DL_LAB1_Report

學號：312554004　　姓名：林垣志

(1) Introduction:

In this lab, I will implement a simple neural networks with two hidden layers, use forward propagation to get the output value (prediction), calculate the error between the predicted value and the actual value and send it back through backpropagation function, so as to update the weights of each layer, keep the steps until the number of epochs is reached to make the prediction closer to the actual value.

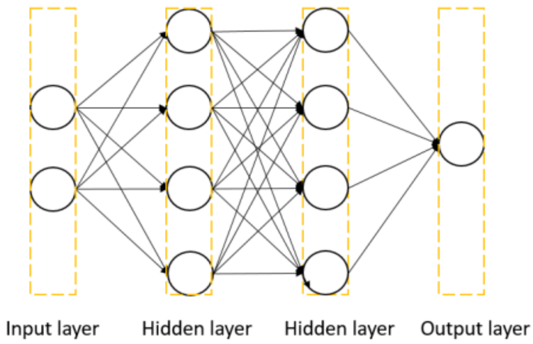
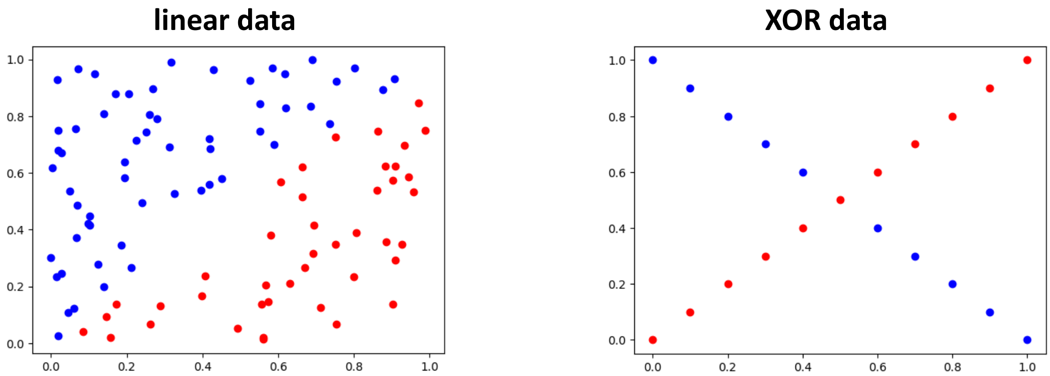


Figure 1. Two-layer neural network

Our task is to classify the dataset provided by TA. The dataset include linear data and XOR data. The architecture implements the "Layers" class and "NeuralNetwork" class. Through the Layers class to add how many layers I will use in total, and then use NeuralNetwork class to construct the network. The parameter are set by ourselves, include optimizer、epochs, and learning rate.



(2) Experiment setups:

(a) Sigmoid function

The figure (code) below shows the process of sigmoid function and derivative. They are both implemented in "NeuralNetwork" class. Sigmoid function is used in forward propagation, and the derivative of sigmoid function is used in backpropagation.

```
"""
Define the sigmoid activator, the derivative ==> y' = y(1 - y)
"""
def sigmoid(self, x, der = False):
    if der == True:
        y_prime = np.multiply(x, 1.0 - x)
    else: 
        y_prime  = 1.0 / (1.0 + np.exp(-x))
    return y_prime
```

The derivative of sigmoid function is shown below:

$$\sigma(\mathrm{x}) = \frac{1}{1+e^{-x}}$$

$$\frac{d(\sigma(\mathrm{x}))}{dx} = \frac{0*(1+e^{-x}) - (1)*(e^{-x}*(-1))}{(1+e^{-x})^2}$$

$$\frac{d(\sigma(\mathrm{x}))}{dx} = \frac{(e^{-x})}{(1+e^{-x})^2} = \frac{1\text{-}1+(e^{-x})}{(1+e^{-x})^2} = \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2}$$

$$\frac{d(\sigma(\mathrm{x}))}{dx} = \frac{1}{1+e^{-x}} * \left(1 - \frac{1}{1+e^{-x}}\right) = \sigma(\mathrm{x})(1-\sigma(\mathrm{x}))$$

