

NYCU DL

Lab1 – Backpropagation

312554006 羅名志

1. Introduction

The lab is about implementing a neural network with an input layer, two hidden layers, and an output layer (as the diagram shown below). The most important part is to design the backward mechanism in the model and to update the model weights to perform better. Through this lab, we need to learn how to hand crafted a simple neural network, design the forward mechanism with different activation functions, and most importantly, to update model weights based on chain rules and different optimizers.

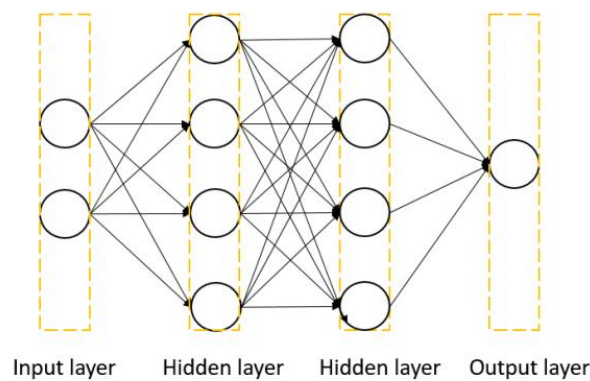


Figure 1. Two-layer neural network

Credited to: Lab1-Backpropagation_spec.pdf provided by TAs

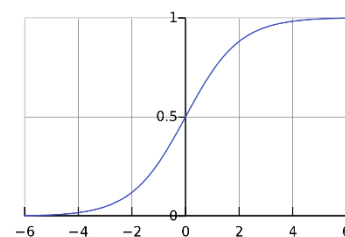
2. Experiment setups

A. Sigmoid functions:

Based on the definition of the sigmoid function, the output is ranges from 0 to 1. Below are the code snippet and the visualization of sigmoid function.

```
# activation functions and their derivatives
def sigmoid(x):
    return 1.0/(1.0 + np.exp(-x))

def derivative_sigmoid(x):
    return np.multiply(x, 1.0 - x)
```



B. Neural network

The design of the neural network can be easily divided into two parts: Forward and Backward. The following is how the model be trained and tested.

(1) The constructor of class object SimpleNN :

```
class SimpleNN:
    """
    Simple neural network with two hidden layers
    """
    def __init__(self, h1_size, h2_size, lr=0.01, iters=1000, activationf='sigmoid', optimizer='gd'):
        self.h1_size = h1_size
        self.h2_size = h2_size
        self.lr = lr
        self.iters = iters
        self.activationf = activationf
        self.optimizer = optimizer
        self.lossf = cross_entropy

        # define activation function
        if self.activationf == 'sigmoid':
            self.act = sigmoid
            self.diff_act = derivative_sigmoid
        elif self.activationf == 'relu':
            self.act = sigmoid
            self.diff_act = derivative_sigmoid
        elif self.activationf == 'tanh':
            self.act = tanh
            self.diff_act = derivative_tanh
        else:
            self.act = without_act
            self.diff_act = derivative_without_act
```

(2) Before training, we need to initialize the weights of the model:

```
def init_params(self):
    """
    Define initial weights of our model
    """
    self.W = [np.random.randn(2, self.h1_size), np.random.randn(self.h1_size, self.h2_size), np.random.randn(self.h2_size, 1)]
    self.b = [np.zeros((1, self.h1_size)), np.zeros((1, self.h2_size)), np.zeros((1, 1))]
    self.z = [None, None, None]
    self.a = [None, None, None]
    self.momentum_w = [np.zeros_like(w) for w in self.W]
    self.adasquare_w = [np.zeros_like(w) for w in self.W]
    self.adam_w1 = [np.zeros_like(w) for w in self.W]
    self.adam_w2 = [np.zeros_like(w) for w in self.W]
```

* The last four self-objects are aim for different optimizers design

(3) Forward function: using linear transformation and different activation functions to forward each layer's hidden value to the next layer.

```
# input layer
x = X
# first hidden layer
self.z[0] = x@self.W[0]+self.b[0] # (n, h1)
self.a[0] = self.act(self.z[0])
# second layer
self.z[1] = self.a[0]@self.W[1]+self.b[1] # (n,h2)
self.a[1] = self.act(self.z[1])
# output layer
self.z[2] = self.a[1]@self.W[2]+self.b[2] # (nx1)
self.a[2] = sigmoid(self.z[2])
return self.a[2]
```

(4) Backward function: leave to Part C.

