NYCU DL

Lab1 – Backpropagation

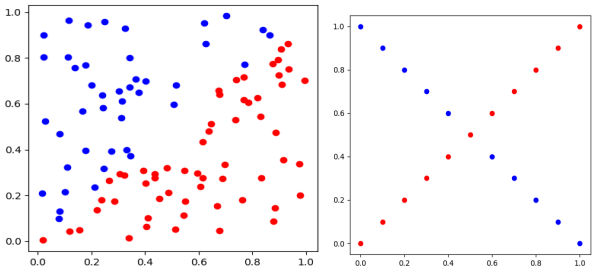
REPORT

[110704071] [Ray Peng(彭叡楀)]

1. Introduction (5%)

   In this lab, we try make a 2-layer neural network by using the basic library (ex. NumPy).

   In this lab we use 2 different kinds of data to train and test whether our model can work well (one in linear relation and the other one is not). We need to classify them into 2 different classes.



   The most important part in the lab will be the implementation of activation function, forward and backpropagation, as they are crucial for the neural network to learn from the data, adjust weights, and minimize the loss during training. Properly implementing these components ensures that the network can effectively update its parameters and make accurate predictions on new data.

2. Implementation Details (15%):

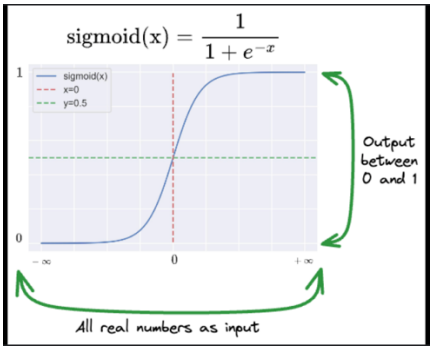
   A. Sigmoid function:

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

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

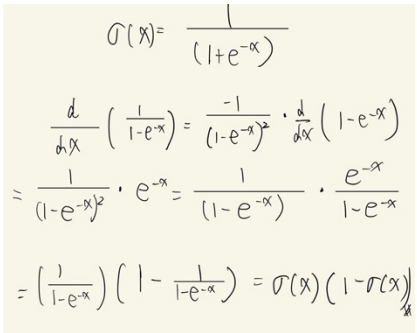
Since we need to classify our data into 2 classes, it’s suitable for us to use the Sigmoid function to implement the division.

Formula: $sigmoid(x) = 1/(1 + e_{-x})$

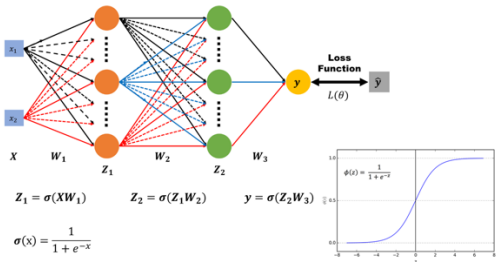
The sigmoid function maps the input to a range between [0, 1], making it useful for classification tasks where the output can be interpreted as the probability of the class being 1. This allows for a clear threshold-based classification. Specifically, if the sigmoid output is greater than 0.5, we classify the input as belonging to class 1, and if it's less than or equal to 0.5, we classify it as belonging to class 0.



Beside the implementation of the sigmoid function. I also implement the derivative of the sigmoid function:



B. Neural network architecture



In the figure above, we can see the structure of a 2-layer neural network. First, we multiply the input by the weights to get the weighted sum. This sum is then passed through an activation function (such as the Sigmoid function) to introduce non-linearity, transforming the data. This process is repeated in the next layer.

Finally, we compute the loss based on the network's output compared to the expected target, which allows us to update the weights during backpropagation.