(b) Neural network

The "NeuralNetwork" class has function such as forward, backward, fit, etc., and I choose MSE (mean square error) as my loss function, the formula is shown below.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - pred\_y_i\right)^2$$

$$\frac{d\ MSE\left(y_i\ ,\ pred\_y_i\right)}{d\ pred\_y_i} = \frac{-2*\left(y_i\ ,\ pred\_y_i\right)}{N}$$

```
""" Add Layers class ,count the layers what we have """
class Layers :
    def layer(units = 4, activation = 'sigmoid'):

""" two hidden_layer """
class NeuralNetwork:
    def __init__(self):

    def set_learning_rate(self, eta):

    def set_optimizer(self, optimizer = ''):

    def set_epoch(self, epoch):

    def Clear(self):

    """ decide what unit & activation func """
    def add(self, layer = (4, '') ):

    """ Forward Propagation """
    def forward(self, X):

    """ Back Propagation """
    def backward(self, ground_truth, pred_y):

    """ Back Propagation without activation"""
    def backward_noAct(self, ground_truth, pred_y):

    """ Update weights """
    def update(self):

    """  Define the sigmoid activator, the derivative ==> y' = y(1 - y) """
    def sigmoid(self, x, der = False):

    """
    Define the Rectifier Linear Unit (ReLU) activator
    the derivative ==> y' = 1 if y > 0 , y' = 0 if y <= 0
    """
    def ReLU(self, x, der = False):

    """  Define the tanh activator, the derivative ==> y' = 1 - y^2 """
    def tanh(self, x, der = False):

    def derivative_mse(self, y, pred_y):

    """ loss func """
    def loss_mse(self, prediction, ground_truth):

    """ print graph """
    def show_result(self, x, y):

    """ Training Model """
    def fit(self, inputs_list, targets_list):

    """ Testing """
    def predict(self, X):
```

(c) Backpropagation

The figure (code) below shows the process of backpropagation. we need to use backpropagation to update model weights, then we should use chain rule to compute $\frac{dL}{dW_1}, \frac{dL}{dW_2}, \frac{dL}{dW_3}$, and the backward gradient will be calculated according to the activation function used.

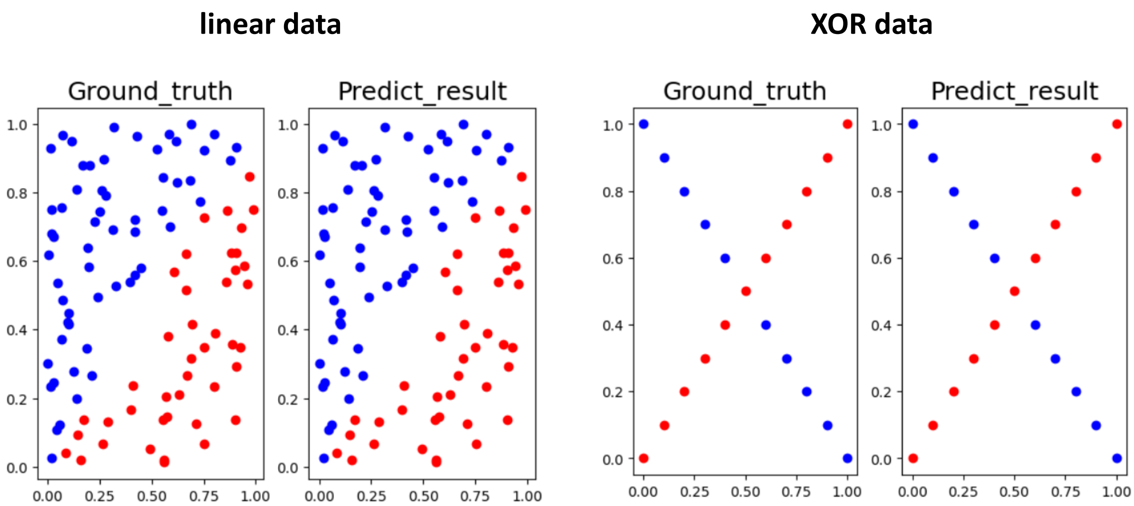
```python
""" Back Propagation """
def backward(self, ground_truth, pred_y):
    backward_gradient = None
    for j in range(len(self.z) - 1, -1, -1):
        if j == len(self.z) - 1:
            delta_Out = self.derivative_mse(ground_truth, pred_y)

            if self.activation[len(self.allLayer) - j - 1] == 'sigmoid':
                backward_gradient = np.multiply(self.sigmoid(pred_y, der = True), delta_Out)

            elif self.activation[len(self.allLayer) - j - 1] == 'tanh':
                backward_gradient = np.multiply(self.tanh(pred_y, der=True), delta_Out)

            elif self.activation[len(self.allLayer) - j - 1] == 'ReLU':
                backward_gradient = np.multiply(self.ReLU(pred_y, der=True), delta_Out)

            self.dl_dw[j] = self.z[j-1].T @ backward_gradient
        else:
            if self.activation[len(self.allLayer) - j - 1] == 'sigmoid':
                backward_gradient = np.multiply(self.sigmoid(self.z[j], der=True), backward_gradient @ self.W[j+1].T)

            elif self.activation[len(self.allLayer) - j - 1] == 'tanh':
                backward_gradient = np.multiply(self.tanh(self.z[j], der=True), backward_gradient @ self.W[j+1].T)

            elif self.activation[len(self.allLayer) - j - 1] == 'ReLU':
                backward_gradient = np.multiply(self.ReLU(self.z[j], der=True), backward_gradient @ self.W[j+1].T)

            if j == 0:
                self.dl_dw[j] = self.x.T @ backward_gradient
            else:
                self.dl_dw[j] = self.z[j-1].T @ backward_gradient
```

