisingly, the accuracy can rise to 98% with the same model architecture. The last one is the convolution layer version, I used the convolution layer as the input layer, two hidden layers remained unchanged. Unfortunately, the results showed poor performance, only 38% accuracy, even with Adam as an optimizer.

2. Implementation Details (15%):

Sigmoid function:

Directly use the hint given in the specification which is computed by 1.0 / (1.0 + np.exp(-x)).

Neural network architecture:

There are four layers in my model architecture, an input layer with 2 neurons matched to the input data, then 2 hidden layers with 4 neurons respectively, and finally an output layer with one neuron matched to the output result.

Backpropagation

Output Layer:

output_delta = $(a3 - y) \odot a3 \odot (1 - a3)$ ← element-wise multiplication

$\partial L/\partial W3 = (1/m) \cdot a2^T \cdot$ output_delta

$\partial L/\partial b3 = (1/m) \cdot$ sum(output_delta, axis=0)

Second Hidden Layer:

hidden2_error = output_delta $\cdot W3^T$

hidden2_delta = hidden2_error $\odot a2 \odot (1 - a2)$

$\partial L/\partial W2 = (1/m) \cdot a1^T \cdot$ hidden2_delta

$\partial L/\partial b2 = (1/m) \cdot$ sum(hidden2_delta, axis=0)

First Hidden Layer:

hidden1_error = hidden2_delta $\cdot W2^T$

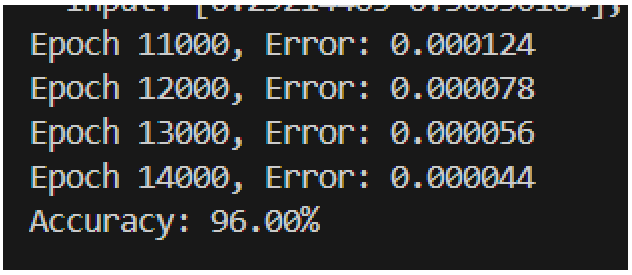
hidden1_delta = hidden1_error $\odot a1 \odot (1 - a1)$

$\partial L/\partial W1 = (1/m) \cdot X^T \cdot$ hidden1_delta

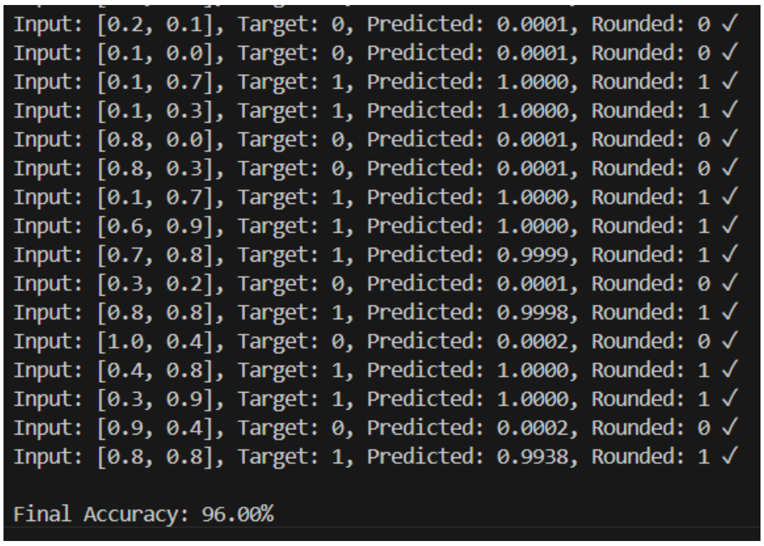
$\partial L/\partial b1 = (1/m) \cdot$ sum(hidden1_delta, axis=0)