C. Backpropagation

(1) Backward function: using chain rule, we can adjust the weights from back layers to front layers of the model.

```
# prevent denominator = 0
pred_y = np.clip(pred_y, 1e-15, 1 - 1e-15)

da_2 = -(Y/pred_y) + (1-Y)/(1-pred_y)
dz_2 = da_2*derivative_sigmoid(self.a[2])
dw_2 = self.a[1].T@dz_2
db_2 = np.sum(dz_2, axis=0, keepdims=True)

da_1 = dz_2@self.W[2].T
dz_1 = da_1*self.diff_act(self.a[1])
dw_1 = self.a[0].T@dz_1
db_1 = np.sum(dz_1, axis=0, keepdims=True)

da_0 = dz_1@self.W[1].T
dz_0 = da_0*self.diff_act(self.a[0])
dw_0 = X.T@dz_0
db_0 = np.sum(dz_0, axis = 0, keepdims=True)

# update params
self.update(dw_2, dw_1, dw_0, db_2, db_1, db_0)
```

(2) Update function: based on the derivative to update each weight by different optimizers.

```
def update(self, dw_2, dw_1, dw_0, db_2, db_1, db_0):
    dw = [dw_0, dw_1, dw_2]
    db = [db_0, db_1, db_2]
    if self.optimizer == 'gd':
        for i in range(len(self.W)):
            self.W[i] -= self.lr*dw[i]
            self.b[i] -= self.lr*db[i]
```

*Due to clarity, here I only show the simplest code with the use of Gradient descent.

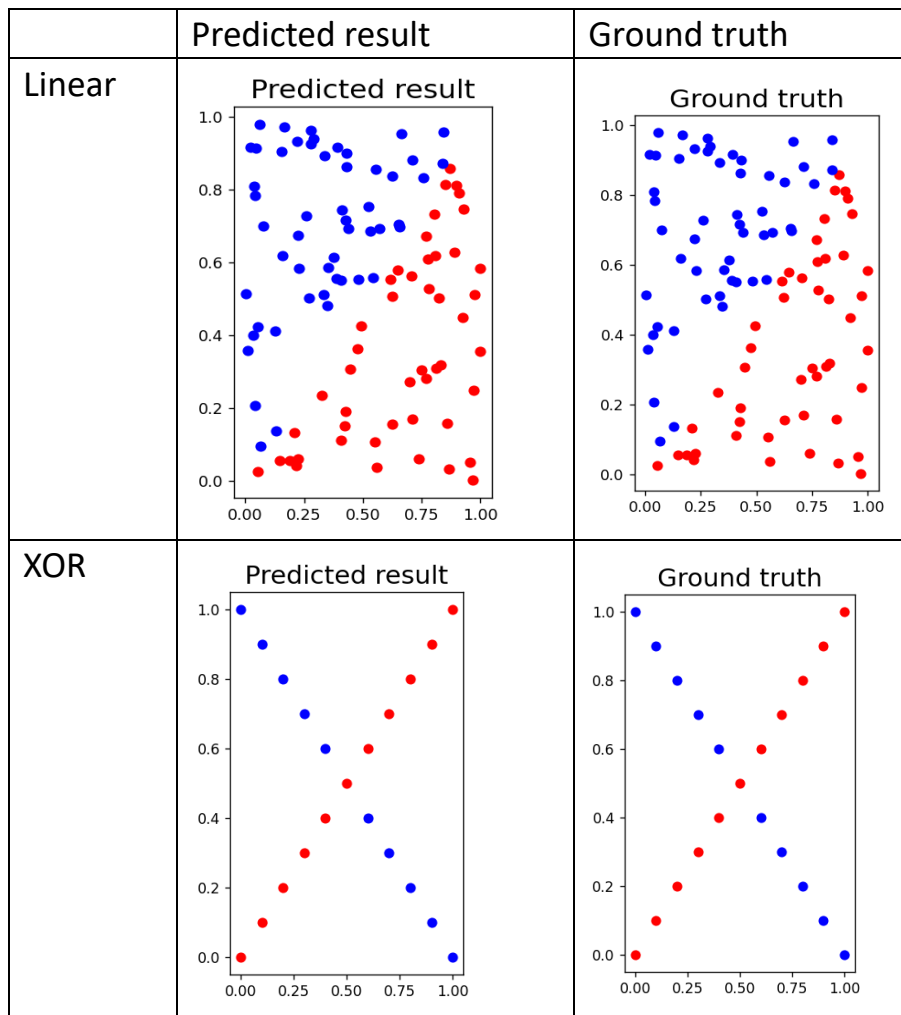
* The default setting for the model is using:

- (1) Optimizer: Gradient descent
- (2) Training epochs: 100,000. With early stop: loss < 0.00001
- (3) Loss function: Cross-entropy (due to our output label is binary)
- (4) Activation function: Sigmoid function
- (5) Hidden units: h1(first hidden layer) = 5 / h2(second hidden layer) = 5
- (6) Learning rate: 0.01

