312554012 王偉誠 Lab1 : back-propagation

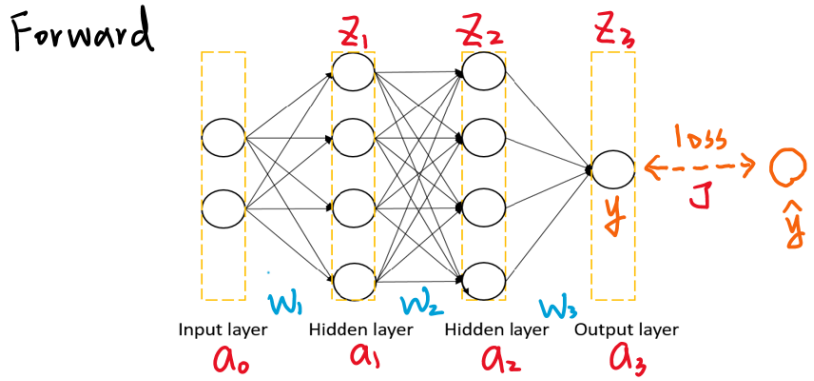
1. Introduction

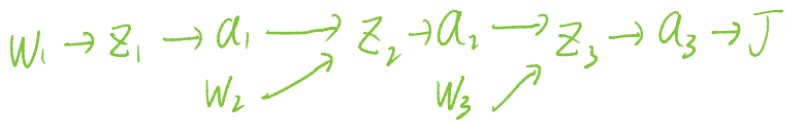
A. LAB objective

This LAB involves constructing a fully connected neural network with two hidden layers. The objective is to classify input data by feeding it with linear and XOR data.

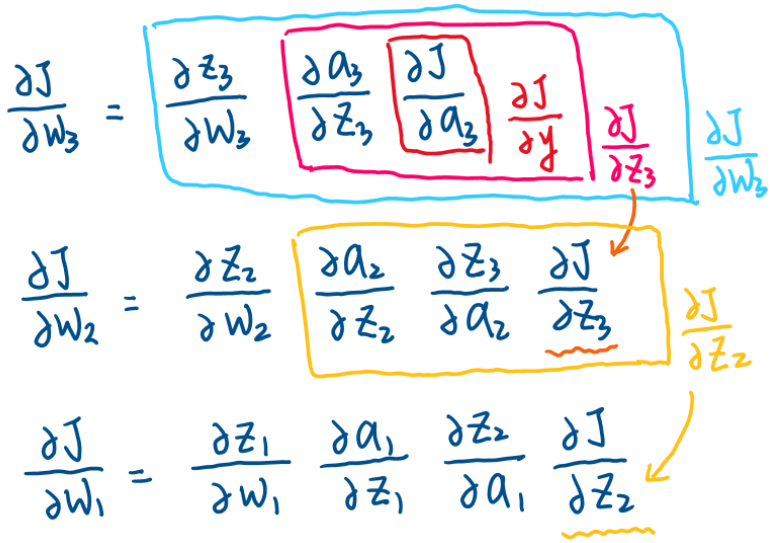
B. Fully connected NN

(1) Based on the diagram of relationships, we can derive the forward computation equations and the diagram illustrating the functional relationships among variables.

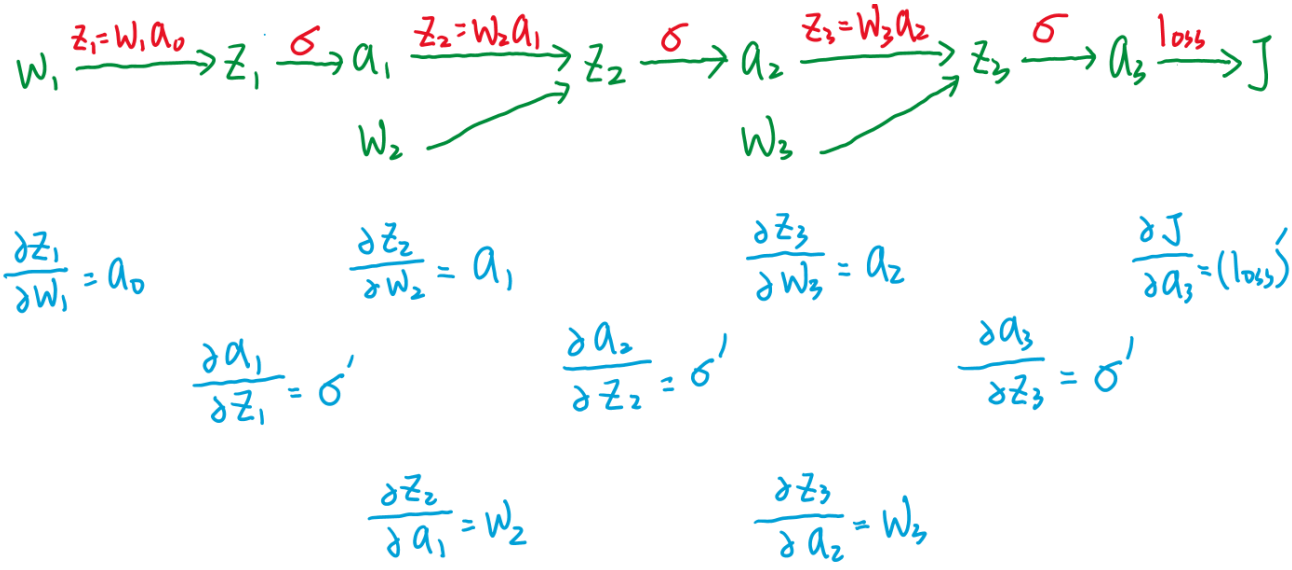




(2) Furthermore, we can utilize the chain rule to derive the backpropagation formula.



(3) Compute the partial derivatives for each variable.



