Lab1: Backpropagation

Wei-Yun Hsu
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University

1. Introduction

In this lab, we will only use Numpy and other python standard libraries to

- implement simple neural networks with two hidden layers
- implement backpropagation
- visualize the training loss and testing results

A. Datasets

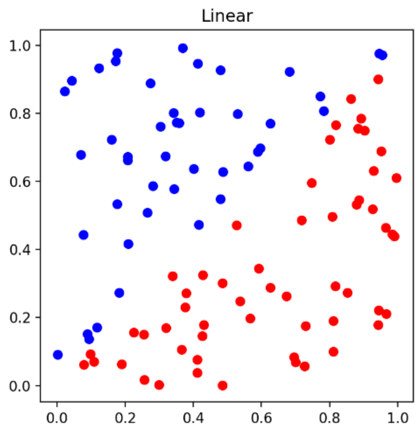
Two data generator in this lab:

- generate_linear

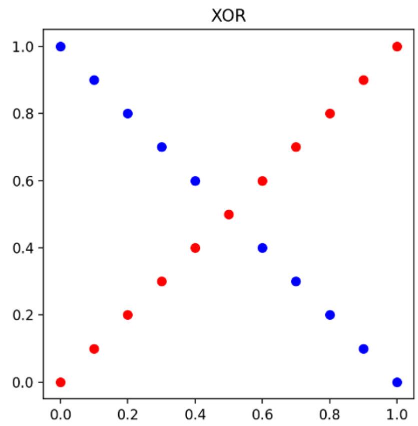
```python
def generate_linear(n=100):

    pts = np.random.uniform(0, 1, (n, 2))
    #print(pts)
    inputs = []
    labels = []
    for pt in pts:
        inputs.append([pt[0], pt[1]])
        distance = (pt[0]-pt[1])/1.414
        if pt[0] > pt[1] :
            labels.append(0)
        else:
            labels.append(1)
    #print(labels)
    return np.array(inputs), np.array(labels).reshape(n, 1)
```

- generate_XOR_easy

```python
def generate_XOR_easy():

    inputs = []
    labels = []

    for i in range(11):
        inputs.append([0.1*i, 0.1*i])
        labels.append(0)

        if 0.1*i==0.5 :
            continue

        inputs.append([0.1*i, 1-0.1*i])
        labels.append(1)

    return np.array(inputs), np.array(labels).reshape(21,1)
```



(a) linear data



(b) XOR data

2. Experiment setups

A. Sigmoid function

I mainly use sigmoid function as my activation function σ. The sigmoid function is defined as

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

For the derivative of sigmoid function, we have

$$\sigma'(x) = -\frac{-e^{-x}}{(1 + e^{-x})^2} = \frac{1}{1 + e^{-x}} \times \frac{-e^{-x}}{1 + e^{-x}} = \sigma(x)\big(1 - \sigma(x)\big)$$

The implementations of sigmoid function and its derivative as shown below.

```
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

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

B. Neutral Network

Each layer is considered with a weight matrix W and the shape is $M \times N$ where M and N are the numbers of input features and output features respectively. The input vector x get output y in a neural layer can be written as

$$z = W^T x + b$$

$$y = \sigma(z)$$

In my implementation, there are four neurons in each hidden layer. You can refer to `Layer` object and `Network` object in `main.py` for more details.

C. Loss function

I use MSE as my loss function $L(\theta)$. Given the output $\hat{y}$ of neural network and the ground truth y, the loss can be written as

$$L(\theta) = MSE(\hat{y} - y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

The implementations of sigmoid function and its derivative as shown below.