3. Results of your testing

A. Screenshot and comparison figure

* All experiment results shown below are under the default setting

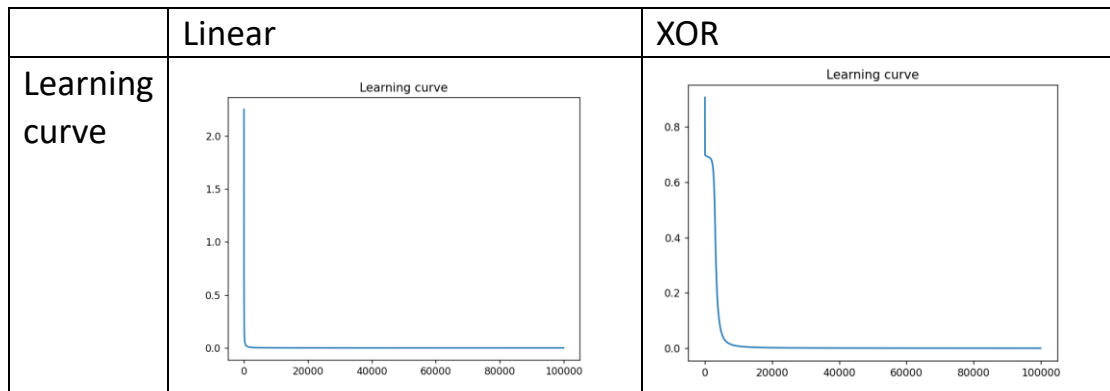


B. Show the accuracy of your prediction

	Linear	XOR
Accuracy	loss=0.00022 accuracy=100.00%	loss=0.00028 accuracy=100.00%

*Get 100% accuracy on both input data. The reason of this outstanding result is probably the task is quite simple for neural network to learn

C. Learning curve (loss, epoch curve)

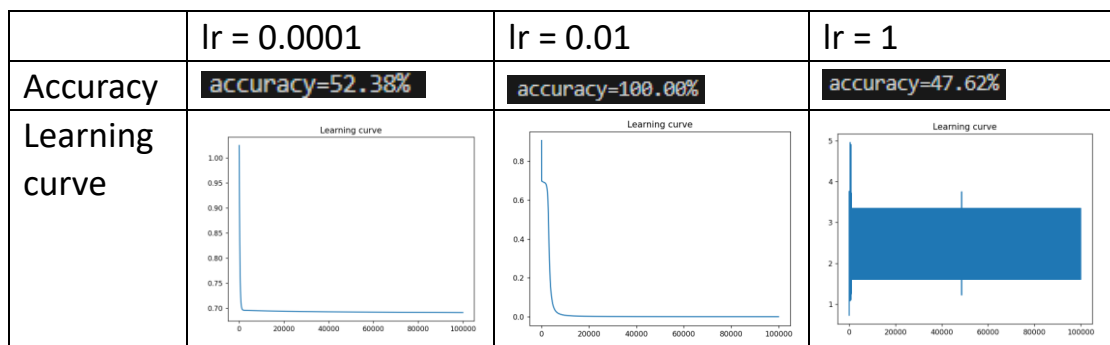


* Perhaps due to the specific pattern of XOR data, the loss drop later than the Linear one. But in both cases, the learning curves show that the model can efficiently learn to perform well and have low loss within 10000 epochs.

4. Discussion

* To keep the report clear and easy to understand, I show the following results half with the use of Linear dataset and half in XOR dataset. Also, all experiment settings are default as explained in Part 2.

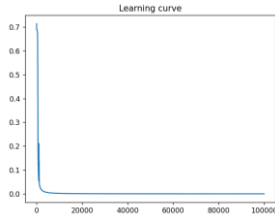
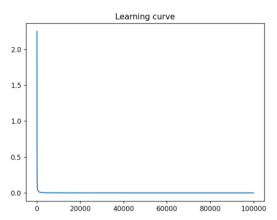
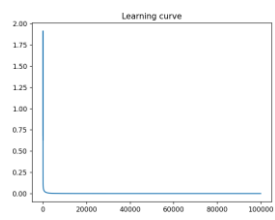
A. Try different learning rates



* Using XOR as input

* The result shows that our setting with lr = 0.01 get the best accuracy. Moreover, if we set the learning rate too large, the model won't able to converge, showing the oscillation like the right subgraph above. Last, if the learning rate too small, the speed of convergence will be slower.