(4) Since the input data consists of k data points, denoted as X_{2*k} (where X_{2*k} represents the calculated value of a0 as mentioned above) , the output data is represented as y_{1*k} (where y_{1*k} represents the calculated value of a3 as mentioned above)

$$\therefore\ a1_{n*k} = w1_{n*2} * a0_{2*k}$$

$$a2_{n*k} = w2_{n*n} * a1_{n*k}$$

$$a3_{1*k} = w3_{1*n} * a2_{n*k}$$

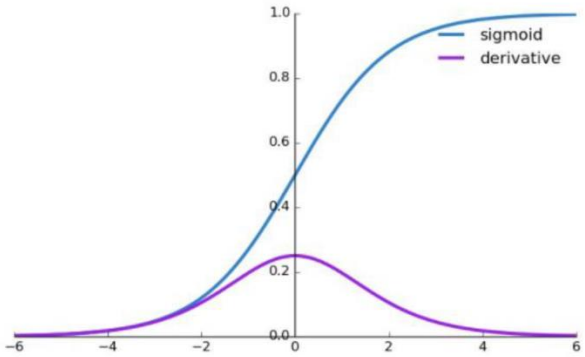
$\rightarrow w1_{n*2}$, $w2_{n*n}$, $w3_{1*n}$ (can get weight dimension)

2. Experiment setups:

A. Sigmoid functions

The sigmoid function is one of the activation functions used to address non-linear problems by employing a non-linear equation. It is suitable for tackling classification problems involving datasets such as XOR, which require non-linear classification boundaries.

Here are the graphs of the sigmoid function and its derivative:



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

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

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

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

推導過程：

$$\frac{\partial_\sigma}{\partial_x} = \frac{d}{dx}\left[\frac{1}{1+e^{-x}}\right] = \frac{d}{dx}(1+e^{-x})^{-1}$$

$$= -(1+e^{-x})^{-2} \times -e^{-x}$$

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

$$= \sigma(x)(1-\sigma(x))$$

B. Neural network

According to the fully connected NN diagram mentioned above, we have implemented two hidden layers, each consisting of 10 neurons. The operation of the NN is as follows: Initially, the weights w1, w2, and w3 are randomly initialized, and their dimensions can be determined using the calculations provided in Introduction B(4). Through the forwarding pass, we obtain the output, and then we perform backpropagation to compute the gradients and update the weights. The learning rate affects the magnitude of the weight updates. By iterating this process repeatedly, we can train the model.

The forwarding process of the NN can be implemented according to the diagram in Introduction B(1).

```
def forwarding_pass(func, x, w1, w2, w3):
    a0 = x
    z1 = w1 @ a0
    a1 = activation(func, z1)
    z2 = w2 @ a1
    a2 = activation(func, z2)
    z3 = w3 @ a2
    a3 = sigmoid(z3)
    pred_y = a3
    return pred_y, a0, a1, a2, a3
```

C. Backpropagation

The implementation of the backpropagation process follows the diagrams provided in Introduction B(2) and B(3), allowing us to achieve the desired outcome. In this case, the cross-entropy loss function is employed. When computing the cross entropy or its derivative, an epsilon term is incorporated to prevent division by zero or logarithm of zero errors.
(I use the Momentum Optimizer. In the extra part, we will discuss the details in depth.)

```
def backpropagation(func, pred_y, y, a3, w3, a2, w2, a1, w1, a0):
    eps = 0.0001
    dJ_da3 = -( y / (pred_y + eps) - (1 - y) / (1 - pred_y + eps) ) # derivative of cross-entropy
    dJ_dz3 = derivative_sigmoid(a3) * dJ_da3
    dJ_dw3 = dJ_dz3 @ a2.T

    dJ_da2 = w3.T @ dJ_dz3
    dJ_dz2 = derivative_activation(func, a2) * dJ_da2
    dJ_dw2 = dJ_dz2 @ a1.T

    dJ_da1 = w2.T @ dJ_dz2
    dJ_dz1 = derivative_activation(func, a1) * dJ_da1
    dJ_dw1 = dJ_dz1 @ a0.T

    return dJ_dw1, dJ_dw2, dJ_dw3
```

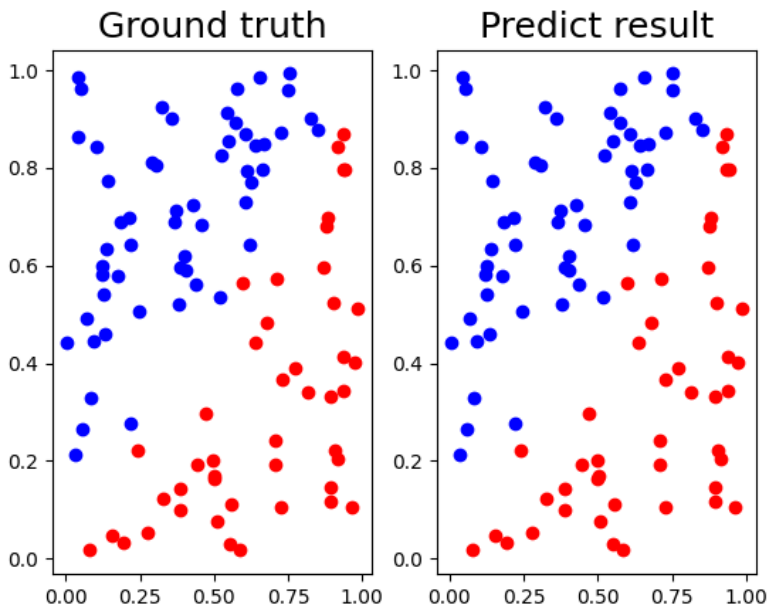
```python
def weight_update(optimizer, learning_rate, n, dJ_dw1, dJ_dw2, dJ_dw3, w1, w2, w3, m1, m2, m3):
    if optimizer == True:
        m1 = 0.9 * m1 - learning_rate * (dJ_dw1 / n)
        m2 = 0.9 * m2 - learning_rate * (dJ_dw2 / n)
        m3 = 0.9 * m3 - learning_rate * (dJ_dw3 / n)
        w1 = w1 + m1
        w2 = w2 + m2
        w3 = w3 + m3
    else:
        w1 = w1 - learning_rate * (dJ_dw1 / n)
        w2 = w2 - learning_rate * (dJ_dw2 / n)
        w3 = w3 - learning_rate * (dJ_dw3 / n)
    return w1, w2, w3, m1, m2, m3
```