At first, I initialize the weight and the bias of each layer. To construct the important components of the network.

```
def __init__(self, size_in, size1, size2, size_out, learning_rate=0.01):
    self.learning_rate = learning_rate

    self.weight1 = np.random.uniform(0, 1, (size_in, size1))
    self.bias1 = np.zeros((1, size1))

    self.weight2 = np.random.uniform(0, 1, (size1, size2))
    self.bias2 = np.zeros((1, size2))

    self.weight_out = np.random.uniform(0, 1, (size2, size_out))
    self.bias_out = np.zeros((1, size_out))
```

Then, I implement the forward method based on the above figure. I use NumPy matrix multiplication to compute the weighted sum and add the bias to ensure the output is not always zero. After that, we pass the sum through the sigmoid function to introduce non-linearity, obtaining the activation output Z. This activation is then passed to the next layer for further processing.

```
def forward(self, x):
    self.dot1 = np.dot(x, self.weight1) + self.bias1
    self.z1 = sigmoid(self.dot1)
    self.dot2 = np.dot(self.z1, self.weight2) + self.bias2
    self.z2 = sigmoid(self.dot2)
    self.dot_out = np.dot(self.z2, self.weight_out) + self.bias_out
    self.zout = sigmoid(self.dot_out)

    return self.zout
```

Below is the procedure of the training process. After the forward, I will use MSE to calculate the loss of the function than we will use it into our back propagation to adjust our model.

```python
def train(self,x,y,epoch=100):
    losses = []

    for ep in range(epoch):
        out = self.forward(x)
        loss = np.mean((y - out) ** 2)
        losses.append(loss)
        self.backward(x, y, out)

        if (ep + 1) % 500 == 0:
            print(f"epoch {ep + 1} loss: {loss:.4f}")

    plt.plot(range(1, epoch + 1), losses, label="Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.title("Training Loss Over Epochs")
    plt.legend()
    plt.grid()
    plt.savefig("/home/pc3429/Ray/DL_LAB1_110704071_彭叡楀/epoch-loss.png", dpi=300, bbox_inches='tight')

    return losses
```

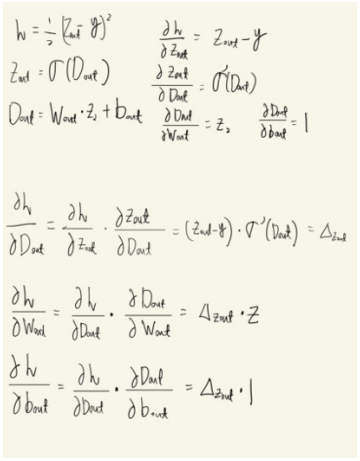
C. Backpropagation

```python
def backward(self, x, y, out):
    del_zout = (out - y) * sigmoid_derivative(self.dot_out)
    del_weight_out = np.dot(self.z2.T, del_zout)
    del_bias_out = np.sum(del_zout, axis=0, keepdims=True)

    del_z2 = np.dot(del_zout, self.weight_out.T) * sigmoid_derivative(self.dot2)
    del_weight2 = np.dot(self.z1.T, del_z2)
    del_bias2 = np.sum(del_z2, axis=0, keepdims=True)

    del_z1 = np.dot(del_z2, self.weight2.T) * sigmoid_derivative(self.dot1)
    del_weight1 = np.dot(x.T, del_z1)
    del_bias1 = np.sum(del_z1, axis=0, keepdims=True)

    self.weight1 -= self.learning_rate * del_weight1
    self.bias1 -= self.learning_rate * del_bias1

    self.weight2 -= self.learning_rate * del_weight2
    self.bias2 -= self.learning_rate * del_bias2

    self.weight_out -= self.learning_rate * del_weight_out
    self.bias_out -= self.learning_rate * del_bias_out
```

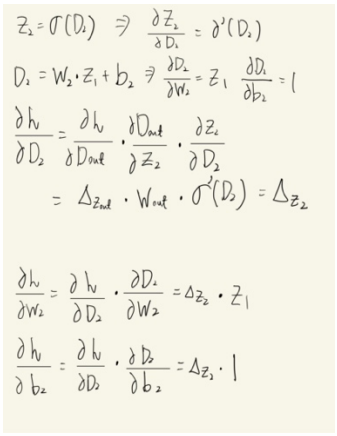
Backpropagation is a procedure that uses the chain rule to compute the gradient of the loss function with respect to the model's parameters. This allows us to update the model weights and improve its ability to classify the data correctly.

I use the calculus and chain rule to derive every derivative we need:

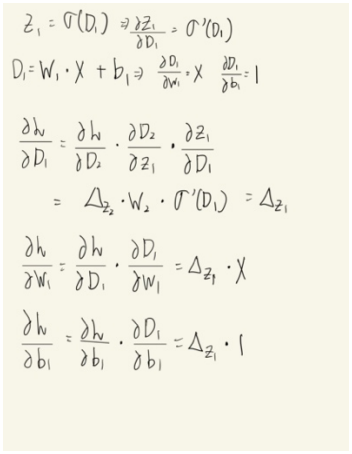
Output layer:



2nd layer:



1st layer:



I implement my code by the above derivation. And use the following method to ensure the correct form of the gradient update:

Matrix multiply: To ensure matrix dimensions align correctly, I transpose certain matrices or change the order to enable proper calculations for obtaining the gradient updates.

