Deep Learning and Practice

Lab2

310551092 戴晨軒

Project github：https://github.com/nomiaro/Deep-Learning-and-Practice/tree/main/Lab2-Backpropagation

---

Introduction

In this lab, our task goal is to implement a two-hidden-layer neural network using Numpy and the python standard library to predict the color of input random 2D coordinates.

Experiment setups

A. Sigmoid functions:

In the Sigmoid function, I just put the input into it in the forwarding pass and use the recorded input with the loss return from back to calculate its derivative loss in the backpropagation.

```
class sigmoid:
    def __init__(self):
        pass

    def forward(self, x):
        self.input = x
        return 1.0 / (1.0+np.exp(-x))

    def backprop(self, loss):
        derivative_sig = self.forward(self.input) * (1-self.forward(self.input))
        return derivative_sig * loss
```

B. Neural network:

In the neural network, I create a linear class to generate weights matrices. And then take turns using the linear class and the Sigmoid function to achieve forwarding pass. Finally, use the loss function to calculate the loss.

```
# hyperparameters
epoch_num = 100000
lr = 0.001
```

```
class linear:
    def __init__(self, input_dim, output_dim):
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.w = np.random.randn(input_dim, output_dim)
        self.gradient = np.zeros_like(self.w)

    def forward(self, x):
        self.input = x
        out = x @ self.w
        return out
```

```
class Model:
    def __init__(self, input_dim=2, hidden_dim=10, output_dim=1):
        self.fc1 = linear(input_dim, hidden_dim)
        self.fc2 = linear(hidden_dim, hidden_dim)
        self.fc3 = linear(hidden_dim, output_dim)

        self.sig1 = sigmoid()
        self.sig3 = sigmoid()
        self.sig2 = sigmoid()

    def forward(self, x):
        out = self.fc1.forward(x)
        out = self.sig1.forward(out)
        out = self.fc2.forward(out)
        out = self.sig2.forward(out)
        out = self.fc3.forward(out)
        out = self.sig3.forward(out)

        return out
```

```
    def CrossEntrpyLoss(self, pred, label):
        loss = -(label*np.log2(pred) + (1.0-label)*np.log2(1-pred))
        return loss
```

C. Backpropagation:

In backpropagation, I alternately use the sigmoid function and the linear class to propagate the loss through the neural network from end to start and use the chain rule to compute the gradients of the weights in the linear class. Finally, the network is updated using the weights minus the corresponding gradients multiplied by the learning rate.

```
def backprop(self, pred, label):
    derivative_loss = -label*(1/pred) + (1-label)*(1/(1-pred))
    derivative_loss = self.sig3.backprop(derivative_loss)
    derivative_loss = self.fc3.backprop(derivative_loss)
    derivative_loss = self.sig2.backprop(derivative_loss)
    derivative_loss = self.fc2.backprop(derivative_loss)
    derivative_loss = self.sig1.backprop(derivative_loss)
    derivative_loss = self.fc1.backprop(derivative_loss)

def update(self, lr):
    self.fc1.update(lr)
    self.fc2.update(lr)
    self.fc3.update(lr)
```

Backpropagation in the sigmoid function.

```
def backprop(self, loss):
    derivative_sig = self.forward(self.input) * (1-self.forward(self.input))
    return derivative_sig * loss
```

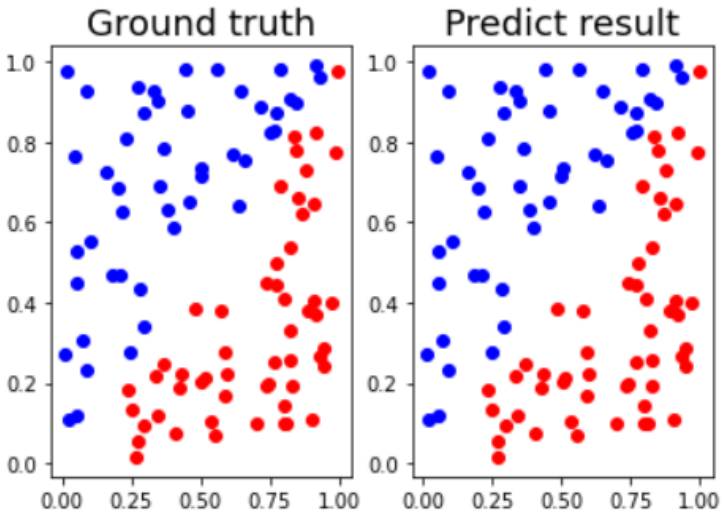
Backpropagation in the linear class.

```
def backprop(self, loss):
    self.gradient = self.input.T @ loss

    loss = loss @ self.w.T
    return loss

def update(self, lr):
    self.w -= lr * self.gradient
```