```
def mse_loss(y_pred, y_true):
    return np.mean((y_pred-y_true)**2)

def derivative_mse_loss(y_pred, y_true):
    return 2 * (y_pred-y_true) / y_true.shape[0]
```

And then see how to compute gradient by using backpropagation.

D. Backpropagation

The backpropagation algorithm can be divided into two parts: propagation and weight update. To update the weight matrices of the network, we need to compute $\frac{\partial C}{\partial W}$ where C is the cost between $\hat{y}$ and y and minimize C from $L(\theta)$. But $\frac{\partial C}{\partial W}$ is hard to compute, we use chain rule to solve it.

$$\frac{\partial C}{\partial W} = \frac{\partial z}{\partial W}\frac{\partial C}{\partial z}$$

You can refer to `Layer` object and `Network` object in `main.py` for more details.

D-1. Forward

The forward gradient can be calculated by

$$\frac{\partial z}{\partial W} = \frac{\partial (W^T x + b)}{\partial W} = x'$$

where x' automatically extend a column for bias when `forward` function.

D-2. Backward

The backward gradient can be calculated by

$$\frac{\partial C}{\partial z} = \frac{\partial y}{\partial z}\frac{\partial C}{\partial y}$$

First part, we can get $\frac{\partial y}{\partial z}$ by $y = \sigma(z)$ and $\frac{\partial y}{\partial z} = \sigma'(z) = \sigma(z)\big(1 - \sigma(z)\big)$.

Second part, we can consider two cases: output layer and hidden layer. For the output layer, cost C is computed by loss function $L(\theta) = MSE(\hat{y} - y)$

$$\frac{\partial C}{\partial y} = \frac{\partial MSE(\hat{y} - y)}{\partial y}$$

For the hidden layer, calculating $\frac{\partial C}{\partial y}$ is more difficult than the output layer.

$$\frac{\partial C}{\partial y_{this}} = \frac{\partial z_{next}}{\partial y_{this}} \frac{\partial C}{\partial z_{next}}$$

$$\frac{\partial z_{next}}{\partial y_{this}} = {w_{next}}^T, \; z_{next} = y_{this} w_{next}$$