After computing the gradients, we can update the model weights.

$$W_1 = W_1 - learning\_rate * \nabla W_1$$
$$W_2 = W_2 - learning\_rate * \nabla W_2$$
$$W_3 = W_3 - learning\_rate * \nabla W_3$$

(3) Results of your testing:

(a) Screenshot and comparison figure



The above chart shows that the network accurately predicts the answers.

(b) Show the accuracy of your prediction

linear data

```
Epoch 0 loss : 0.3277935312661204
Epoch 500 loss : 0.026226085873059034
Epoch 1000 loss : 0.014460312810533562
Epoch 1500 loss : 0.01011844764200088
Epoch 2000 loss : 0.007627796056786311
Epoch 2500 loss : 0.005982193597947854
Epoch 3000 loss : 0.004803961550849091
Epoch 3500 loss : 0.003920464091435555
Epoch 4000 loss : 0.003241411711016297
Epoch 4500 loss : 0.002711950245257585
Epoch 5000 loss : 0.0022947195027362716
Epoch 5500 loss : 0.0019627079681656247
Epoch 6000 loss : 0.0016959089369927133
Epoch 6500 loss : 0.0014793719279280844
Epoch 7000 loss : 0.001301874940179907
Epoch 7500 loss : 0.001154957994867108
Epoch 8000 loss : 0.001032208381143062
Epoch 8500 loss : 0.0009287327121109972
Epoch 9000 loss : 0.0008407685428302175
Epoch 9500 loss : 0.00076539930412367
```

```
 [2.87381535e-04]
 [9.82583278e-01]
 [9.32676274e-01]
 [9.99994777e-01]
 [9.99997930e-01]]
Accuracy: 100.0%
```

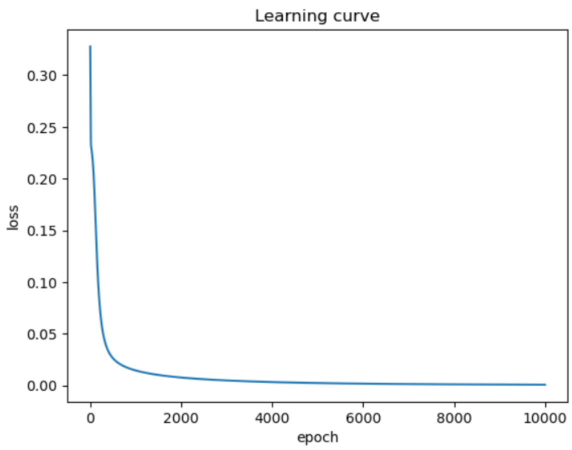
XOR data

```
Epoch 0 loss : 0.24891065760766548
Epoch 500 loss : 0.23854069382025195
Epoch 1000 loss : 0.21349099986683265
Epoch 1500 loss : 0.18972581855746726
Epoch 2000 loss : 0.17469187505256836
Epoch 2500 loss : 0.16428712229181397
Epoch 3000 loss : 0.05114153865173264
Epoch 3500 loss : 0.022659614349025317
Epoch 4000 loss : 0.010467173212606018
Epoch 4500 loss : 0.005564496717093072
Epoch 5000 loss : 0.0034479501019247984
Epoch 5500 loss : 0.0023833148063334485
Epoch 6000 loss : 0.0017741934709360728
Epoch 6500 loss : 0.0013907452635803235
Epoch 7000 loss : 0.0011316852425364214
Epoch 7500 loss : 0.0009470597113961478
Epoch 8000 loss : 0.0008099192794241787
Epoch 8500 loss : 0.0007046481225287711
Epoch 9000 loss : 0.0006216606242271037
Epoch 9500 loss : 0.0005547881724635741
```