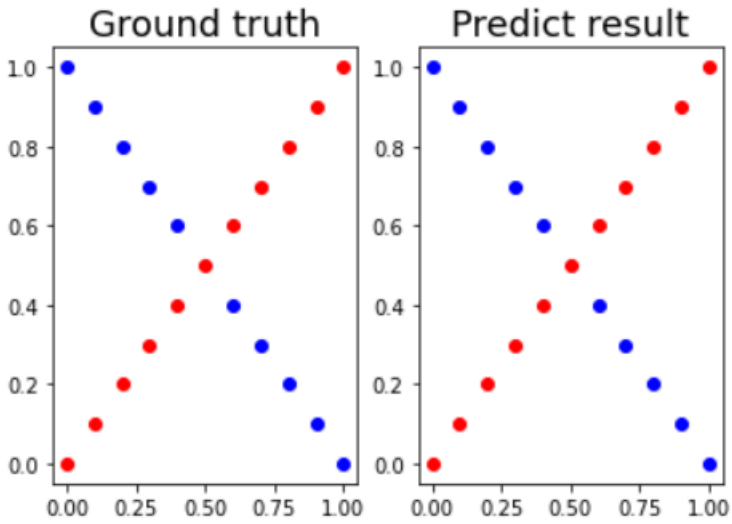
Results of your testing

A. Screenshot and comparison figure

Linear



XOR



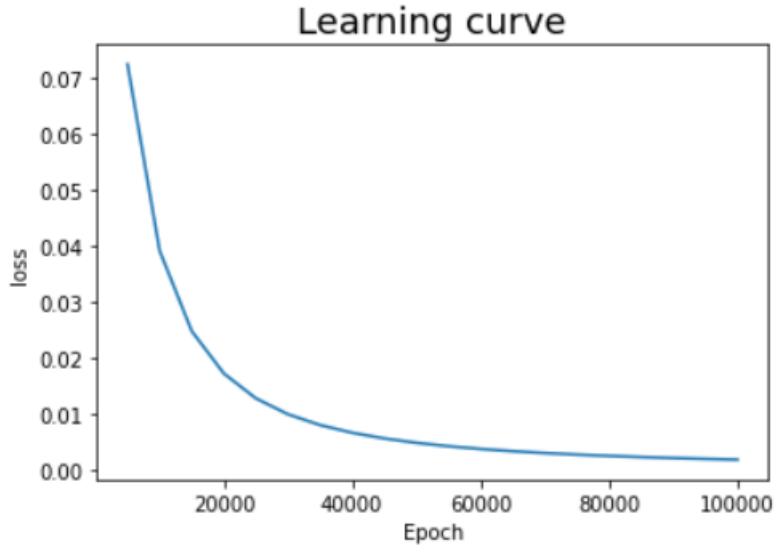
B. Show the accuracy of your prediction

```
Linear predict:
[[1.43510935e-11]
 [5.80406958e-12]
 [1.00000000e+00]
 [7.44829632e-12]
 [1.00000000e+00]
 [1.60078015e-11]
 [6.93161641e-09]
 [8.80987351e-11]
 [5.50176120e-12]
 [1.23593363e-08]
 [1.00000000e+00]
 [5.96255962e-12]
 [1.85372320e-05]
 [1.00000000e+00]
 [1.00000000e+00]
 [7.05001724e-12]
 [9.99991143e-01]
 [6.07197956e-12]
 [5.75929246e-12]
 [9.37003098e-12]
 [9.99999984e-01]
 [9.99999393e-01]
 [7.78196505e-12]
 [1.00000000e+00]
 [1.40402451e-09]
 [9.99999951e-01]
 [1.00000000e+00]
 [6.40546024e-12]
 [1.00000000e+00]]
```