Therefore, we compute from the output layer and send parameters to the previous layer, we can compute $\frac{\partial C}{\partial z}$ every layer.

D-3. Weight update

When we get the forward gradient and backward gradient, the gradient of the weight can be calculated by multiplying these two gradients. Then, we put the new parameter η which is so-called learning rate to update the weight of the neural network. In this lab, I set η to 1.0 by default.

$$W = W - \eta \frac{\partial C}{\partial W}$$

3. Results of your testing

For this section, I use the following configuration

- four neurons in each hidden layer (two hidden layers)
- learning rate is set to 1.0
- activation function is sigmoid function
- the number of epochs is set to 10000

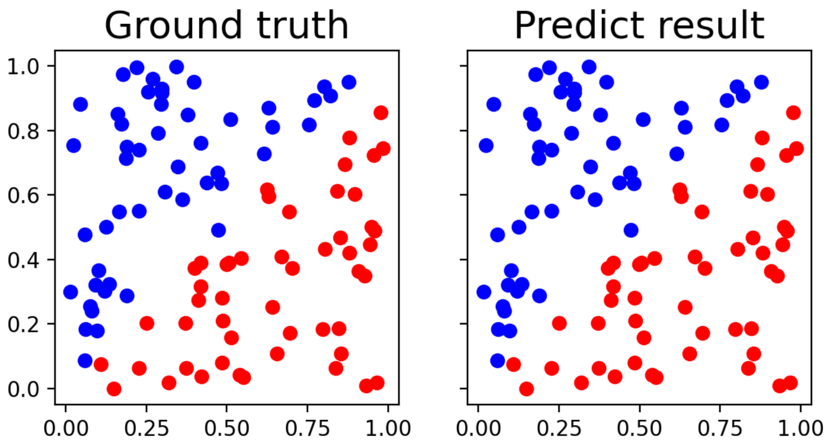
A. Screenshot and comparison figure

From here, the misclassified data will be marked by a black circle.

A-1. Linear data

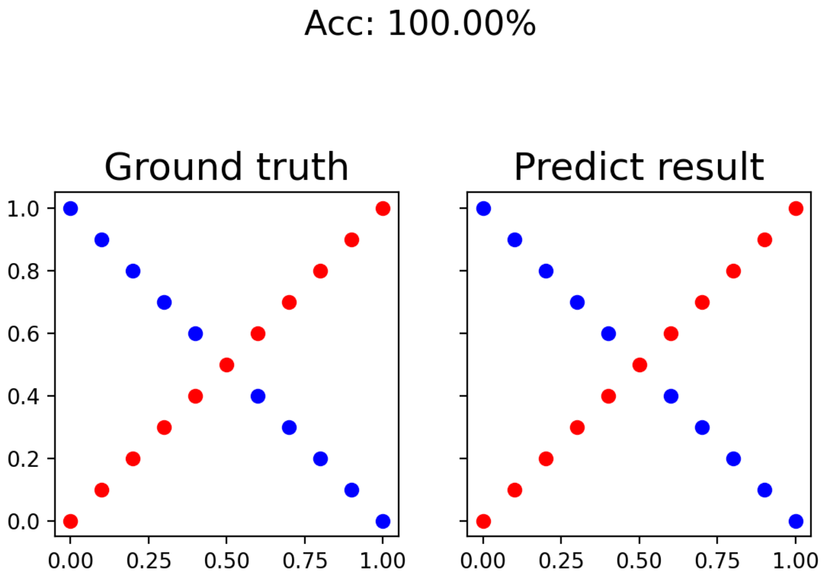
Print training loss	Print testing result
Epoch 0 loss : 0.2729007726939363 Epoch 500 loss : 0.048533941326452494 Epoch 1000 loss : 0.014427102040086275 Epoch 1500 loss : 0.00890961444438997 Epoch 2000 loss : 0.006605916295984738 Epoch 2500 loss : 0.0053609577849786216 Epoch 3000 loss : 0.00458318263347911 Epoch 3500 loss : 0.004047570776328569 Epoch 4000 loss : 0.003652367493349963 Epoch 4500 loss : 0.0033456395515729996 Epoch 5000 loss : 0.003098309401258322 Epoch 5500 loss : 0.00289286557648148 Epoch 6000 loss : 0.002718130887808911 Epoch 6500 loss : 0.0025666287448268127 Epoch 7000 loss : 0.0024331630393618015 Epoch 7500 loss : 0.0023140069591675804 Epoch 8000 loss : 0.0022064165167780454 Epoch 8500 loss : 0.002108326498193257 Epoch 9000 loss : 0.0020181535981444437 Epoch 9500 loss : 0.001934665026134552	