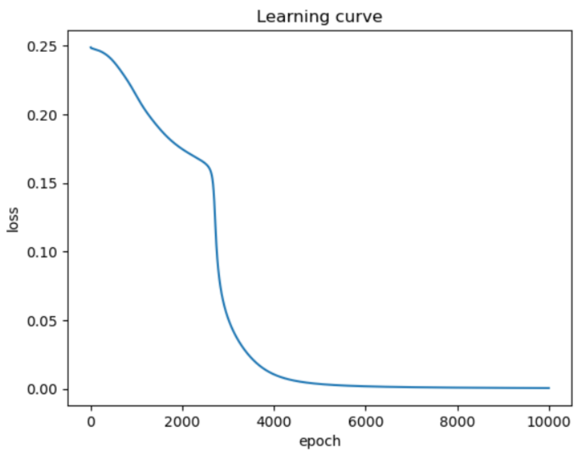
```
 [4.54367403e-03]
 [9.93941337e-01]
 [3.03301517e-03]
 [9.91033618e-01]]
Accuracy: 100.0%
```

(c) Learning curve (loss, epoch curve)

linear data



XOR data



It can be seen from the above table that in linear data, which is a relatively simple problem. It's unlikely XOR data will last somewhere for a period of epoch before it converges, and then it will start to decline.

(4) Discussion:

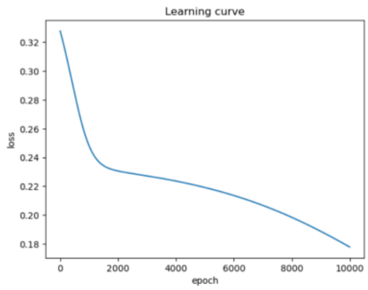
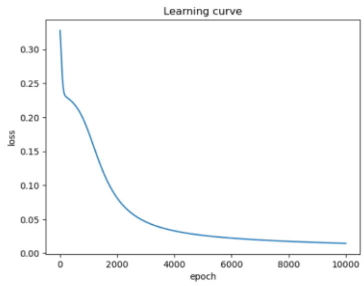
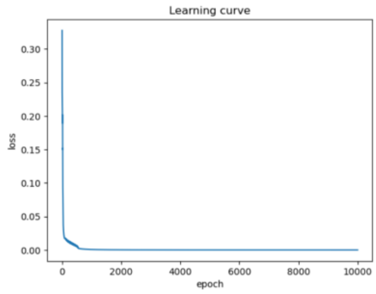
(a) Try different learning rates

Accuracy of learning rate：

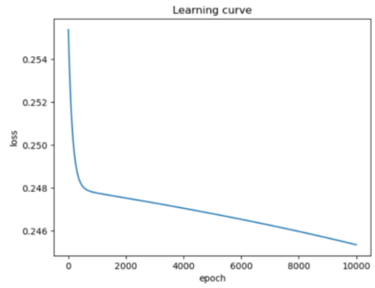
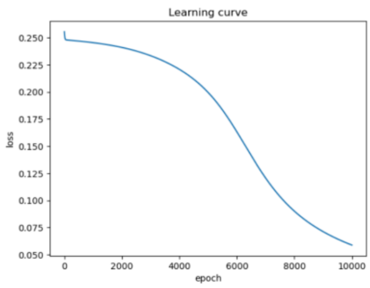
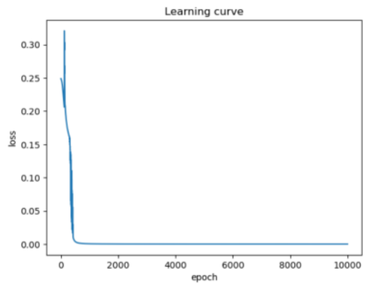
Learning rate	linear data	XOR data
0.01	83%	61.90%
0.1	100%	90.47%
10	100%	100%