3. Results of your testing

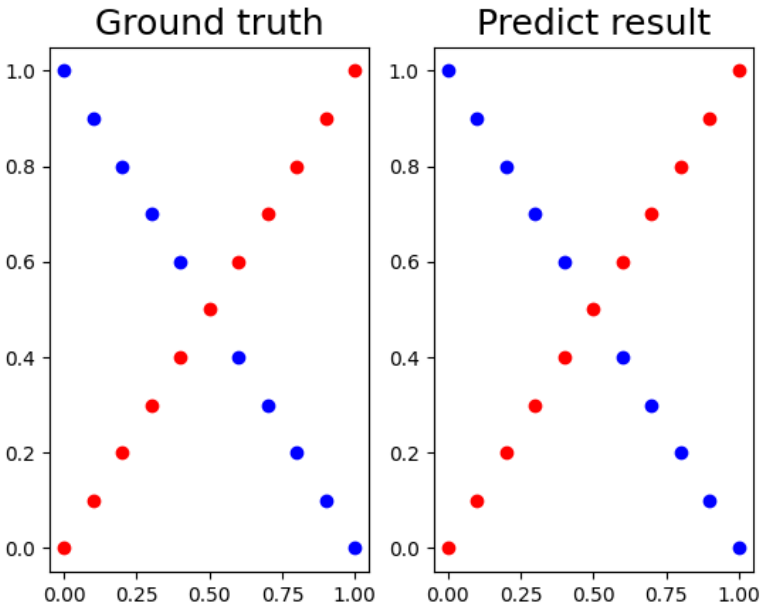
A. Screenshot and comparison figure

Both layers consist of 10 neurons each, learning rate = 0.1

Linear data: accuracy = 100%



XOR data: accuracy = 100%



B. Show the accuracy of your prediction

Linear training

```
Training ...
epoch 100 loss : 0.4137254512412997
epoch 200 loss : 0.10116533042046656
epoch 300 loss : 0.05668578606138413
epoch 400 loss : 0.04180481958449406
epoch 500 loss : 0.03434249936403518
epoch 600 loss : 0.029757891362925847
epoch 700 loss : 0.026583322044295716
epoch 800 loss : 0.02420773354145286
epoch 900 loss : 0.02233254513705826
epoch 1000 loss : 0.020794662929888387
epoch 1100 loss : 0.01949728334173696
epoch 1200 loss : 0.01837909643083457
epoch 1300 loss : 0.017399186861810162
epoch 1400 loss : 0.01652904905106762
epoch 1500 loss : 0.01574809722661112
epoch 1600 loss : 0.015041008263291172
epoch 1700 loss : 0.014396079319518196
epoch 1800 loss : 0.013804173914359832
epoch 1900 loss : 0.013258023091472398
epoch 2000 loss : 0.012751748457094094
epoch 2100 loss : 0.012280528193250505
epoch 2200 loss : 0.011840357770606916
epoch 2300 loss : 0.011427874953068922
epoch 2400 loss : 0.011040229434523296
epoch 2500 loss : 0.010674984093428456
epoch 2600 loss : 0.010330039062688604
epoch 2700 loss : 0.010003572542542059
epoch 2800 loss : 0.009693994091118126
epoch 2900 loss : 0.009399907346147008
epoch 3000 loss : 0.009120079968057282
epoch 3100 loss : 0.008853419178613898
epoch 3200 loss : 0.008598951682981972
epoch 3300 loss : 0.008355807060436789
epoch 3400 loss : 0.008123203925476876
epoch 3500 loss : 0.007900438320747136
epoch 3600 loss : 0.00768687392225885
epoch 3700 loss : 0.007481933727166847
epoch 3800 loss : 0.007285092962733586
epoch 3900 loss : 0.007095873007664075
epoch 4000 loss : 0.006913836157748019
epoch 4100 loss : 0.006738581099610001
epoch 4200 loss : 0.006569738981468081
epoch 4300 loss : 0.006406969989722337
epoch 4400 loss : 0.006249960356104998 5
epoch 4500 loss : 0.006098419732920165
epoch 4600 loss : 0.005952078884247736
epoch 4700 loss : 0.005810687649406242 5
epoch 4800 loss : 0.005674013141855108
epoch 4900 loss : 0.005541838152380272
epoch 5000 loss : 0.005413959730086311
```

Linear testing

```
Testing ...
[[2.06731907e-07 9.99996052e-01 3.69096878e-07 9.99999605e-01
  9.99999573e-01 7.45664570e-04 9.99999403e-01 1.04259632e-07
  9.99999402e-01 9.99999543e-01 9.99996468e-01 9.99999359e-01
  9.99999292e-01 1.80406256e-06 9.99999584e-01 9.99999596e-01
  9.99996806e-01 9.78849586e-01 1.35494978e-07 2.15284394e-07
  9.68142772e-08 1.93109022e-07 1.30775805e-07 6.05360005e-03
  1.60466881e-07 1.04294108e-07 9.99999523e-01 3.75337074e-06
  9.99999175e-01 9.99999603e-01 9.99999562e-01 1.02253202e-07
  4.28676333e-07 1.58632867e-07 9.99999427e-01 1.02924403e-07
  9.99993563e-01 9.99998781e-01 1.13077125e-07 9.99999550e-01
  9.99999446e-01 9.99999061e-01 3.86597537e-03 9.99999507e-01
  9.99999550e-01 9.99999390e-01 9.99999556e-01 1.06538877e-07
  9.31982839e-08 9.99998922e-01 1.37818544e-03 1.13254299e-07
  9.12490069e-08 9.99999533e-01 9.93221464e-08 9.99999418e-01
  9.99998538e-01 9.99999559e-01 5.15216538e-07 9.04828252e-06
  3.83456039e-07 9.99968571e-01 9.99999247e-01 9.99999448e-01
  2.58863126e-07 9.99997487e-01 9.98399894e-01 9.99998964e-01
  1.15016501e-07 9.42468656e-08 1.27517792e-06 9.57630252e-08
  1.17064852e-07 9.99999606e-01 4.41185022e-06 9.99989888e-01
  9.99999029e-01 9.99999558e-01 7.10464319e-07 9.99999498e-01
  8.51606797e-01 9.99999444e-01 1.15298151e-07 1.35617493e-07
  9.99993842e-01 4.02983794e-07 9.99998980e-01 9.99998294e-01
  9.99999521e-01 1.96776779e-07 9.99999517e-01 9.99999553e-01
  1.07298111e-07 9.99999485e-01 1.08377494e-07 1.39734400e-07
  9.99998140e-01 1.41893041e-07 9.99998592e-01 9.89086070e-01]]
accuracy : 100.0 %
```