[2.10902457e-04] [1.75088496e-04] [1.73292865e-01] [9.99813943e-01] [9.99853961e-01] [6.77267781e-01] [9.99841579e-01] [1.74735869e-04] [3.24615066e-04] [2.46613695e-04] [9.99884593e-01] [9.99884325e-01] [9.79540166e-01] [9.99880010e-01] [2.26666366e-03] [9.99389940e-01] [9.98870888e-01] [9.99768276e-01]] Testing loss: 0.00659643 Acc: 100/100 (100.00%)

Acc: 100.00%



A-2. XOR data

<table>
<tr><th>Print training loss</th><th>Print testing result</th></tr>
<tr><td>Epoch 0 loss : 0.2574909075266212
Epoch 500 loss : 0.24376037365142528
Epoch 1000 loss : 0.207089569806467З
Epoch 1500 loss : 0.08083030245615805
Epoch 2000 loss : 0.027546906515086615
Epoch 2500 loss : 0.011134971976424015
Epoch 3000 loss : 0.005111554265601757
Epoch 3500 loss : 0.002846886124618083
Epoch 4000 loss : 0.0018292641216626434
Epoch 4500 loss : 0.001292988886161031
Epoch 5000 loss : 0.000975269439096211
Epoch 5500 loss : 0.0007703257879861402
Epoch 6000 loss : 0.000629500750072816
Epoch 6500 loss : 0.0005279434242905352
Epoch 7000 loss : 0.0004518750396333452
Epoch 7500 loss : 0.00039313912747969735
Epoch 8000 loss : 0.000346644271291075
Epoch 8500 loss : 0.00030907022218783876
Epoch 9000 loss : 0.0002781710568962003
Epoch 9500 loss : 0.00025237969669470903</td><td>[[0.00151847]
[0.99924064]
[0.0014837]
[0.99949133]
[0.00261496]
[0.99963566]
[0.00799689]
[0.99957433]
[0.02333452]
[0.96643932]
[0.03331251]
[0.02261929]
[0.96334083]
[0.01028697]
[0.99992706]
[0.00431974]
[0.99998221]
[0.00198452]
[0.99998728]
[0.00104773]
[0.99998865]]
Testing loss: 0.00023057
Acc: 21/21 (100.00%)</td></tr>
</table>



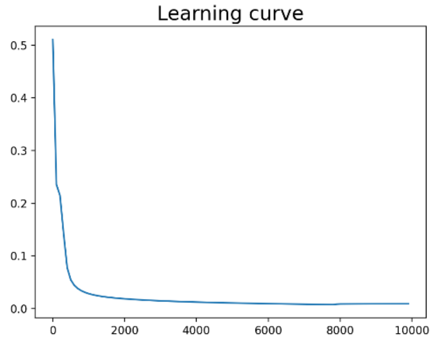
B. Show the accuracy of your prediction

The accuracy of testing data is shown on the section A.

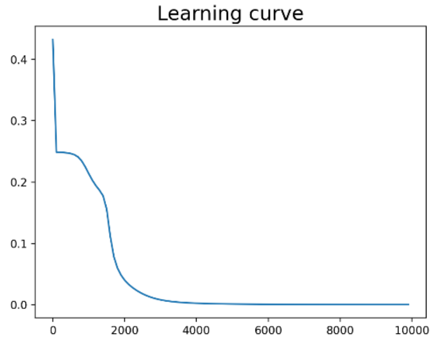
Linear data: 100/100 (100.00%)

XOR data: 21/21 (100.00%)

C. Learning curve (loss, epoch curve)



(a) Linear data