As we can see, when learning rate increasing to 10, the accuracy of linear data model and XOR data model also increase.

linear data’s learning curve：

Learning rate = 0.01	Learning rate = 0.1	Learning rate = 1
		

XOR data’s learning curve：

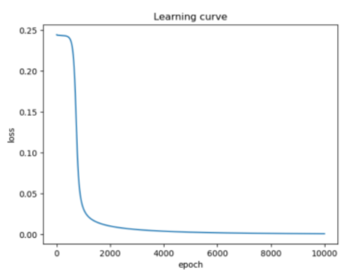
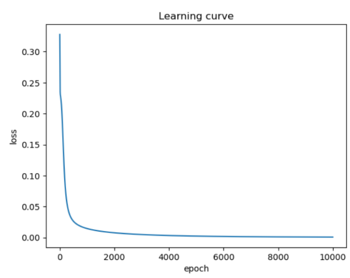
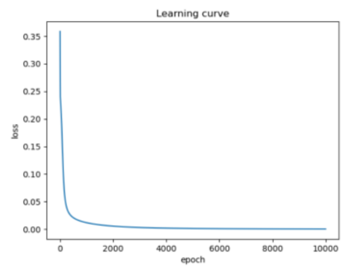
Learning rate = 0.01	Learning rate = 0.1	Learning rate = 1
		

As I set the hidden layer has 4 units both two datasets. It can be seen from the above two table that when the learning rate is too large, the learning curve will oscillate a little bit and even affect the accuracy.

(b) Try different numbers of hidden units

hidden unit	**linear data**	**XOR data**
2	100%	76.19%
4	100%	100%
10	100%	100%

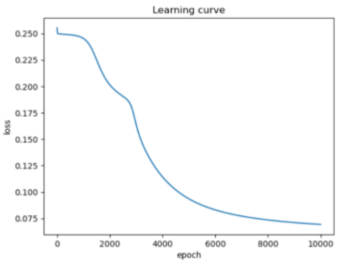
linear data’s learning curve：

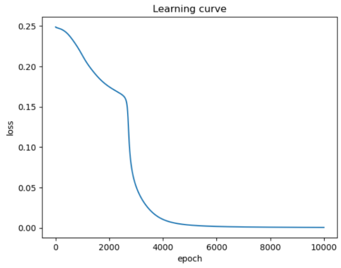
hidden unit = 2	hidden unit = 4	hidden unit = 10
		

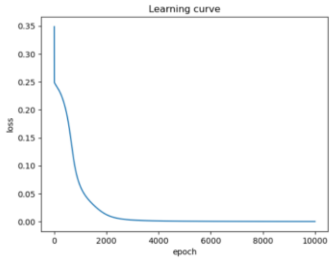
As I set learning rate = 0.1, It can be seen from the above table that when the unit increases, it leads to slower convergence, potentially increasing complexity and the overall loss value.

XOR data's learning curve：

hidden unit = 2　　hidden unit = 4　　hidden unit = 10

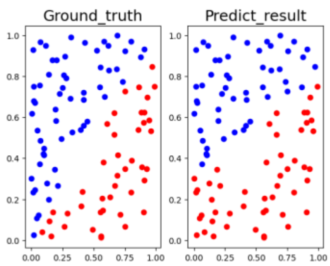
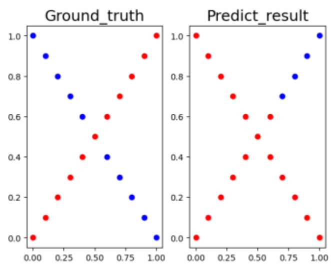
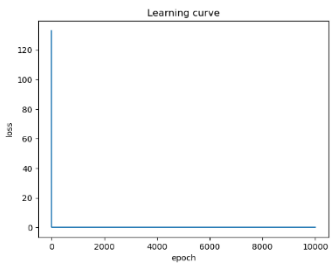
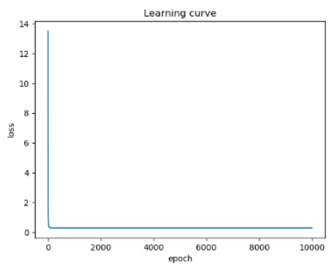