XOR training

```
Training ...
epoch 100 loss : 0.6653445425154287
epoch 200 loss : 0.6474825628395305
epoch 300 loss : 0.6281300112123017
epoch 400 loss : 0.6029398011611152
epoch 500 loss : 0.5681956815108885
epoch 600 loss : 0.5209395546048553
epoch 700 loss : 0.46104717288420966
epoch 800 loss : 0.39493575060459146
epoch 900 loss : 0.3324379195583182
epoch 1000 loss : 0.27886286738038296
epoch 1100 loss : 0.23465158720650267
epoch 1200 loss : 0.19834049902455506
epoch 1300 loss : 0.1682558743717377
epoch 1400 loss : 0.14308163870213633
epoch 1500 loss : 0.12191652328043755
epoch 1600 loss : 0.1041440574947012
epoch 1700 loss : 0.08929008599880518
epoch 1800 loss : 0.07693802291550701
epoch 1900 loss : 0.06670093536606383
epoch 2000 loss : 0.05822359308809774
epoch 2100 loss : 0.051191530523873215
epoch 2200 loss : 0.045336393038808945
epoch 2300 loss : 0.04043565627618318
epoch 2400 loss : 0.036308402171802236
epoch 2500 loss : 0.032809304552965934
epoch 2600 loss : 0.02982235021125366
epoch 2700 loss : 0.027255107748218602
epoch 2800 loss : 0.025033858942033017
epoch 2900 loss : 0.023099630548126936
epoch 3000 loss : 0.02140503751349847
epoch 3100 loss : 0.019911806113092335
epoch 3200 loss : 0.018588844358191903
epoch 3300 loss : 0.01741074286230665
epoch 3400 loss : 0.016356609915877687
epoch 3500 loss : 0.01540916431315633
epoch 3600 loss : 0.014554026450904966
epoch 3700 loss : 0.013779161964225408
epoch 3800 loss : 0.013074442944398293
epoch 3900 loss : 0.012431300083673007
epoch 4000 loss : 0.011842445417442051
epoch 4100 loss : 0.011301650129760718
epoch 4200 loss : 0.010803565516795802
epoch 4300 loss : 0.010343577949278576
epoch 4400 loss : 0.009917690757634878
epoch 4500 loss : 0.009522427547179102
epoch 4600 loss : 0.009154752659427813
epoch 4700 loss : 0.008812005421782853
epoch 4800 loss : 0.008491845540700945
epoch 4900 loss : 0.008192207544661045
epoch 5000 loss : 0.007911262611439804
```

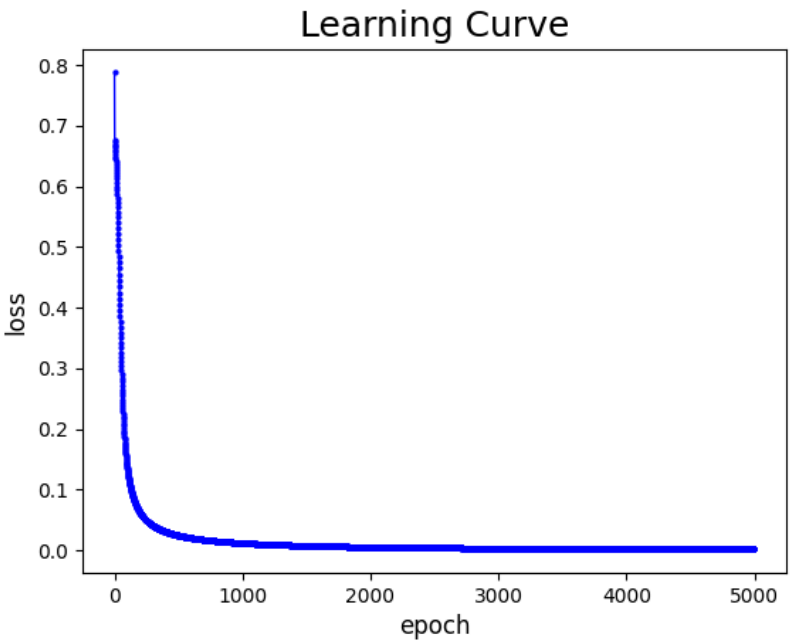
XOR testing

```
Testing ...
[[0.00705865 0.99905373 0.00726961 0.99897215 0.00746848 0.99877455
  0.00763297 0.99795665 0.00774196 0.96028033 0.00777871 0.00773351
  0.96916213 0.00760493 0.99745518 0.00739952 0.9978112  0.00712996
  0.9978055  0.00681246 0.99777819]]
accuracy : 100.0 %
```

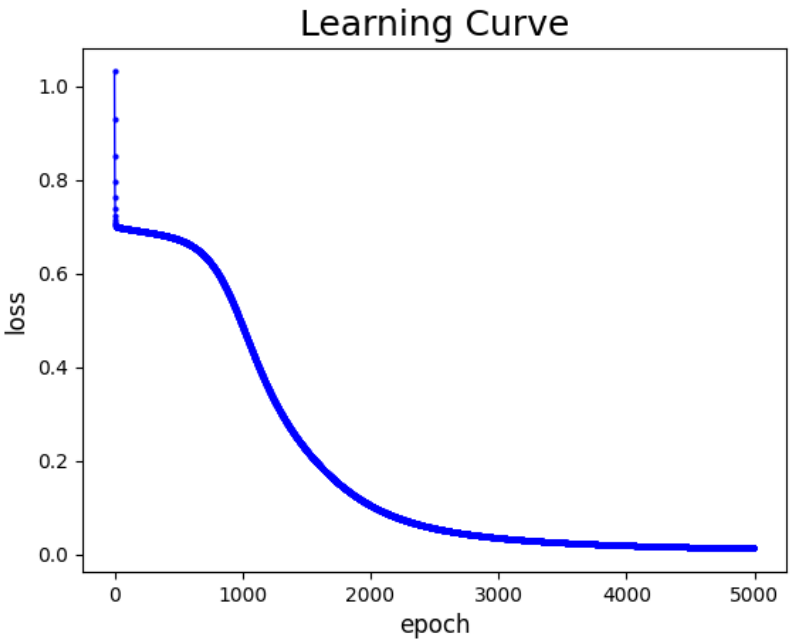
D. Learning curve (loss, epoch curve)