(b) XOR data

4. Discussions

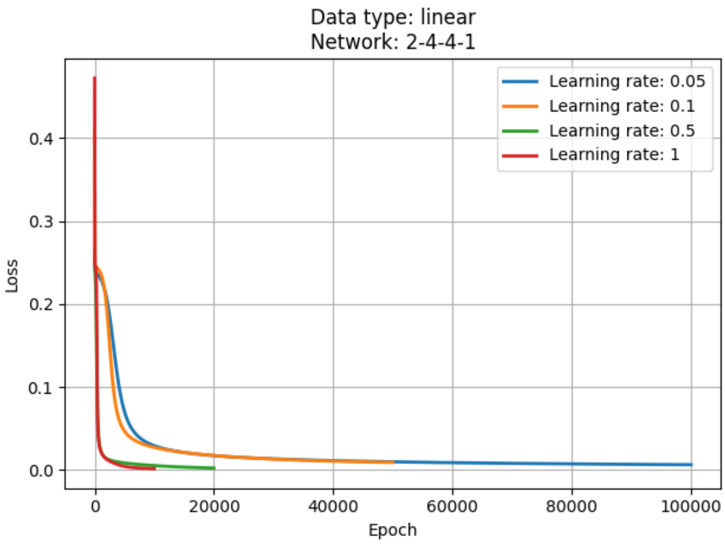
A. Try different learning rates

I used the same network as section 3 and tried 4 different learning rate.

A-1. Linear data

Accuracy of different learning rate:

- Learning rate=0.05: 100/100 (100%)
- Learning rate=0.1: 100/100 (100%)
- Learning rate=0.5: 100/100 (100%)
- Learning rate=1.0: 100/100 (100%)

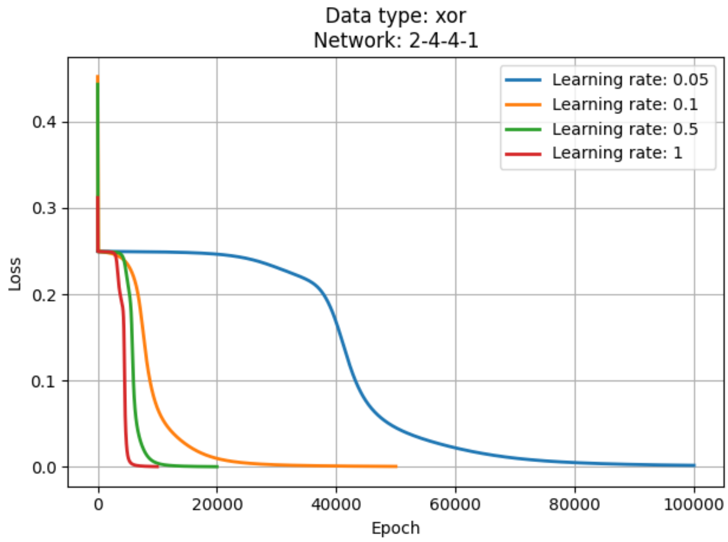


A-2. XOR data

Accuracy of different learning rate:

- Learning rate=0.05: 21/21 (100%)
- Learning rate=0.1: 21/21 (100%)
- Learning rate=0.5: 21/21 (100%)

- Learning rate=1.0: 21/21 (100%)



A-3. Observations

From the results above, it is obvious that the 4 neurons for each hidden layer is enough to achieve the satisfiable performance (>90%). The difference of the four learning rate is the loss decrease slower with the small learning rate and the network convergence requires more epochs.

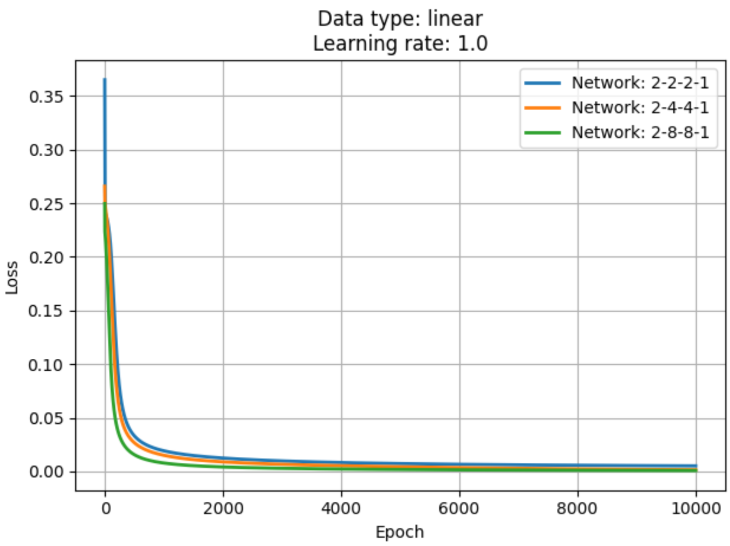
B. Try different numbers of hidden units

Here, I modified the 3 different numbers of hidden units and fixed the learning rate is equal to 1.0.

B-1. Linear data

Accuracy of different numbers of hidden units:

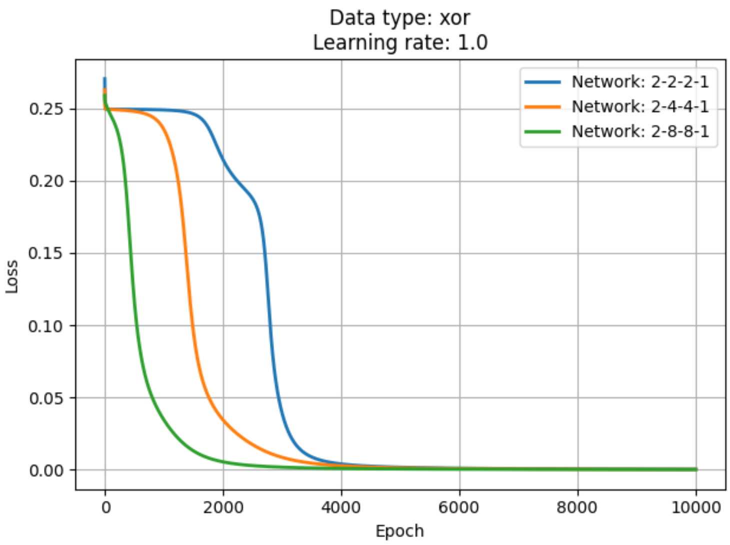
- 2 neurons: 100/100 (100%)
- 4 neurons: 100/100 (100%)
- 8 neurons: 100/100 (100%)



B-2. XOR data

Accuracy of different numbers of hidden units:

- 2 neurons: 21/21 (100%)
- 4 neurons: 21/21 (100%)
- 8 neurons: 21/21 (100%)



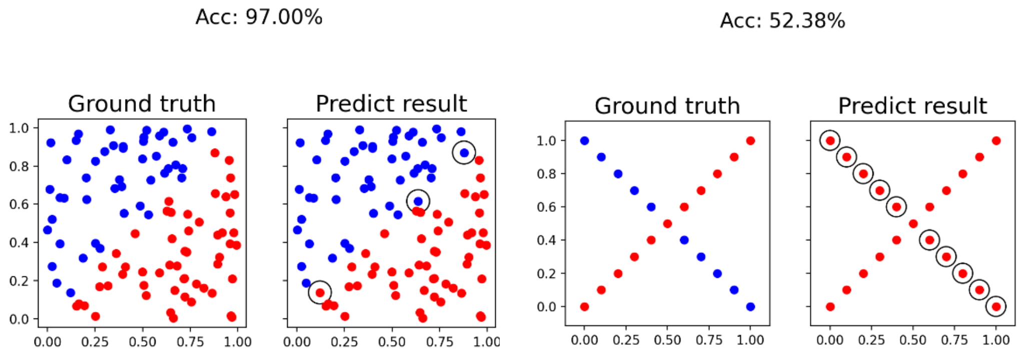
B-3. Observations

From the result of the linear data, there is no great difference between the 3 cases. I think that two neurons is enough to predict this linear data, because this data can be separate by a single straight line. But for the XOR data, it doesn't have a linear solution, the network need to learn more patterns to classify exactly. When we want to classify this data faster, increasing the number of hidden units is a good strategy.

