311551180_資工碩一_楊佳誠

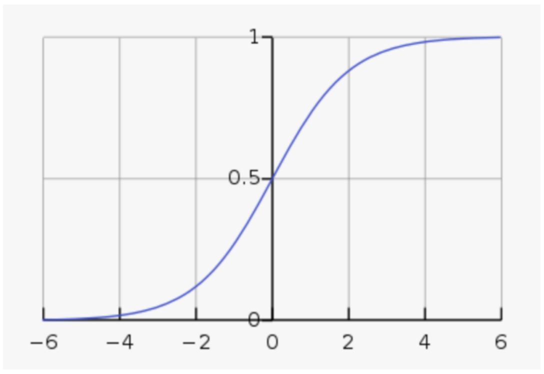
Introduction

The Multi-Layer Perceptron is a fundamental neural network that consists of Affine Layers, Activation Functions, and a Loss Function. To implement an MLP, a solid understanding of linear algebra, calculus, and probability is necessary. The main idea of MLP is to modify the parameters to minimize the model's loss. To achieve this, the backpropagation algorithm plays a significant role in computing the partial loss with respect to each parameter, which gradually enhances the model's performance.

Experiment setups

A. Sigmoid functions

The sigmoid function is an S-shaped function commonly used as an activation function. If the value of sigmoid(x) is greater than 0.5, it is classified as one class, while if it is less than or equal to 0.5, it is classified as the other class.



When given x, the sigmoid function will output 1/(1+e^(-x)).

During backpropagation, when given the output of sigmoid y, the derivative_sigmoid will compute y*(1-y).

B. Neural network

Here are some common notations:

W : weight

X : input or the result during forward

b : bias

y_true : the ground truth of x

lr : learning rate

The Multi-Layer Perceptron structure is designed as follow:

Input Layer matrix : n(number of data) * dimension of each data

Hidden Layer 1 : X2 = sigmoid(X1W1+b1).

Hidden Layer 2 : X3 = sigmoid(X2W2+b2).

Output Layer : output = sigmoid(X3W3+b3).

Compute the mean square error : np.mean(np.square(output-y_true))

C. Backpropagation

Using backpropagation to compute partial loss with respect to each parameter :

dLoss/db3 = (output – target) * derivative_sigmoid(output)