Bias term: Since the bias will affect all the part of data, I sum up all its components across the batch and use the mean to represent the overall gradient update.

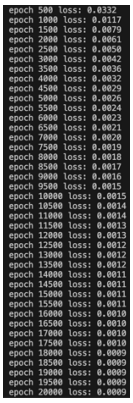
3. Experimental Results (45%)

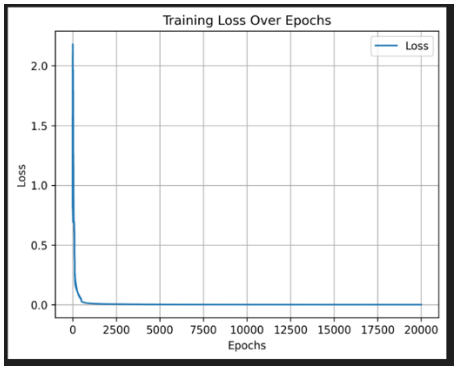
A. Screenshot and comparison figure

The numbers of the components:

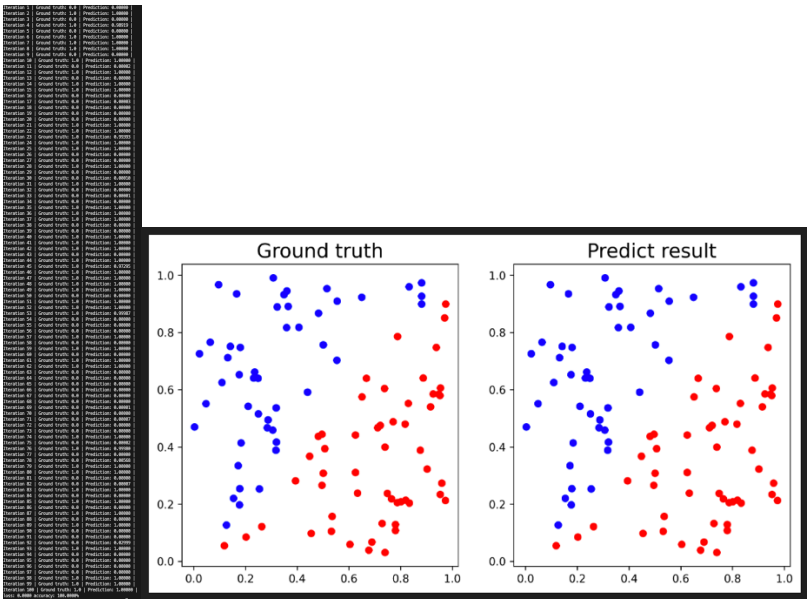
Learning_rate:0.15, size1: 8(hidden layer), size2: 8(hidden layer) epochs: 20000

1. generate_linear(n=100)

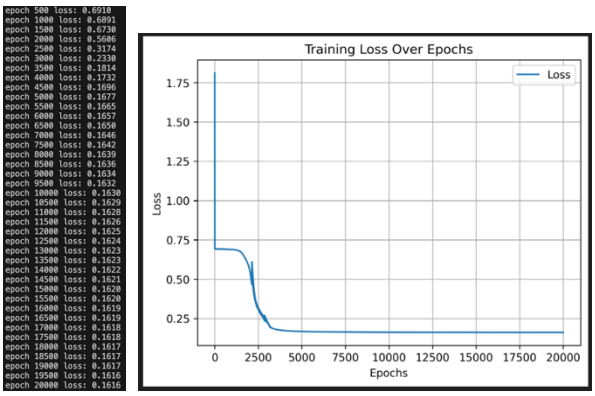




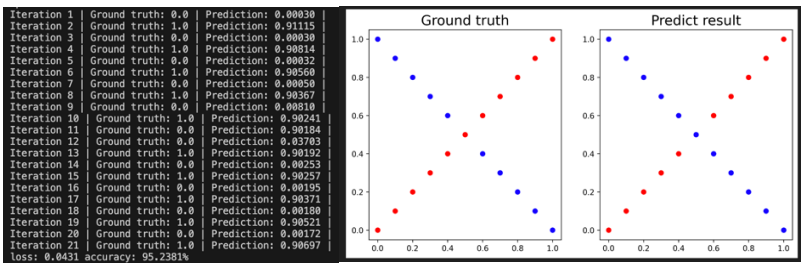
After the training, the loss decrease to nearly zero and hits the 100% accuracy.