The learning rate is set to the same numbers as linear data. As we can see, there doesn't seem to be much difference in convergence time among different units, but as the number of units increases, the learning curve becomes smoother, indicating better learning performance with more units. Also, we can observed that only two units in each hidden layer of XOR data model are too few to train its model.

(c) Try without activation functions

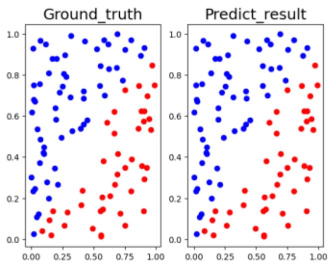
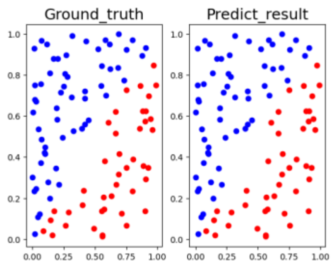
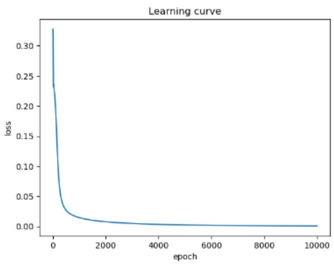
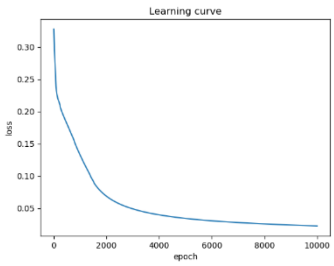
linear data	XOR data
	
[0.67367188] [0.7992729] [0.34441301] [1.01974434]] Accuracy: 87.0%	[0.39370079] [0.70866142] [0.39370079] [0.78740157] [0.39370079]] Accuracy: 33.33333333333333%
	

In this experiment, the learning rate and hidden layer's units are set to the same numbers both two data. linear data model has 87% accuracy, and the accuracy of XOR data model decrease to 33%, it's really worse, but we can observed that without activation has an impact on accuracy. The results show that model without nonlinear activation functions can't classify XOR problem, because it's just like a single layer perceptron, can't solve XOR problem.

(5) Extra:

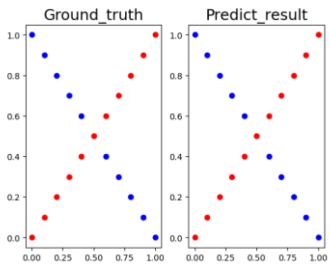
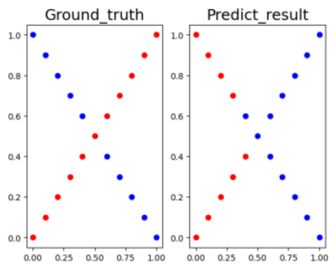
(a) Implement different optimizers.

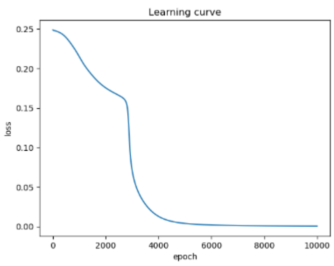
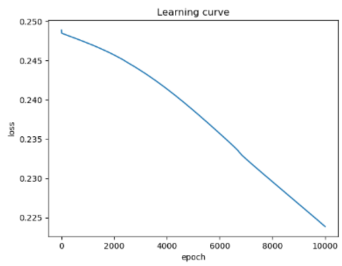
linear data：

<table>
<tr><th>Momentum</th><th>Adagrad</th></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>[9.82735228e-01]
[9.32883167e-01]
[9.99994997e-01]
[9.99997986e-01]]
Accuracy: 100.0%</td><td>[0.67208684]
[0.6364211]
[0.94712031]
[0.9828024]]
Accuracy: 100.0%</td></tr>
<tr><td>
</td><td>
</td></tr>
</table>

As I set learning rate = 0.1 and hidden layer has 4 units, the results shows that optimizer use momentum can reduce the learning rate to prevent oscillation and accelerate convergence, and even the convergence faster than the optimizer with Adagrad.