B. Try different numbers of hidden units

	$h(h_1, h_2) = (1,1)$	$h(h_1, h_2) = 5$	$h(h_1, h_2) = 25$
Accuracy	accuracy=100.00%	accuracy=100.00%	accuracy=100.00%
Learning curve			
loss	loss=0.00011	loss=0.00005	loss=0.00004

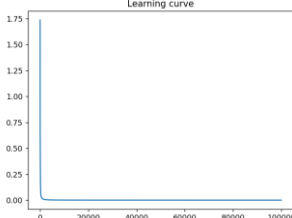
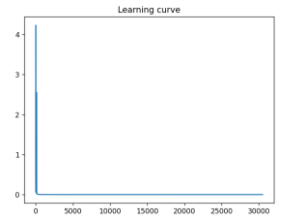
* Using Linear input

* Although the accuracy achieves 100%, the number of hidden units still significantly affects the model. For example, we can observe from the learning curve and loss results that if the number of hidden units is extremely small (e.g., 1), the loss would be high and may yield poor results with more complicated input data. Similarly, when the number of hidden units is relatively large, although we may achieve better results, convergence is slower, leading to longer training times.

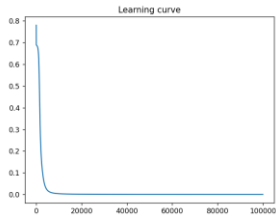
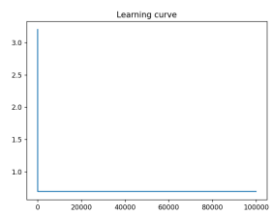
C. Try without activation functions

* Because there is an important difference between the results using Linear and XOR in this case, both cases are used here to demonstrate the difference.

(1) Linear input

	w/ activation	w/o activation
Accuracy	accuracy=100.00%	accuracy=100.00%
Learning curve		
Loss	loss=0.00003	loss=0.00001

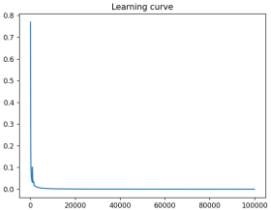
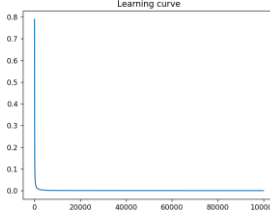
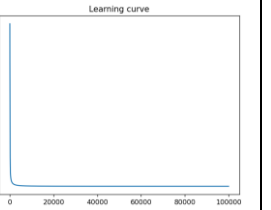
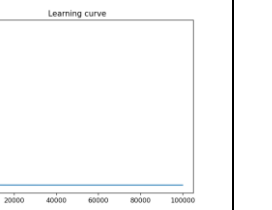
(2) XOR input

	w/ activation	w/o activation
Accuracy	accuracy=100.00%	accuracy=52.38%
Learning curve		
Loss	loss=0.00012	loss=0.69201

* In the Linear input case, the model without activation function yield better result (with smaller loss). This is probably because the input is 'linear' and easy for the model to learn without using the 'nonlinear' function (activation function). However, in the latter case, the result of the model without using activation function get extremely bad result, showing that for nonlinear input like XOR data, it's important to introduce the activation function.

5. Extra

A. Implement different optimizers

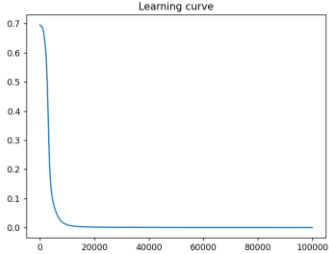
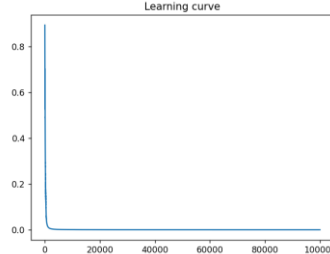
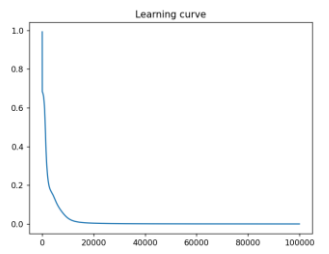
	Gradient descent	Momentum	Adagrad	Adam
Accuracy	accuracy=100.00%	accuracy=100.00%	accuracy=100.00%	accuracy=100.00%
Learning curve				

* Using the Linear input

* To prevent additional complexity and overfitting problems, it is reasonable to optimize only the weights using different optimizers, while keeping the bias part optimized by gradient descent method.

* The result is similar in XOR input case. When using different optimizer, the momentum or decreasing learning rate can help the model converge faster like the result showed above.

B. Implement different activation functions

	Sigmoid	Tanh	Relu
Accuracy	accuracy=100.00%	accuracy=100.00%	accuracy=100.00%
Learning curve			

* Using XOR input

* The result is similar in Linear input, and the tanh activation function under the default setting converges faster. These activation functions are all suitable for both input cases, perhaps we need a more complicated datasets to understand their differences and advantages.