2. generate_XOR_easy()



Since the data's relation is more complicate than the linear data, you can see the loss drop slower than the linear one, also the loss is larger than the normal one. So the accuracy is often slightly lower than the linear one (about 95%)



4. Discussion (15%)

In the below cases, I will use XOR data to implement the effects since it got more complicated relations between data.

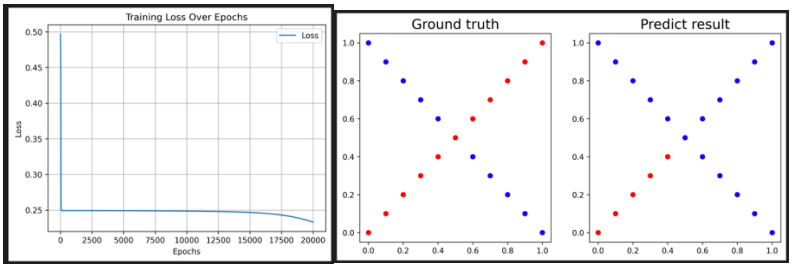
The original numbers of the components:

Learning_rate:0.15, size1: 8(hidden layer), size2: 8(hidden layer)

epochs: 20000

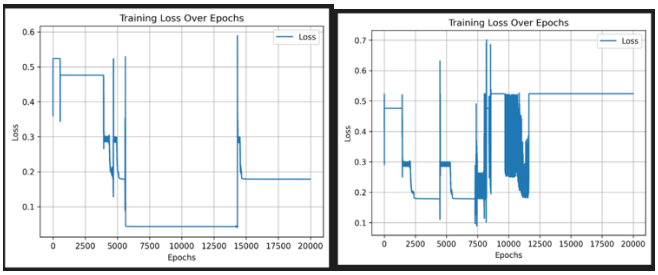
A. Try different learning rates

1) learning_rate = 0.015



We can see that if we use a learning rate that is too low, the loss decreases more slowly because the gradient updates are smaller. This may cause the training process to be slower than expected and result in lower accuracy within a limited number of epochs.

2) learning_rate = 2

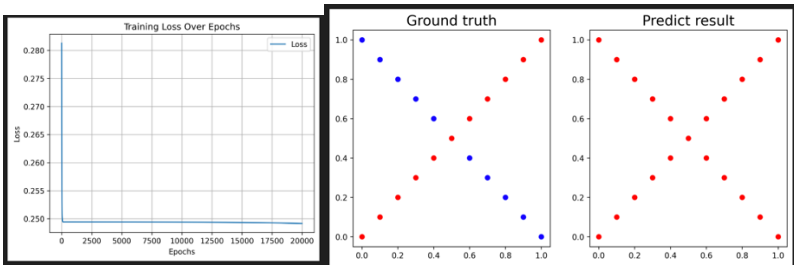


Although the model can sometimes make correct predictions during training, it often fails. You can observe that the training process is highly unstable and does not converge properly. Additionally, even when the model reaches a local minimum, it may not achieve the global minimum, preventing it from making the best possible predictions. This instability and suboptimal convergence could be caused by factors such as improper weight initialization, an inadequate learning rate, or the presence of vanishing gradients.

Based on these observations, finding an appropriate learning rate is crucial for effective training. A learning rate that is too high can cause the model to diverge, while a learning rate that is too low can lead to slow convergence or getting stuck in a suboptimal local minimum. Proper tuning of the learning rate helps stabilize training, improves convergence speed, and enhances the model's ability to generalize to unseen data.