dLoss/dW3 = X3.T x dLoss/db3

dLoss/dX3 = dLoss/db3 x W3.T

dLoss/db2 = derivative_sigmoid(X3) * dLoss/dX3

……

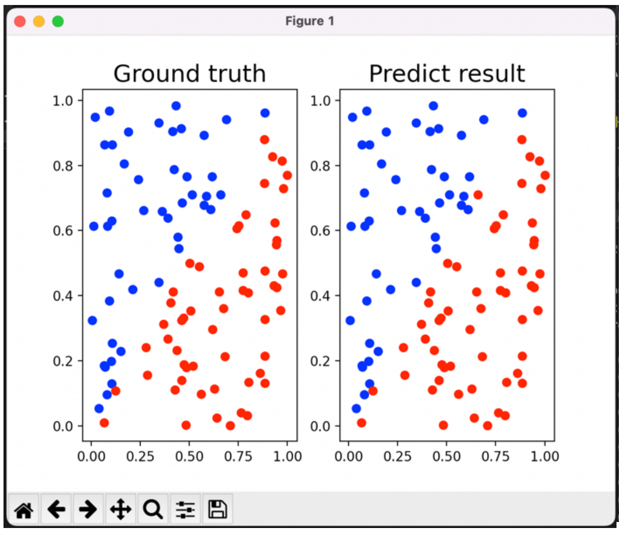
D. Update parameter

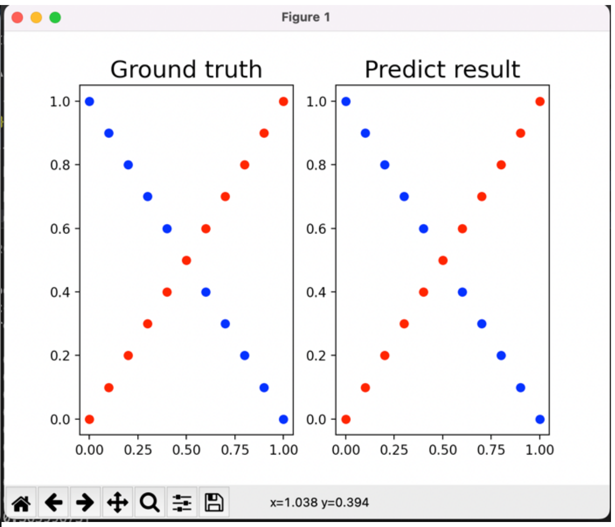
W = W – lr * gradient

b = b – lr * np.sum(gradient,axis=0)

Results of your testing

A. Screenshot and comparison figure

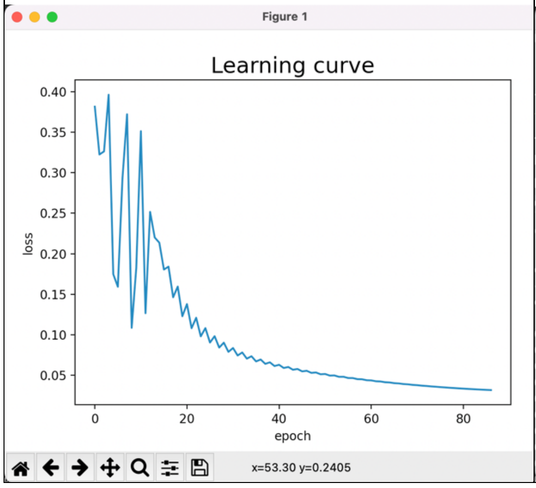
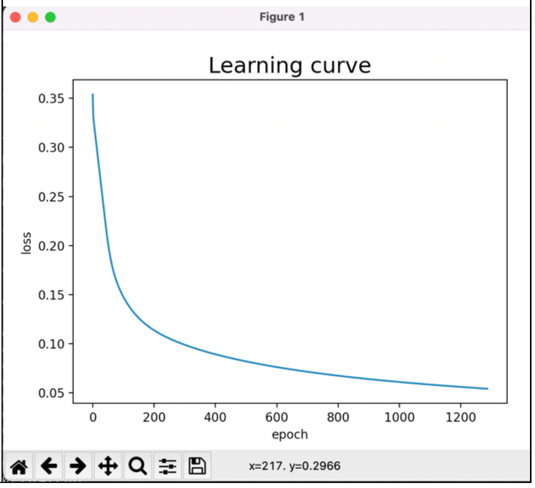




B. Show the accuracy of your prediction

	generate_linear	generate_XOR_easy
Accuracy	100.0	100.0
Loss	0.031645443077537818 71	0.054235702900778601 23

C. Learning curve (loss, epoch curve)

generate_linear	generate_XOR_easy
	

Discussion

A. Try different learning rates

generate_linear

Learning rate	0.1	0.01	0.001
Accuracy	??	100.0	100.0
Epoch For 5 trials	20000 up	159+110+103+115+1469	6169+2343+19238+43543+271

In this experiment, I try different learning rate to solve the linear problem. The best learning rate is 0.01, and the most time-consuming learning rate is 0.1. It's loss decrease extremely slow. (e.g., 10^-9 each epoch)

The choice of learning rate does affect the training process.