3. Experimental Results (45%):

Screenshot and comparison figure



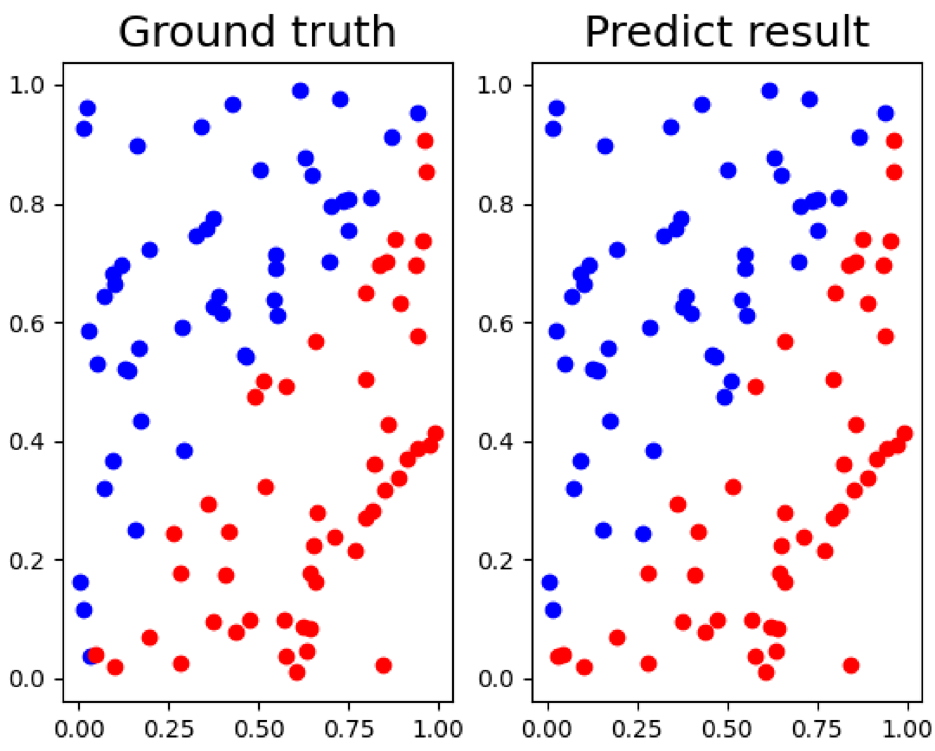
```
Epoch 11000, Error: 0.000124
Epoch 12000, Error: 0.000078
Epoch 13000, Error: 0.000056
Epoch 14000, Error: 0.000044
Accuracy: 96.00%
```



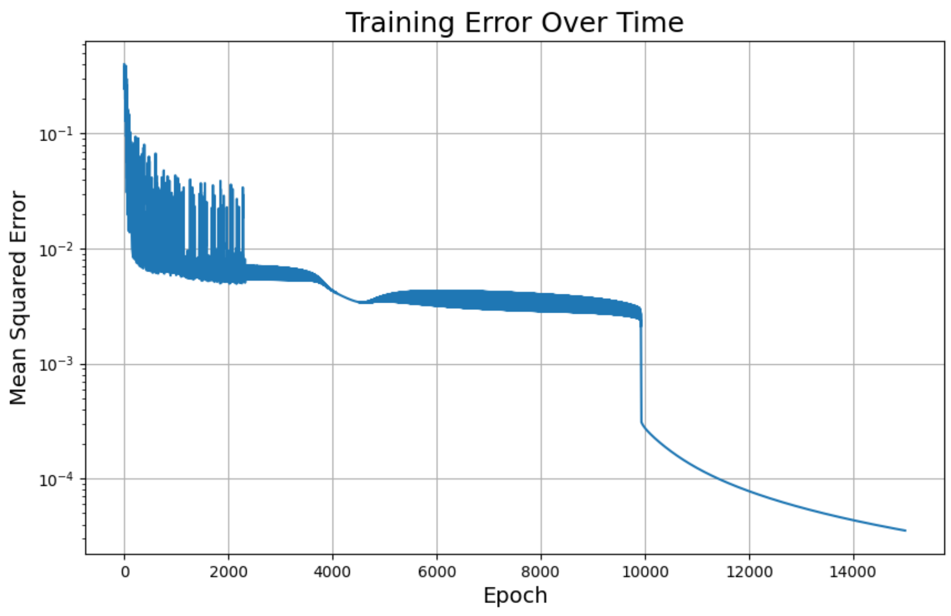
```
Input: [0.2, 0.1], Target: 0, Predicted: 0.0001, Rounded: 0 ✓
Input: [0.1, 0.0], Target: 0, Predicted: 0.0001, Rounded: 0 ✓
Input: [0.1, 0.7], Target: 1, Predicted: 1.0000, Rounded: 1 ✓
Input: [0.1, 0.3], Target: 1, Predicted: 1.0000, Rounded: 1 ✓
Input: [0.8, 0.0], Target: 0, Predicted: 0.0001, Rounded: 0 ✓
Input: [0.8, 0.3], Target: 0, Predicted: 0.0001, Rounded: 0 ✓
Input: [0.1, 0.7], Target: 1, Predicted: 1.0000, Rounded: 1 ✓
Input: [0.6, 0.9], Target: 1, Predicted: 1.0000, Rounded: 1 ✓
Input: [0.7, 0.8], Target: 1, Predicted: 0.9999, Rounded: 1 ✓
Input: [0.3, 0.2], Target: 0, Predicted: 0.0001, Rounded: 0 ✓
Input: [0.8, 0.8], Target: 1, Predicted: 0.9998, Rounded: 1 ✓
Input: [1.0, 0.4], Target: 0, Predicted: 0.0002, Rounded: 0 ✓
Input: [0.4, 0.8], Target: 1, Predicted: 1.0000, Rounded: 1 ✓
Input: [0.3, 0.9], Target: 1, Predicted: 1.0000, Rounded: 1 ✓
Input: [0.9, 0.4], Target: 0, Predicted: 0.0002, Rounded: 0 ✓
Input: [0.8, 0.8], Target: 1, Predicted: 0.9938, Rounded: 1 ✓

Final Accuracy: 96.00%
```