B. Try different numbers of hidden units

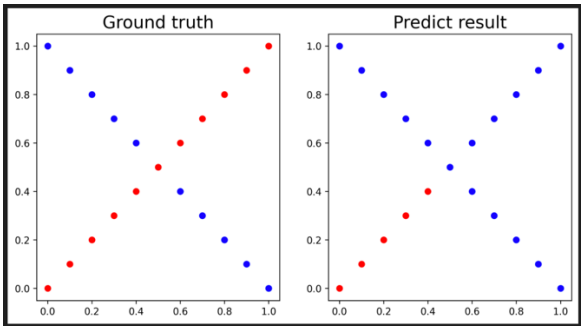
1)Size1 = 2, Size 2 = 2



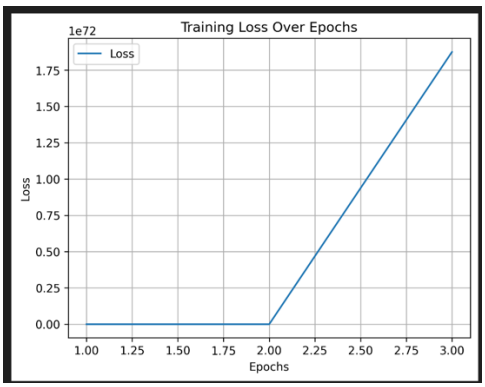
We can see that since the numbers of hidden units decrease, it is hard for the neural network to get the better prediction by the reduced representational capacity. We can see the gradient update slower, the overall model performance declines compared to the original architecture with more hidden units.

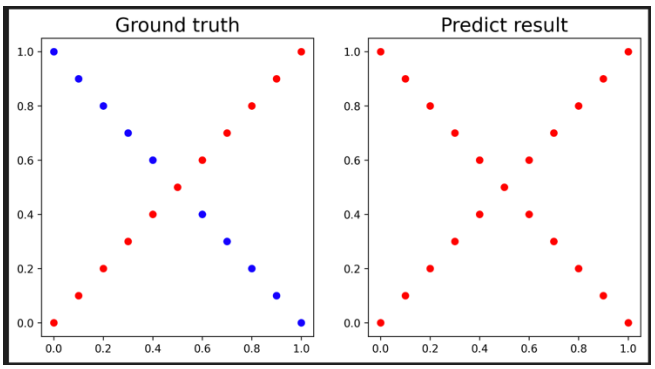
2)More hidden units: Getting more hidden units may cause overfitting, however, based on the lab we need to use the data on training and testing, so it isn't meaningful to observe the impact of overfitting.

C. Try without activation functions:



Here we use the XOR data to test whether remove the activation function will influence the outcome.





Without the sigmoid activation function, the network loses its ability to introduce non-linearity, making it difficult to learn complex patterns in the data. As a result, the model struggles to separate classes effectively, especially when dealing with non-linearly separable data. This leads to poor convergence of the loss function, and the model's classification performance deteriorates significantly.

5. Questions (20%)

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

Ans:

By using the activation function, we can let the model learn the data's complex relationships by introducing non-linearity. This allows the neural network to capture intricate patterns and dependencies instead of behaving like a simple linear model.

Additionally, we can choose the best-fit activation function based on the output characteristics. For example, if the output represents probabilities, using an activation function like

Sigmoid, which outputs values between 0 and 1, would be appropriate.

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

Ans:

A proper learning rate is essential for improving learning efficiency and ensuring a more stable process.

Learning rate is too low: Though the process may be stable, the step size during updates will be too small, leading to a slower convergence rate. This requires more iterations to complete the entire process, wasting time and reducing efficiency. In addition, when encountering a critical point (local minimum or saddle point), the result may get stuck in the vicinity, leading to a suboptimal solution.

Learning rate is too high: It can make the training process faster, but it may prevent the parameters from approaching the optimal solution, causing oscillations around it. A larger learning rate may also cause the loss function to become excessively large and fail to converge, making the entire training process highly unstable.

Therefore, finding the most suitable learning rate is crucial.

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

Ans:

Weight: The weights determine the importance of each input feature. During backpropagation, we adjust the weights gradually to learn the patterns between the data and the output. During testing, we use the trained model to make predictions based on the given inputs, where the

weighted inputs are summed up and passed through an activation function to generate the final prediction.

Bias: The additional variable in the neuron, called the bias, ensures that the output is not zero, even when all inputs are zero. This allows the neuron to adjust its baseline output independently of the input, preventing it from getting stuck in a narrow range. The bias increases the flexibility of the model by allowing it to learn more complex patterns and better adapt to different datasets.