C. Try without activation functions

For this section, I use the following configuration

- four neurons in each hidden layer (two hidden layers)
- learning rate is set to 0.001
- without activation function
- the number of epochs is set to 50000



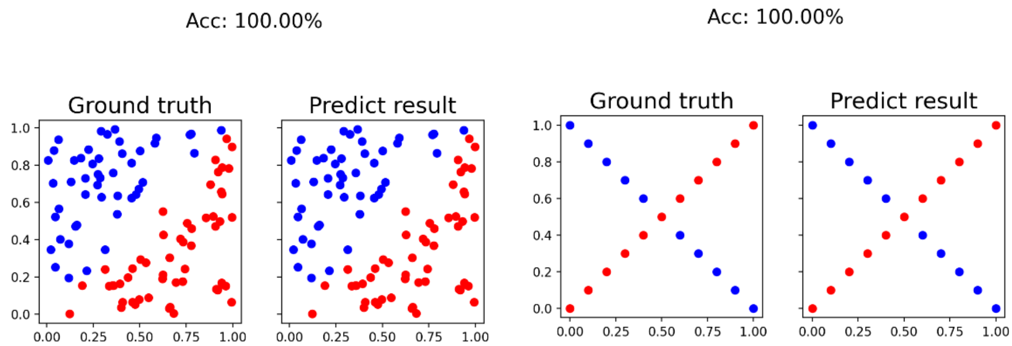
(a) Linear data (b) XOR data

I have already beat the baseline of without using activation function. In the above figure(b), we can see that half of the inputs are misclassified, since the activation function provides the network to solve the non-linear computations. Without activation function, the network can only solve the linear regression problem. Another thing is about learning rate, it is more smaller than the setting of section 3. Without activation function, there is no mechanism to avoid the gradient exploding. If the learning rate is too large, the weight matrices might overflow and result in NaN values during update weight.

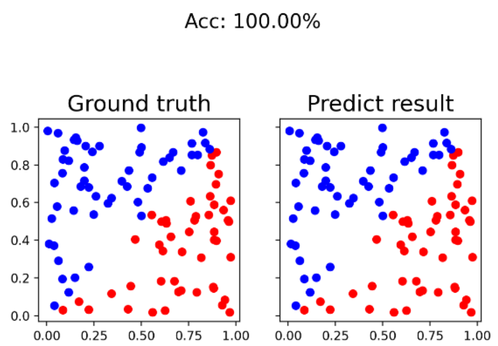
5. Extra

A. Implement different optimizers. (2%)

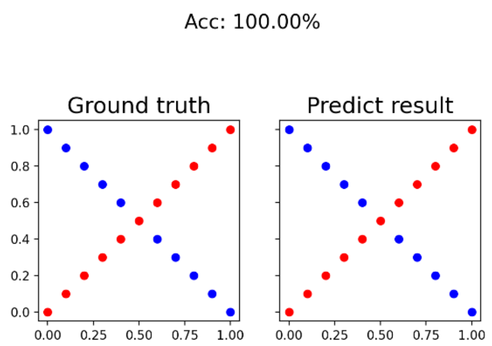
In this section, I use the same configuration as section 3 for Adagrad and momentum respectively.



(a) Linear data using Adagrad (b) XOR data using Adagrad



(c) Linear data using momentum



(d) XOR data using momentum

You can refer to `update` function of `Layer` object in `main.py` for more details.

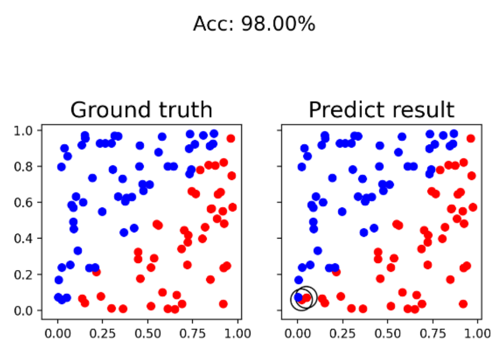
B. Implement different activation functions. (3%)

In this section, I use two different configurations for tanh and ReLU.

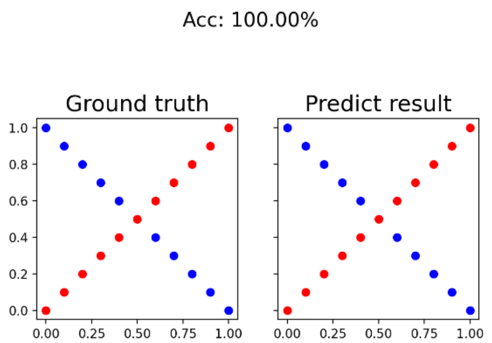
For tanh function,

- four neurons in each hidden layer (two hidden layers)
- learning rate is set to 0.001
- activation function is tanh function
- the number of epochs is set to 50000