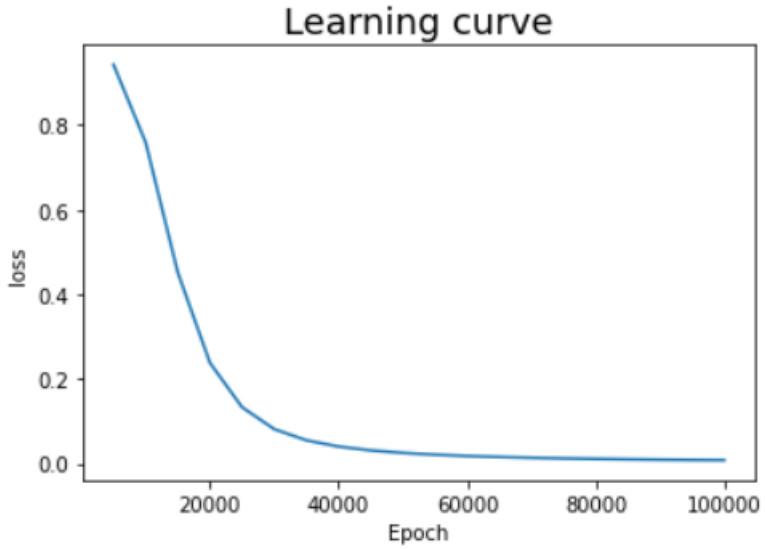
```
XOR predict:
[[2.78711088e-04]
 [9.98017977e-01]
 [1.16170694e-03]
 [9.97778979e-01]
 [3.53414147e-03]
 [9.97238478e-01]
 [6.63308652e-03]
 [9.95937136e-01]
 [8.36462959e-03]
 [9.81255829e-01]
 [8.20212283e-03]
 [6.98710644e-03]
 [9.75612585e-01]
 [5.53459540e-03]
 [9.98105581e-01]
 [4.23854668e-03]
 [9.98879145e-01]
 [3.20816103e-03]
 [9.99024958e-01]
 [2.43021937e-03]
 [9.99056436e-01]]
```