B. Try different numbers of hidden units

`generate_XOR_easy`			
W2_row x W2_col	**10 x 10**	**100 x 100**	**1000 x 1000**
Accuracy	**100.0**	**100.0**	**??**
Epoch	**16528+9216+8392 +8843+4077**	**1357+976+1555 +2764+59**	**6000 up**

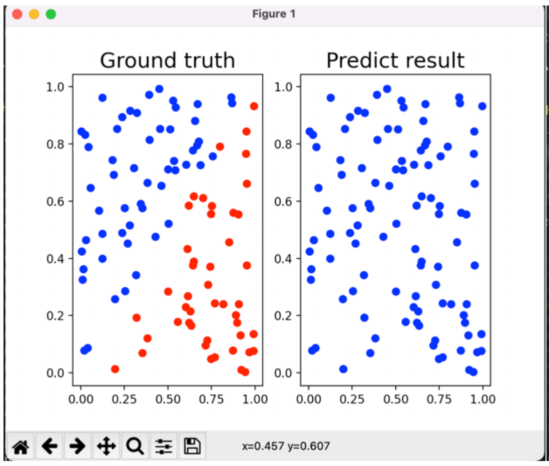
In this experiment, I try different learning rate to solve the xor problem. The best hidden unit is **100 x 100**, and the most time-consuming learning rate is **1000 x 1000**. It's loss decrease extremely slow. (e.g., 10^-6 each epoch),

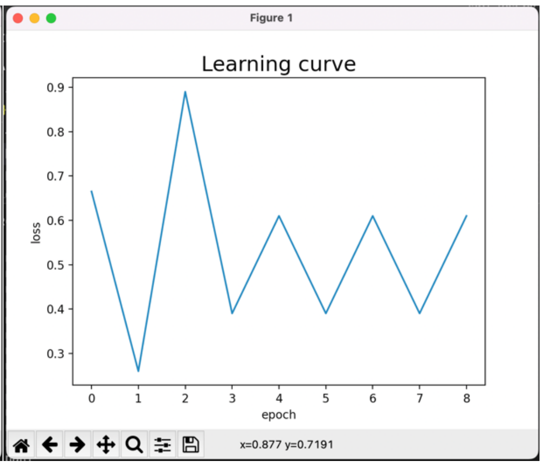
The choice of numbers of hidden units does affect the training process.

C. Try without activation functions

It is impossible to train a model without an activation function. Theoretically, a neural network without an activation function cannot learn non-linear transformations required to solve problems like XOR.

In my experiment, another issue is that the output can become too large as the data flows through the layers, which cause the network to predict all instances as same class.





D. Anything you want to share

Initializing weights can significantly affect the training process and the number of epochs required to converge.

The hyperparameter momentum m can also significantly affect the training process.

Extra

A. Implement different optimizers.

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \eta \frac{\partial L}{\partial \boldsymbol{W}} \tag{6.3}$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} + \boldsymbol{v} \tag{6.4}$$

I implemented an optimizer named Momentum, which can help increase the speed of convergence during training.

```
generate_XOR_easy
```

optimizer	**SGD**	**Momentum**
Accuracy	**100.0**	**100.0**
Epoch	**1357+976+1555** **+2764+59**	**833+823+896+524+438**

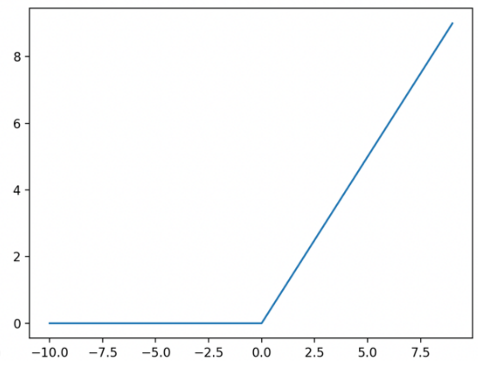
B. Implement different activation functions.

I also implement a RELU function with code:

```
return np.maximum(0, x)
```

The derivative RELU is implemented as follow:

```
    relu_grad = np.zeros_like(x)
    relu_grad[x>=0] = 1
    return relu_grad
```



`generate_XOR_easy`

HiddenLayer1 Activate Function	**Sigmoid**	**RELU**
Accuracy	**100.0**	**100.0**
Epoch	**1357+976+1555 +2764+59**	**796+328+152+127+104**

It seems that using RELU as an activation function in HiddenLayer1 perform better than using Sigmoid.