Both layers consist of 10 neurons each, learning rate = 0.1

Linear data: accuracy = 100%



XOR data: accuracy = 100%



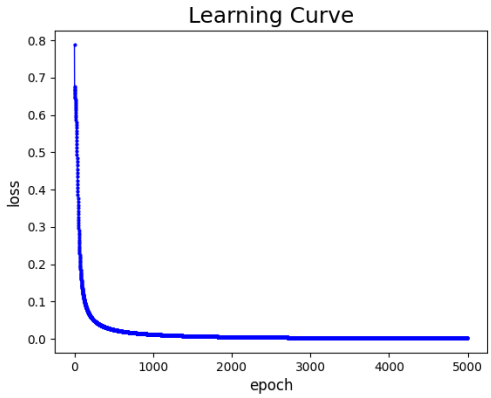
4. Discussion

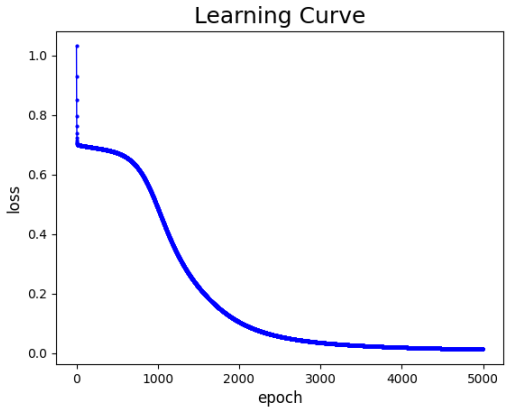
A. Try different learning rates

When the learning rate is larger, the weight updates have a larger magnitude, which allows the loss to converge more quickly. Below are the learning curves for learning rates of 0.1, 1, and 0.01.

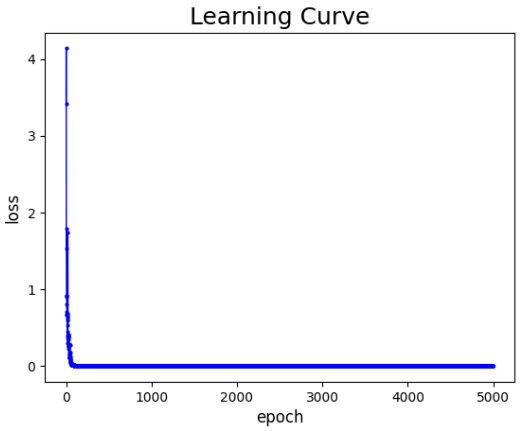
Linear data **XOR data**

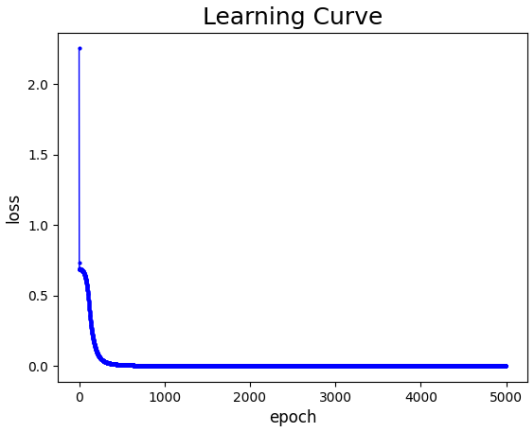
Origin: learning rate = 0.1



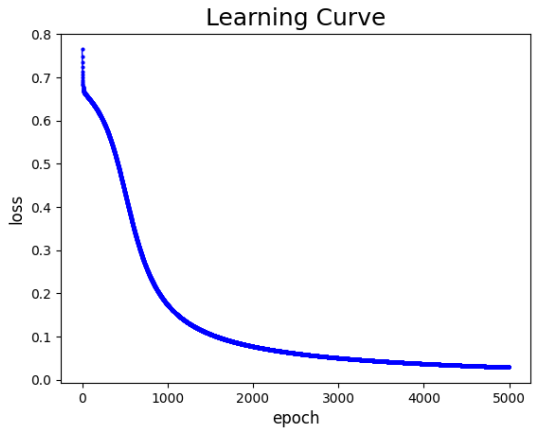


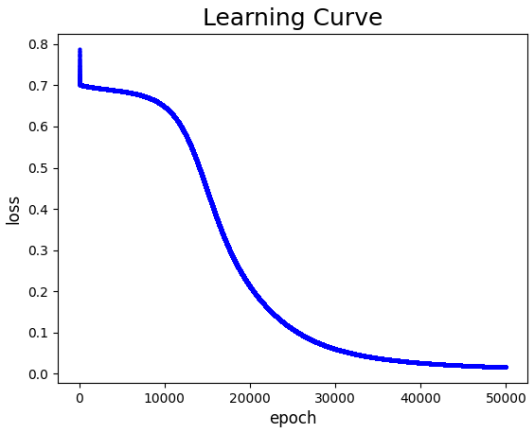
learning rate = 1





learning rate = 0.01





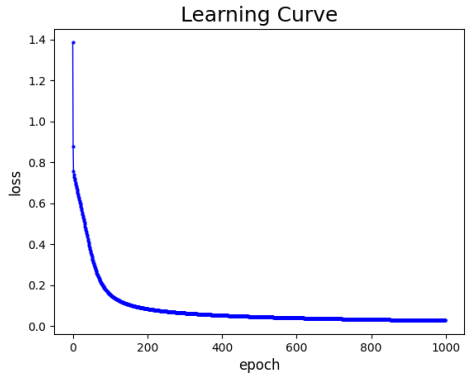
B. Try different numbers of hidden units

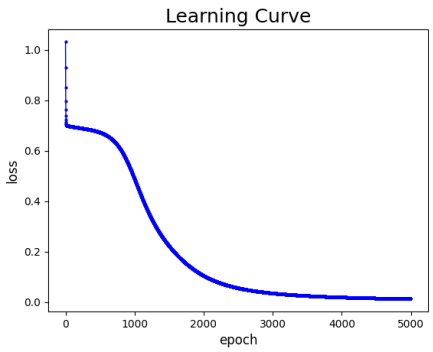
When the size of W (nxn) is changed, the total number of hidden units will also change. As n increases, the number of hidden units increases as well. It is observed that as n becomes larger, resulting in more hidden units, the loss decreases at a faster rate. Below are the learning curves for n values of 10, 15, and 5.