Show the accuracy of your prediction (40%):



Learning curve (loss-epoch curve)



4. Discussion (15%)

Try different learning rates

The learning rate was initialized to 0.3 and the result was 96% accuracy. If I shrank down the learning rate to 0.1, the accuracy was still 96%. Next, I shrank down further to 0.01, the accuracy became 99%! I figured out that the 0.3 learning rate was a little higher, causing the model unable to learn further.

Try different numbers of hidden units

The number of hidden units was 4 initially. I tried to increase the number of hidden units to 8, 16, 32 and 64. The accuracy didn’t have any change, but there were some additional costs in computing matrix multiplication, especially at the number 32 and 64.

Try without activation functions

I used sigmoid as my activation function initially. To find out what will happen if not using the activation function, I had some modifications. I only removed the implementation of the sigmoid function at first, but the result showed NaN at the loss value. I guessed this was caused by underflow in floating point representation. To overcome the problem, I changed the implementation of sigmoid function, simply applied normalization in the sigmoid and sigmoid_derivative to avoid the underflow problem. However, as I expected, the accuracy dropped down to 86%.

5.Questions (20%):

What is the purpose of activation functions? (6%)

To introduce non-linearity into neural networks. Without activation functions, a neural network would behave like a single linear transformation, regardless of how many layers it has. The non-linearity allows networks to learn complex patterns and relationships in data.

What might happen if the learning rate is too large or too small? (7%)

If the learning rate is too large:
Weight updates overshoot optimal values, loss function may oscillate or diverge, training becomes unstable or fails to converge, model might never learn meaningful patterns.
If the learning rate is too small:
Training progresses extremely slowly, model may get stuck in local minima, causing convergence takes an impractical amount of time, finally, gradients may vanish before meaningful learning occurs.

What is the purpose of weights and biases in a neural network? (7%)

Weights are used to scale the importance of each input feature and allow the network to identify which features are most relevant for the task. Besides, storing the learned patterns during training to transform input data into more useful representations at each layer.

Biases are primarily to act like a threshold value that must be overcome to activate a neuron, in addition, allowing neurons to be activated (or not) even when input is zero, or to shift where classification boundaries occur.

6.Extra:

Implement different optimizers. (2%)

I used Adam as my optimizer, which considered the moment estimate for each weight, helping accelerate training in consistent direction and adapting learning rate for each parameter individually. The result showed that the accuracy rose from 96% to 98% with learning rate 0.3.

Implement different activation functions. (3%)

The first one testing activation function was tanh() which was computed by np.tanh(x). The result drastically dropped down to only 49% accuracy (learning rate unchanged 0.3). After some tests, I realized that the problem lay in the high learning rate. If I adjusted the learning rate to 0.01, the accuracy became 99%. Similar results appeared in the ReLU function, the result dropped down to 51% accuracy at the beginning, after adjusting the learning rate, the accuracy became 96%. Strangely, this result didn't appear in sigmoid function.

Implement convolutional layers. (5%)

I implemented the convolution layer in my third version of codes, unfortunately, the result wasn't good, only 38% accuracy, even using Adam as the optimizer doesn't work well. I guessed it was because the dataset was more suitable for binary classification problems rather than convolution which was more suitable for image processing. In addition, convolution spent more computing cost than the common MLP model. As a result, the MLP was the best choice in this dataset.