XOR data：

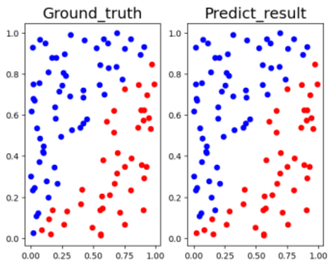
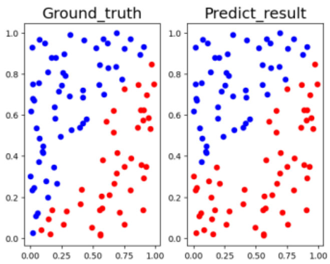
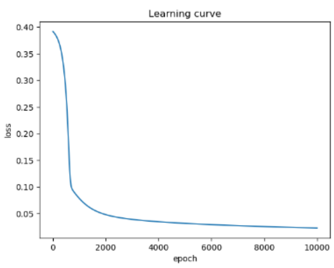
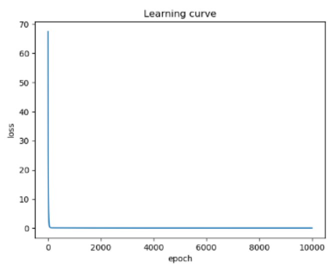
<table>
<tr><th>Momentum</th><th>Adagrad</th></tr>
<tr><td>
</td><td>
</td></tr>
<tr><td>[4.52609250e-03]
[9.93722655e-01]
[3.01936125e-03]
[9.90426793e-01]]
Accuracy: 100.0%</td><td>[0.55659115]
[0.57013502]
[0.55559896]
[0.58240621]]
Accuracy: 52.38095238095239%</td></tr>
</table>

In this experiment, the XOR data model with Adagrad optimizer don't get better performance, but use momentum optimizer can get great performance.

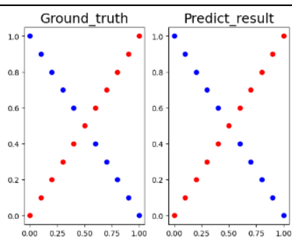
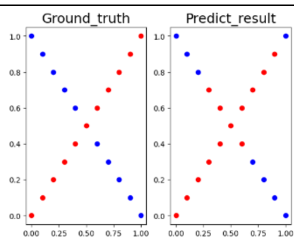
(b) Implement different activation functions.

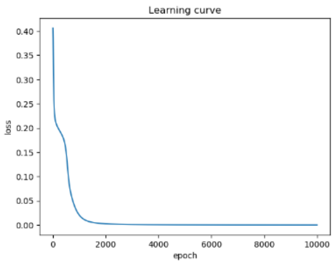
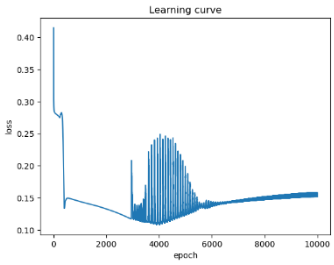
linear data：

Tanh	ReLu
	
[6.63423766e-01] [6.42698912e-01] [9.72564556e-01] [9.76933290e-01]] Accuracy: 98.0%	[0.59698061] [0.6971744] [0.36143548] [1.05795365]] Accuracy: 90.0%
	

The two results use same hidden layer units but different learning rate, ReLU's learning rate = 0.001, because ReLU may be influenced by the vanishing gradient problem.

XOR data：

Tanh	ReLu
	

[-2.23437580e-03] [9.89656676e-01] [-2.58740452e-03] [9.88689657e-01] Accuracy: 100.0%	[0.49045211] [0.94907017] [0.54494679] [1.11821936]] Accuracy: 80.95238095238095%
	

In this experiment, the two results use same hidden layer units and learning rate = 0.1, but performance in ReLU activation function has a little bit of oscillation，I think the learning rate can be to large, and it’s has faster convergence, affect the accuracy.