Linear data

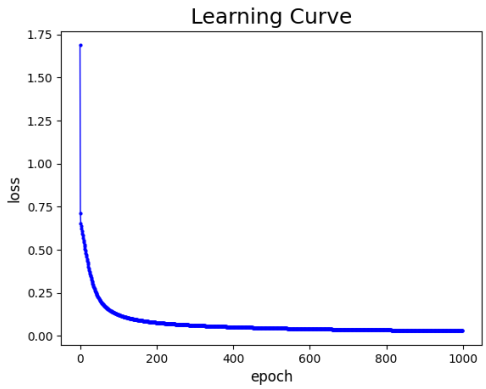
XOR data

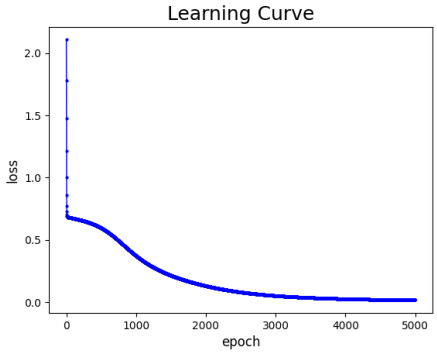
Origin: n = 10



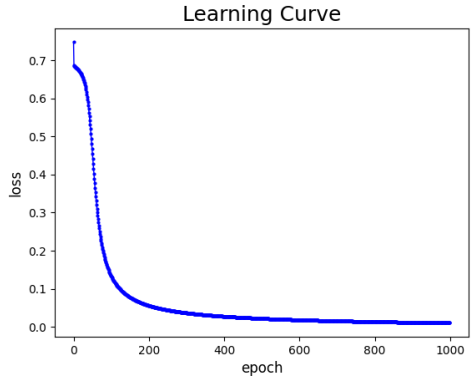


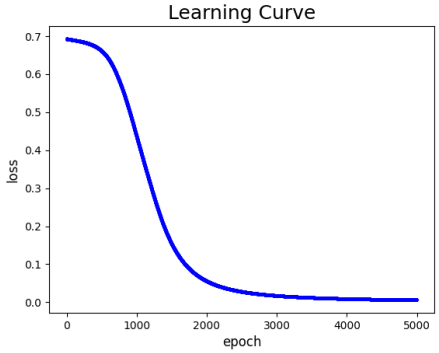
n = 15





n = 5



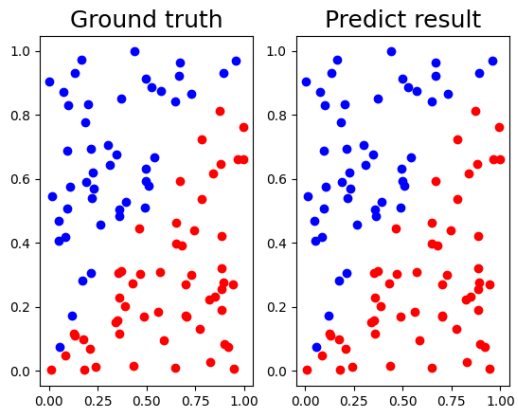


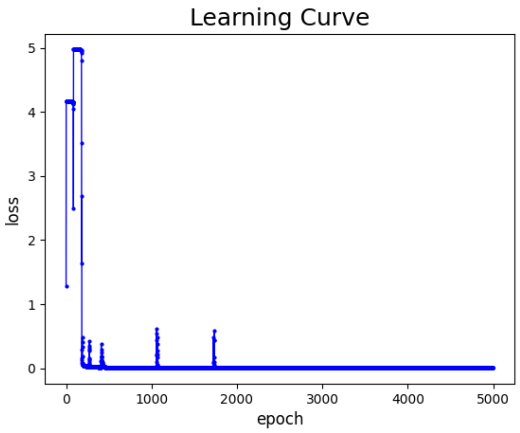
C. Try without activation functions

Without an activation function, the accuracy of XOR classification significantly decreases. This is because the absence of an activation function hinders the network's ability to effectively handle nonlinear data.

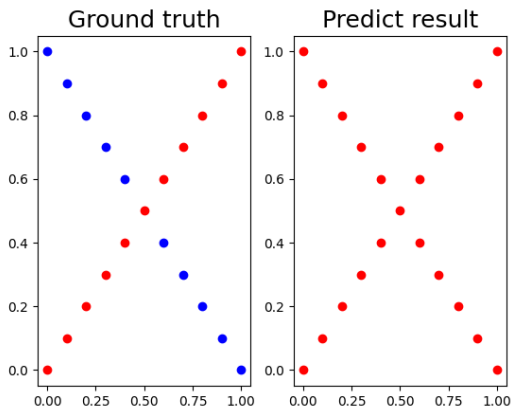
In the learning curve, it is evident that for linear data, using an activation function such as sigmoid leads to less pronounced loss oscillation compared to the case without an activation function. However, for XOR data, the loss appears to stagnate around 0.6920093402625231 and does not decrease further, indicating that the network struggles to properly classify the XOR data without an activation function.