```
def tanh(x):
    return np.tanh(x)

def derivative_tanh(x):
    return 1.0 - x ** 2
```



(a) Linear data

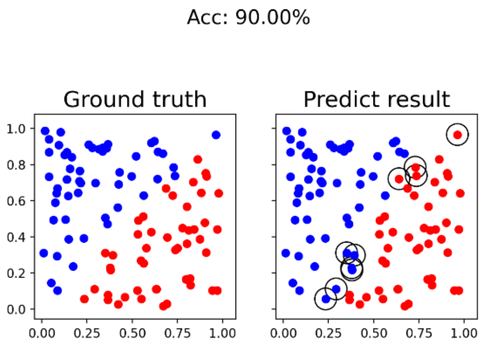


(b) XOR data

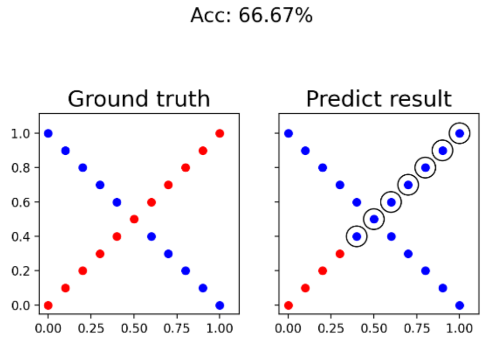
For ReLU function,

- four neurons in each hidden layer (two hidden layers)
- learning rate is set to 0.00002
- activation function is ReLU function
- the number of epochs is set to 100000

```python
def relu(x):
    return np.maximum(0.0, x)

def derivative_relu(x):
    return np.heaviside(x, 0.0)
```



(a) Linear data



(b) XOR data

C. Implement convolutional layers. (5%)