C. Learning curve (loss, epoch curve)

Linear



XOR

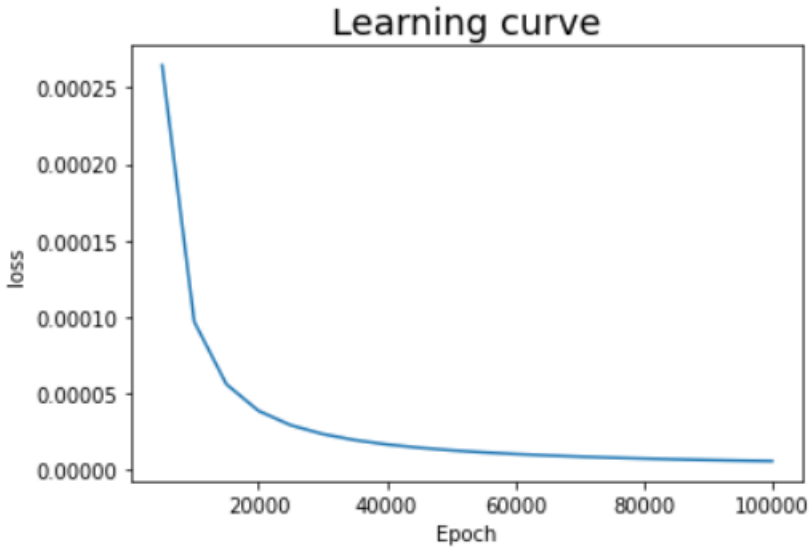


Discussion

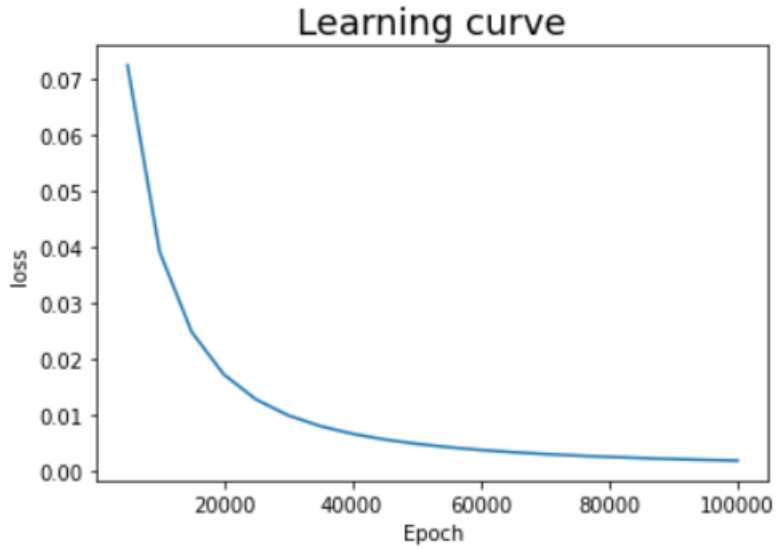
A. Try different learning rates

Learning Rate	Test Accuracy (Linear)
0.1	100%
0.001	100%
0.00001	98%

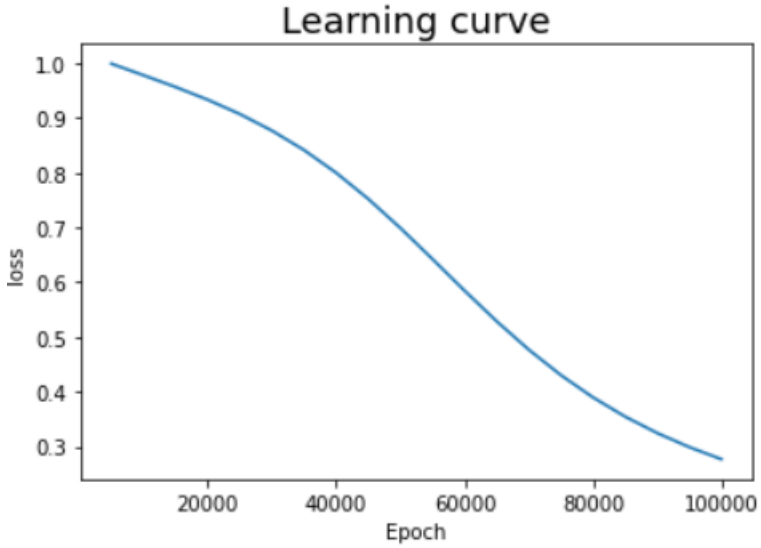
Learning Rate = 0.1



Learning Rate = 0.001



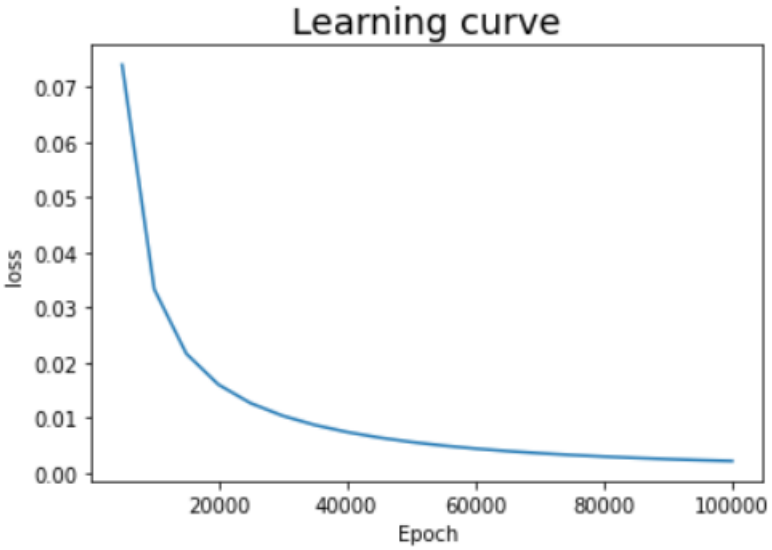
Learning Rate = 0.00001



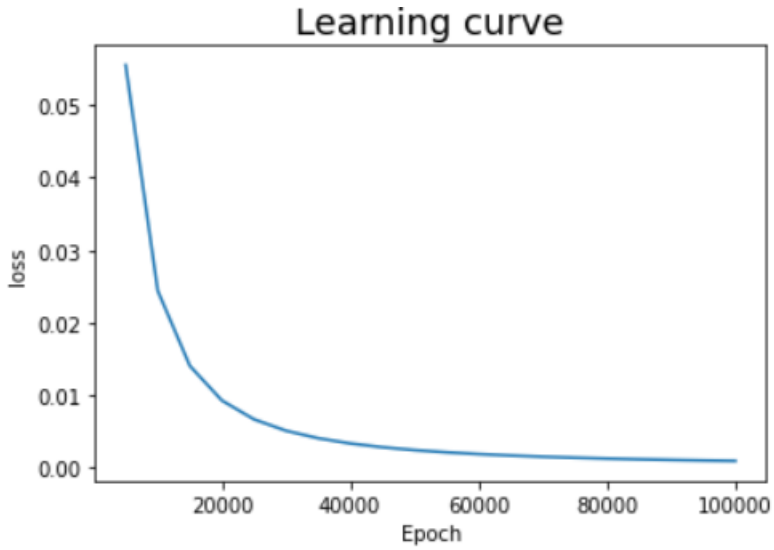
B. Try different numbers of hidden units

Num of hidden units	Accuracy
2	100%
5	100%
10	100%

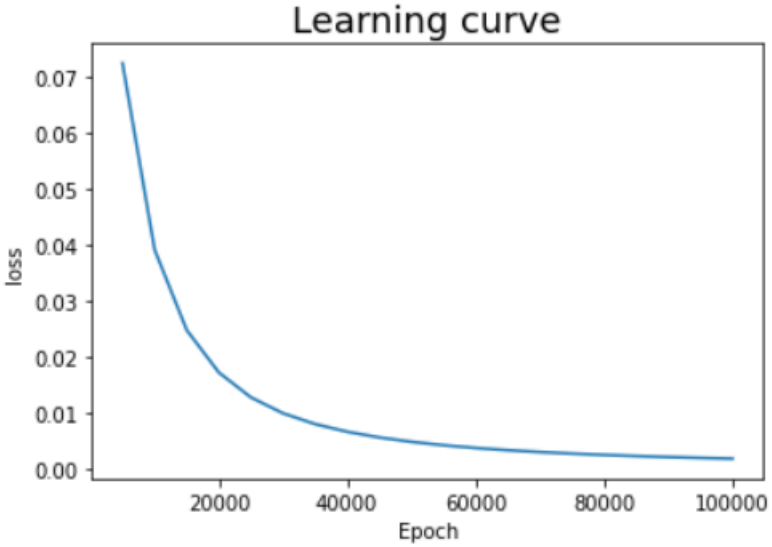
Hidden units = 2



Hidden units = 5



Hidden units = 10



I think it's because our data is too simple, when we use a different number of hidden units, the accuracy and the learning curve are not much different.

C. Try without activation functions

Activation Function	Accuracy
w	100%
w/o	0%

My network will be broken without the activation function, because the loss will explode.

D. Anything you want to share

In part C, I also try to change the loss function from the cross-entropy loss to the MSE loss, and add normalization function, but the loss still cannot converge.,

Extra

B. Implement different activation functions.

Activation Function	Accuracy
Sigmoid	100%
tanh	99%

In this part, since tanh is supported by NumPy, I choose it as the alternative activation function. We can see that because of the simplicity of the data, tanh also has a nice performance.

```python
class tanh:
    def __init__(self):
        pass

    def forward(self, x):
        self.input = x
        return np.tanh(x)

    def backprop(self, loss):
        derivative_tanh = 1.0 / (np.cosh(self.input)**2)
        return derivative_tanh * loss
```