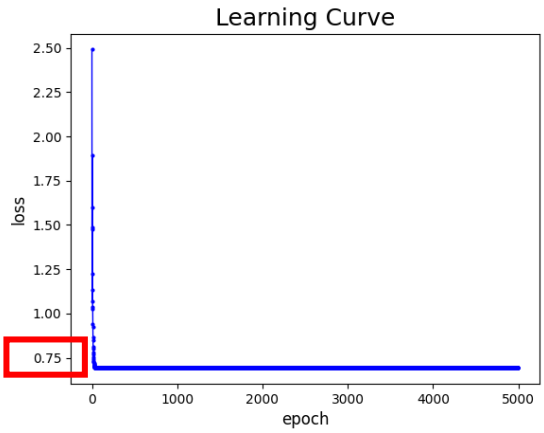
Linear data





XOR data





5. Extra

A. Implement different optimizers.

Origin: Stochastic Gradient Decent (SGD)

$$W \leftarrow W - \eta \frac{\partial L}{\partial W}$$

Different optimizer: Momentum Optimizer

This optimizer simulates the concept of physical momentum. It adjusts the learning rate differently in the dimensions of the same direction. The learning rate increases when the direction remains the same, and decreases when the direction changes. The following diagram illustrates the update formula for the momentum optimizer. In this formula, Vt represents the update velocity, which is influenced by the previous update. If the gradient in the previous step is in the same direction as the current gradient, the update velocity increases (gradient amplification). If the directions are different, the update velocity decreases (gradient attenuation). The parameter β can be thought of as air resistance or ground friction in physical momentum. It is typically set to 0.9 to control the influence of Vt-1 on Vt.

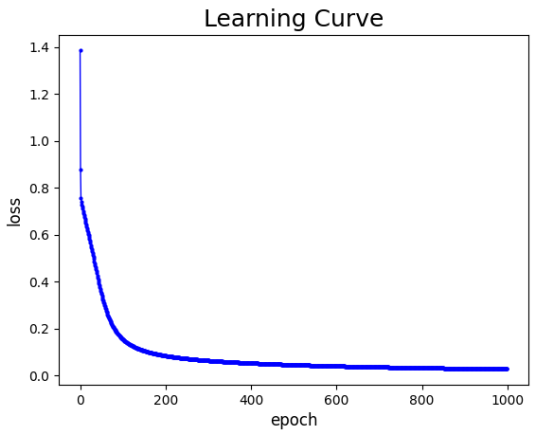
$$V_t \leftarrow \beta V_{t-1} - \eta \frac{\partial L}{\partial W}$$

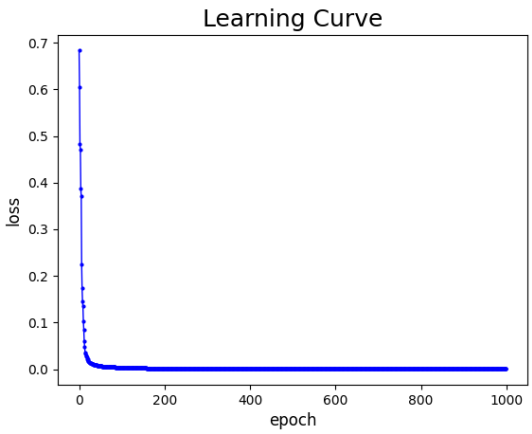
$$W \leftarrow W + V_t$$

When running the neural network with two layers of 10 neurons each and a learning rate of 0.1, it is observed that the use of the Momentum Optimizer results in faster convergence of the loss compared to using Stochastic Gradient Descent (SGD).

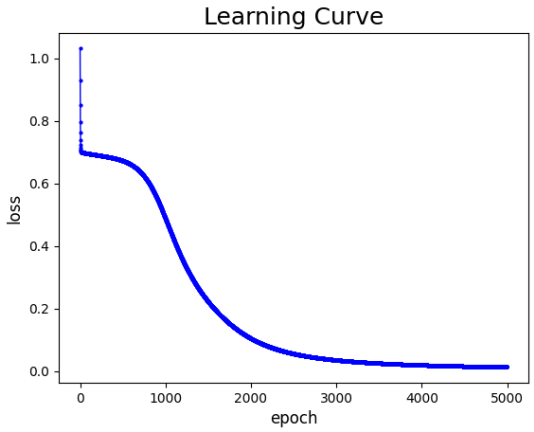
Stochastic Gradient Decent (SGD) **Momentum Optimizer**

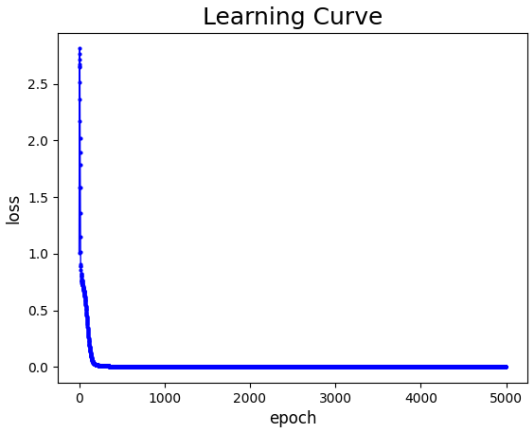
Linear data





XOR data





B. Implement different activation functions.

In the NN diagram provided in Introduction B(1), if we replace the activation functions of a1 and a2 with different functions, we need to ensure that the output layer a3 remains unchanged as sigmoid(z3). This is because the sigmoid activation function guarantees that the output values are bounded between 0 and 1, which is crucial for maintaining consistency during the backpropagation process.

ReLU, both layers consist of 10 neurons each, using learning rate = 0.1 and learning rate = 1

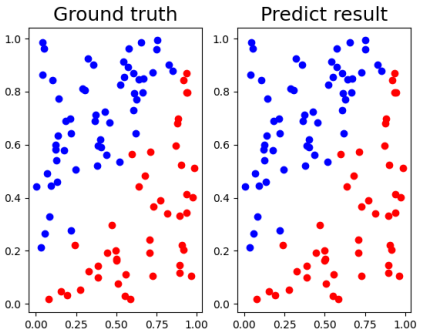
```python
def ReLU(x):
    x = np.maximum(0.0, x)
    return x

def derivative_ReLU(x):
    x[x <= 0] = 0
    x[x > 0] = 1
    return x

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

def derivative_tanh(x):
    return 1 - np.square(x)
```

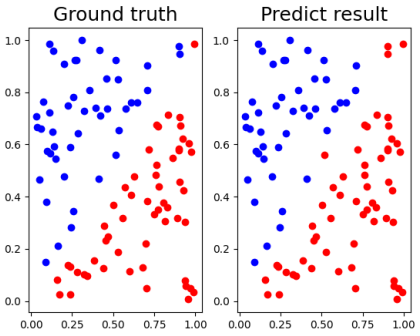
At a learning rate of 0.1, both activation functions perform reasonably well. Although ReLU exhibits some oscillation in the early stages of training, it eventually converges. However, at a learning rate of 1, the sigmoid activation function outperforms ReLU. ReLU exhibits significant oscillation, particularly when handling XOR data, and also yields high loss for linear data.

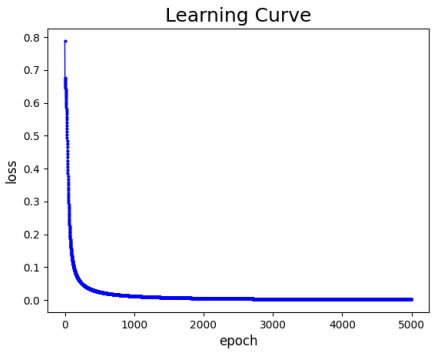
Therefore, it is crucial not to set the learning rate too high during training, as this can prevent the model from finding the optimum point.

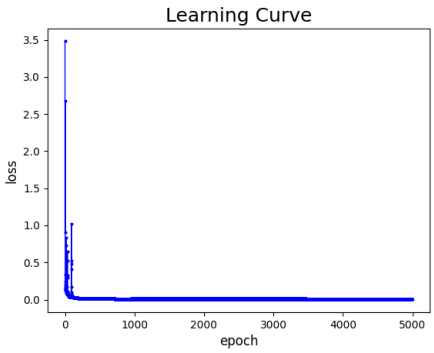
sigmoid **ReLU**

learning rate = 0.1 on linear data:

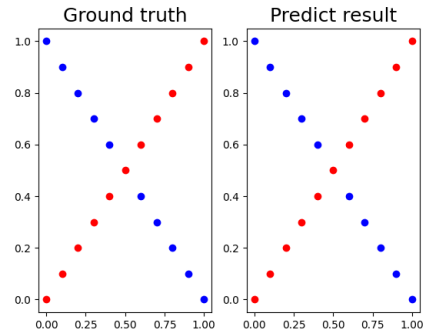


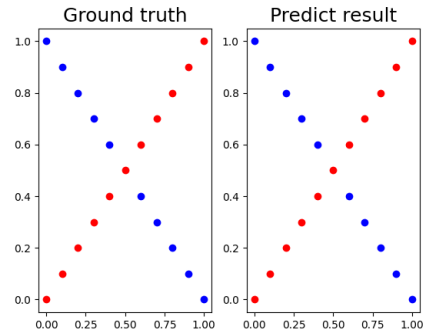


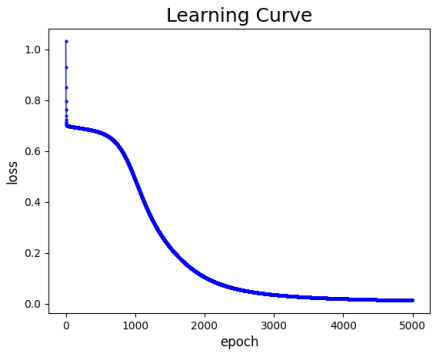


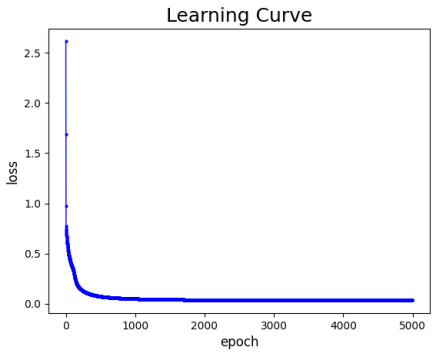


learning rate = 0.1 on XOR data:



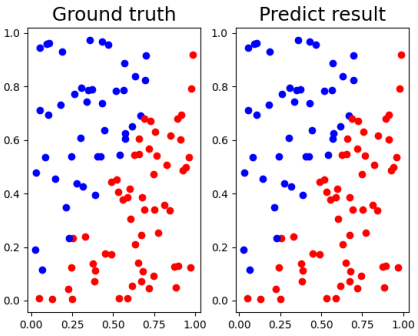


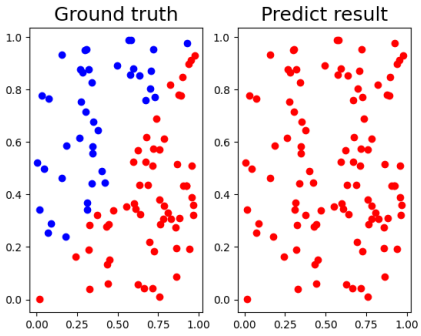


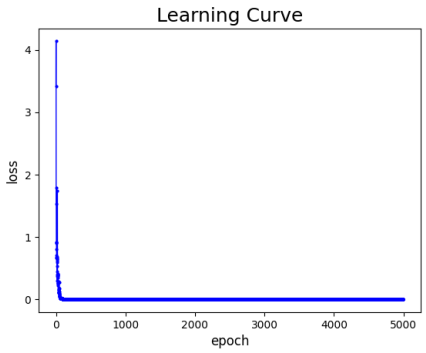


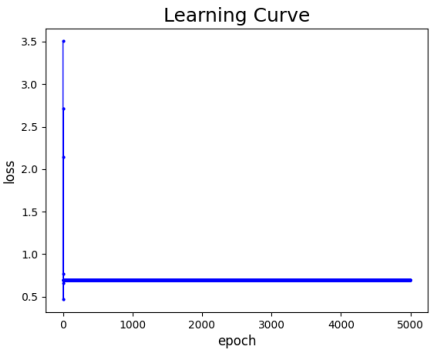
sigmoid **ReLU**

learning rate = 1 on linear data:









learning rate = 1 on XOR data: