Introduction:

當我們知道 gradient descent 可以調整 model 中 weight 對 output 的影響，就可以藉由調整參數來使模型有更好的輸出、表現。但因巨大的模型（例如：電腦視覺、語音辨識）可能有數百萬個參數需要調整，因此訓練的過程變得繁瑣且無效率。

所以現在最大的困難就是，如何把數百萬維的 weight 給計算出來 Backpropagation 的出現就是要處理這種問題，BP 本身就是 GD，但將每一個階段的 weight 對整個 model 輸出的影響，利用微積分的連鎖率清楚的將關係表現出來，且正因為有此關係，weight 所需的改變可以利用 backwardpass 輕鬆算出，且計算量如同 forwardpass。因此，有了 BP 就可以大幅提高訓練的效率。

在作業一中，分別給兩個不同的訓練資料，一為 linear 的資料、二為 XOR 的資料，如圖一。且 network architecture 綁定為一層 2dim input layer、兩層 hidden layer 一層 1 dim outputlayer，如圖二。

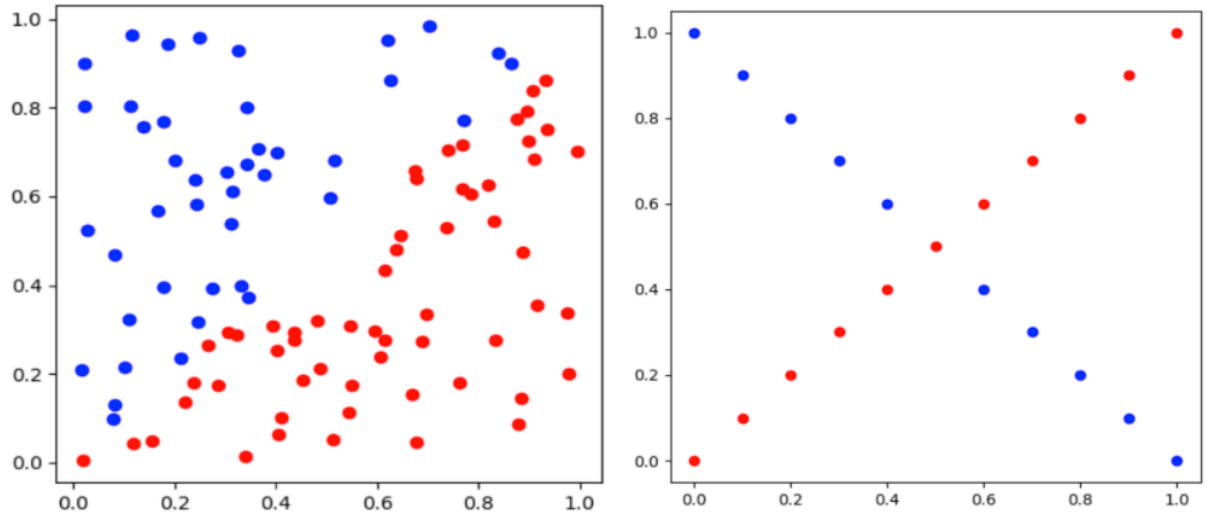


圖 1 。

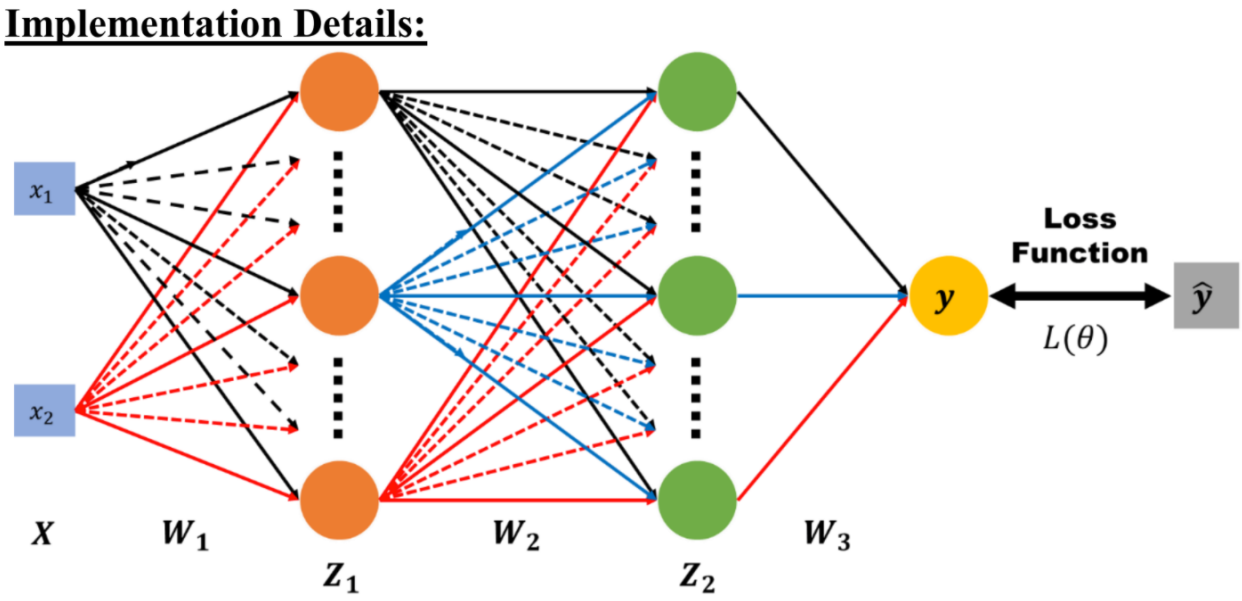


圖 2 。

Experiment setups:

Sigmoid functions:

Sigmoid functions 為其中之一的 activation function，其主要目的為，將 neural 的輸出轉換為非線性的，更直白的說，他能令 neural 對不同的數值做出不同的反應，如同決定是否發火一樣，就如同電子元件一樣超過多少電壓，才視為有輸出。而在我的實作中，我令 activation function 為 network 中的一層 layer，方便管理，如圖三。

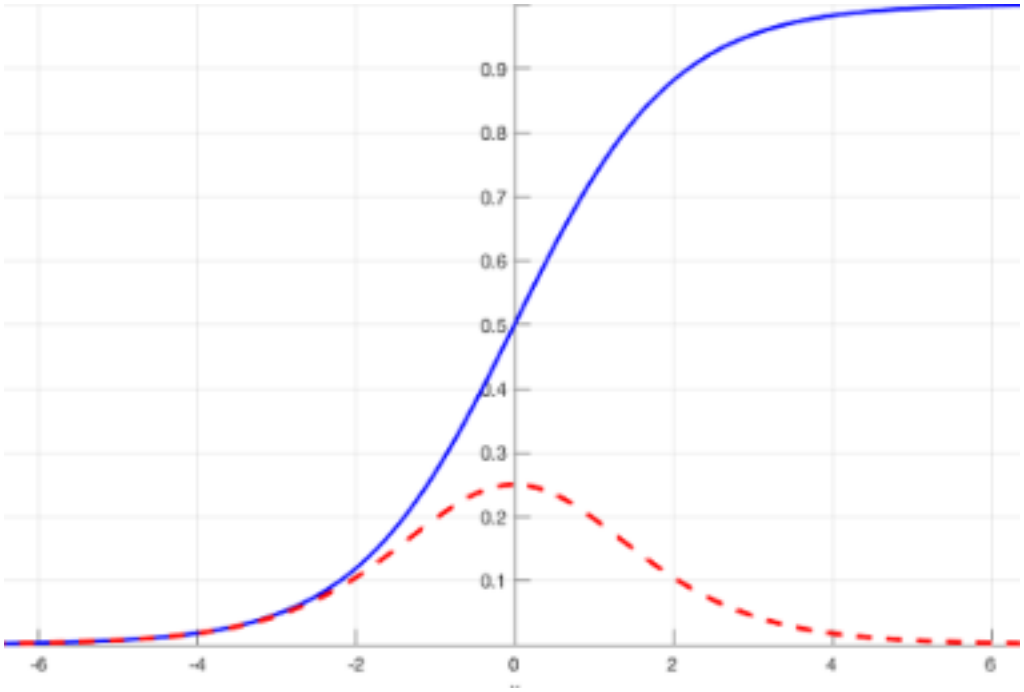
```
class Sigmoid():
  def __init__( self):
    pass

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

  def backward( self, forward_input, grad_input): #derivative_sigmoid
    # forward_input = forwardpass value
    # grad_input = backwardpass value
    # grad_output = Sigmoid'(forward_input) * grad_input
    # Sigmoid'(x) = sigmoid(x) * ( 1 - sigmoid(x))
    return derivative_sigmoid( forward_input) * grad_input
```

圖三。

而 sigmoid 及其導數如圖四。



圖四，藍色為 sigmoid、紅色為其導數

Network:

在整個 model 中最重要的部分就是層與層之間的 weight 參數，下一層的輸出即為上一層的輸出和參數 weight 的線性組合，再加上 activation function。

如前述，我將整個 Network 利用 OO method 架構而成，最上層 Class network 為主要骨幹，管理整個 network 到 forward 和 backward，如圖五。而 model 中的 weight 則用 Class Dense 來管理，Dense 為附屬於 network，主要負責參數傳送，forward、backward、還有 gradient weight 的 update，如圖六。

這麼做的好處就是，network 不是被寫死的，模型很好修改城想要的樣子，責任分工明確。另外，宣告一 list 作為保存每一層結果的輸出（包含 activation function），以便 forwardpass、backwardpass 使用。

```python
class Network():
  def __init__( self, input_layer_size, output_layer_size, hidden_layer1_dim, hidden_layer2_dim,activations):
    self.input_layer_size = input_layer_size
    self.output_layer_size = output_layer_size
    # constructure and build the network
      model = []
      model.append( Dense( input_layer_size, hidden_layer1_dim))
      model.append( Sigmoid())
      model.append( Dense( hidden_layer1_dim, hidden_layer2_dim))
      model.append( Sigmoid())
      model.append( Dense( hidden_layer2_dim, output_layer_size))

  def forward( self, input):
    activations = []
    x = input
    for layer in self.model:
      activations.append( layer.forward(x))
      x = activations[-1]
    activations = [input] + activations
    return activations

  def backward( self, forward_input, grad_input):
    #forward_input = activations , already computed by forwardpass
    for layer_index in range(len(self.model))[::-1]:
        grad_input = self.model[layer_index].backward( forward_input[layer_index], grad_input)
```

圖五。

```python
class Dense():
  # Dense mean the weight matrix between each layers
  def __init__( self, input_size, output_size, learning_rate = 0.1):
    self.input_size = input_size
    self.output_size = output_size
    self.learning_rate = learning_rate
    self.weight = np.random.randn( input_size, output_size)
    self.bias = np.zeros( output_size)

  def forward( self , input):
    # d = W(input) + bias
    return np.add(np.dot( input, self.weight),self.bias)

  def backward( self, forward_input, grad_input):
    # X >> weight(X) >> z >> activation(z) >> a

    # w' = w - learning_rate * (d C /d w)
    # b' = b - learning_rate * (d C /d b)

    # (d C /d w) = (d C /d z) * (d z /d w)
                          # (d z /d w) = forward_input
                # (d C /d z) = (d C /d a) * (d a /d z)
                                      # (d a /d z) = derivate_activation(z) = activation'(z)
                          # (d C /d a) = (d C /d z') * (d z' /d a) + (d C /d z") * (d z" / d a) + ..... can comput by dot product
                # (d C /d z) = activation'(z)*[w3*(d C /d z')+w4*(d C /d z")]
    # forward_input = forwardpass input
    # grad_input = dackwardpass input
    #print(grad_input,"\n")
    #print((self.weight).T)
    grad_output = np.dot( grad_input, (self.weight).T)
    grad_weight = np.dot( forward_input.T, grad_input)
    grad_bias = grad_input.mean(axis=0)
    self.weight = self.weight - self.learning_rate * grad_weight
    self.bias = self.bias - self.learning_rate * grad_bias
    return grad_output
```

圖六。

Backpropagation:

作為 Deep learning 中最基本也最重要的元素，BP 其實也沒有什麼高深的數學，單純就只是微積分中的連鎖率。如果想計算 weight 對其最後 cost function 的影響，則可利用連鎖率拆成如下：

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

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

$$\frac{\partial C}{\partial a} = \frac{\partial C}{\partial z'}\frac{\partial z'}{\partial a} + \frac{\partial C}{\partial z''}\frac{\partial z''}{\partial a} + \cdots\cdots$$

其中，

$$\frac{\partial z}{\partial w} = \text{forward input}$$

$$\frac{\partial a}{\partial z} = \text{derivate\_activtion}(z)$$

因此整個模型就可以一樣利用線性組合，反向傳播計算各 weight 對其輸出的影響，進而調整整個模型。大致上就是 BP 的精神，詳情非常推薦去看台大李宏毅的 BP 教學，Slides 跟講解影片都非常清楚，我大概反覆看了五次才搞明白，但看完就忘了。

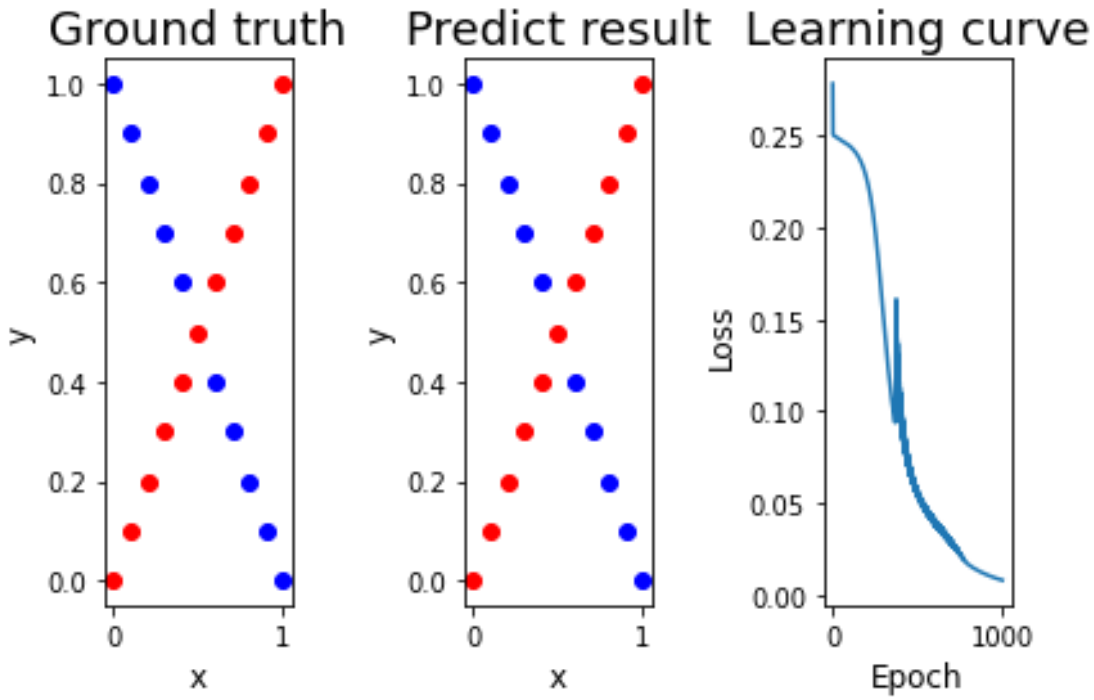
Results of your testing:

Screenshot and comparison fugure:

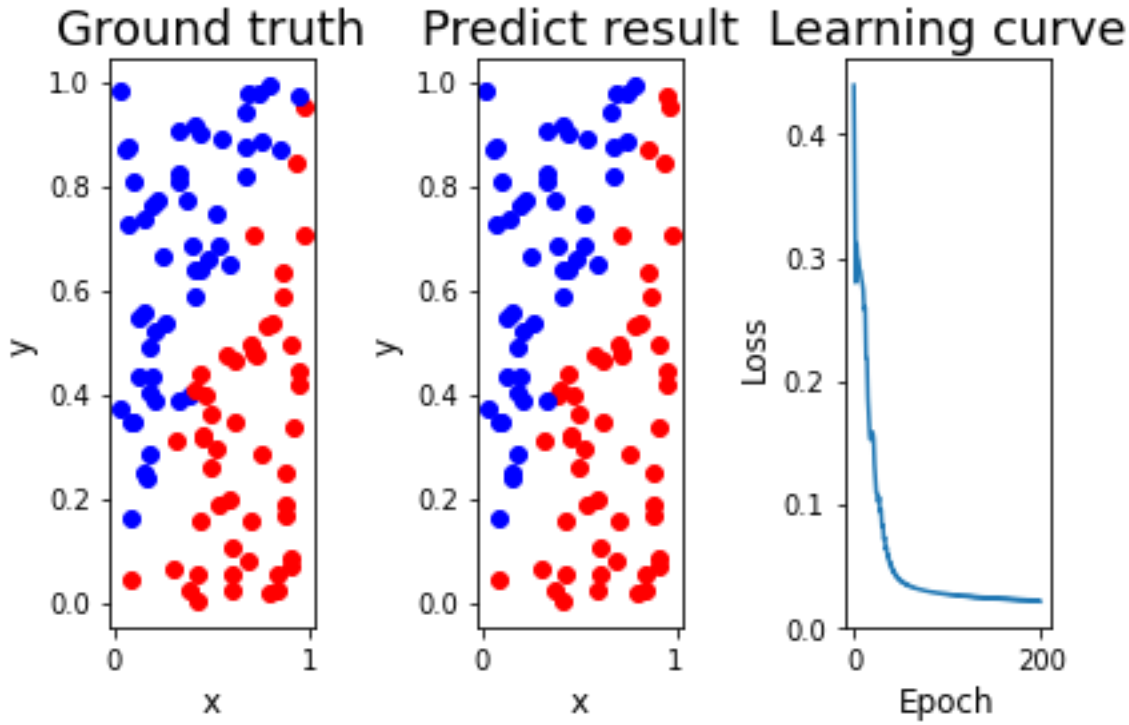
Show the accuracy of your prediction:

Learning curve (loss, epoch curve):

anything you want to present:



XOR problem , epochs = 1000 , hidden_layer1_dim = 10 , hidden_layer2_dim = 5 , act_fun = sigmoid , n = default lreaning_rate = 0.1 accuracy is 1.0



Linear problem , epochs = 200 , hidden_layer1_dim = 10 , hidden_layer2_dim = 5 , act_fun = sigmoid , n = 100 lreaning_rate = 0.1 accuracy is 0.97

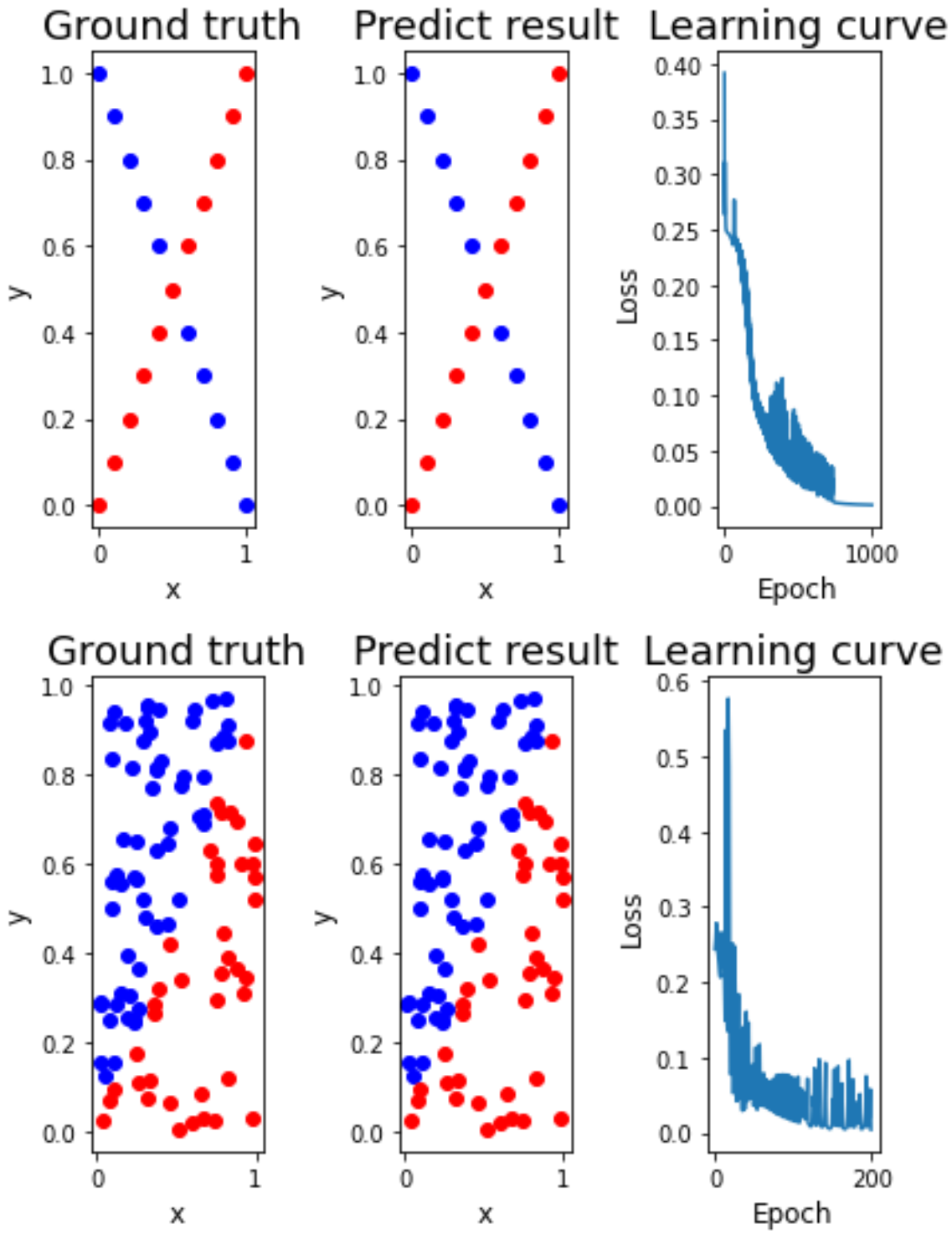
Discussion (30%)

Try different learning rates

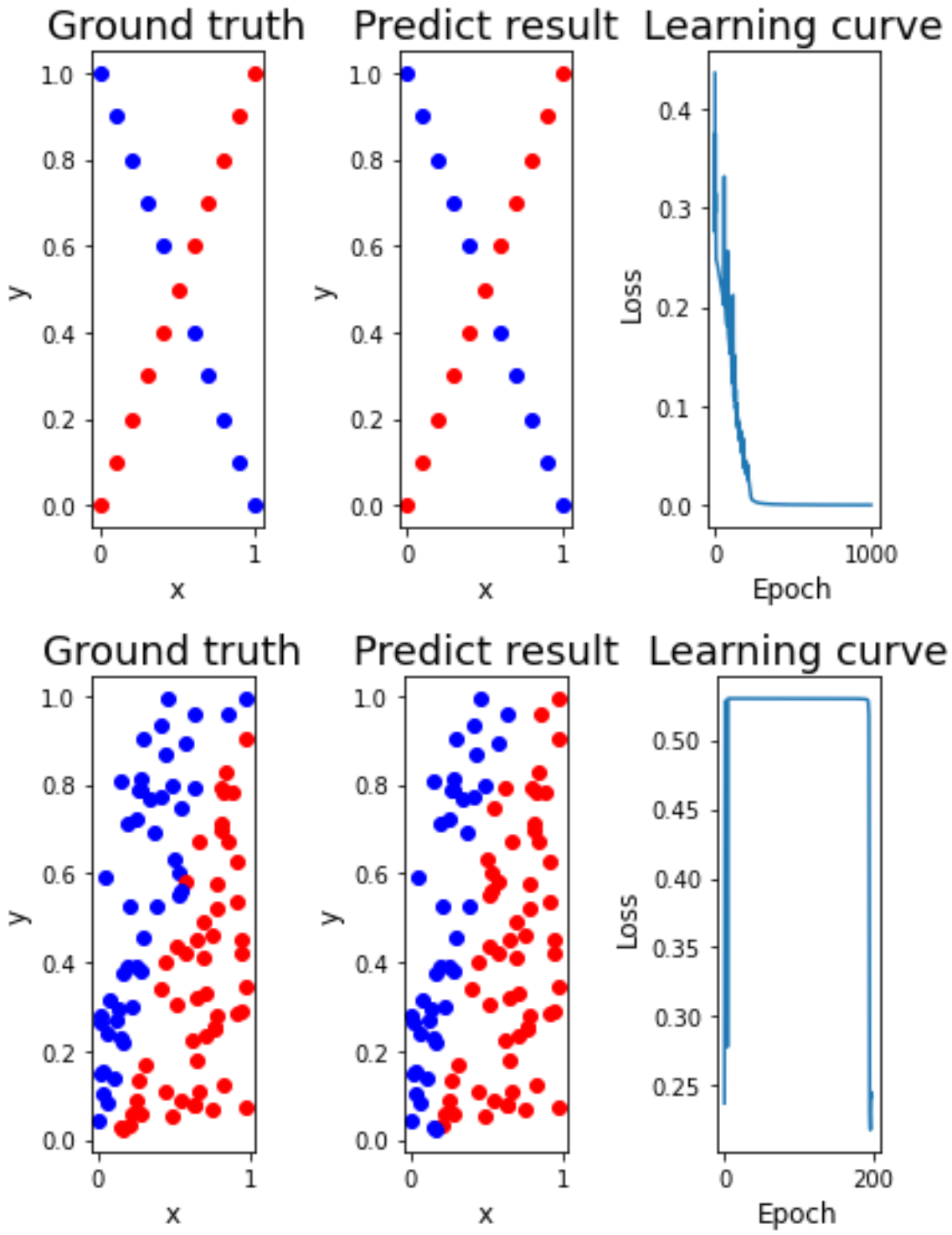
Try different numbers of hidden units

Trywithoutactivationfunctions

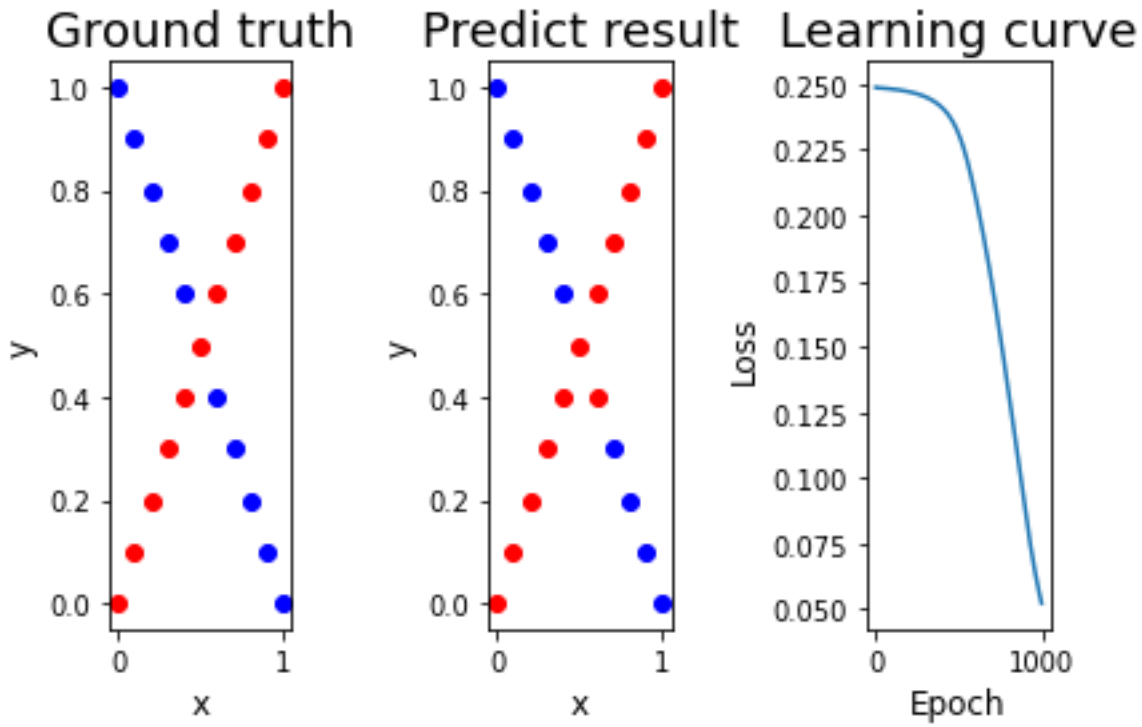
Anything you want to share

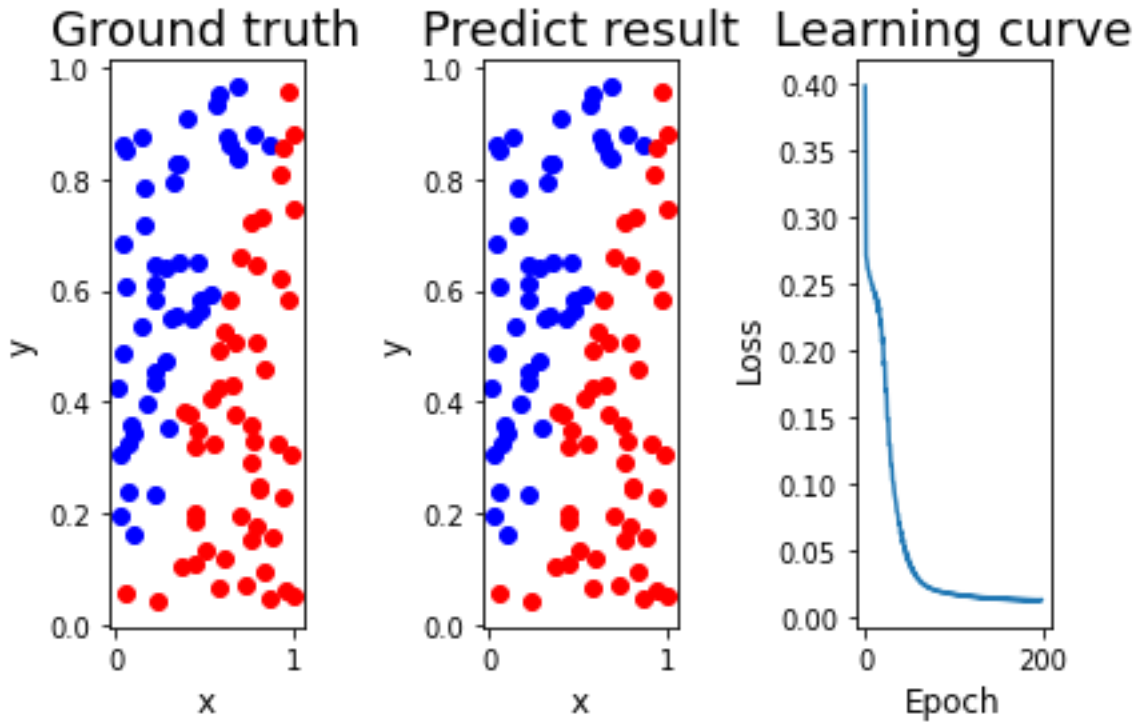


相同的配置但 learning_rate = 0.5 可以發現在這個 case 中，不但沒有更快降低 loss 而且訓練過程中震盪幅度大

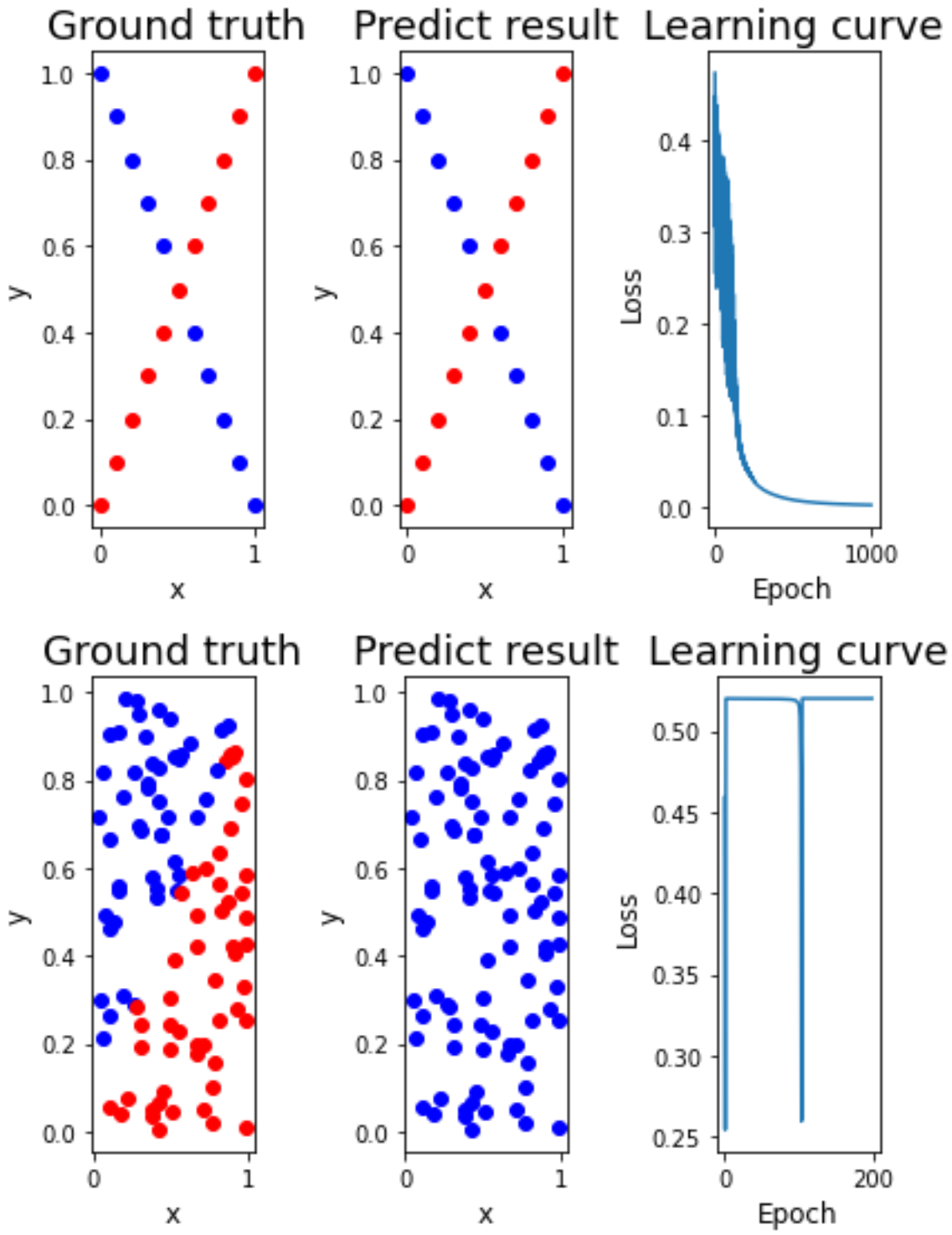


learning_rate = 1 在 learning case 中甚至可以看到很不合常理的圖。



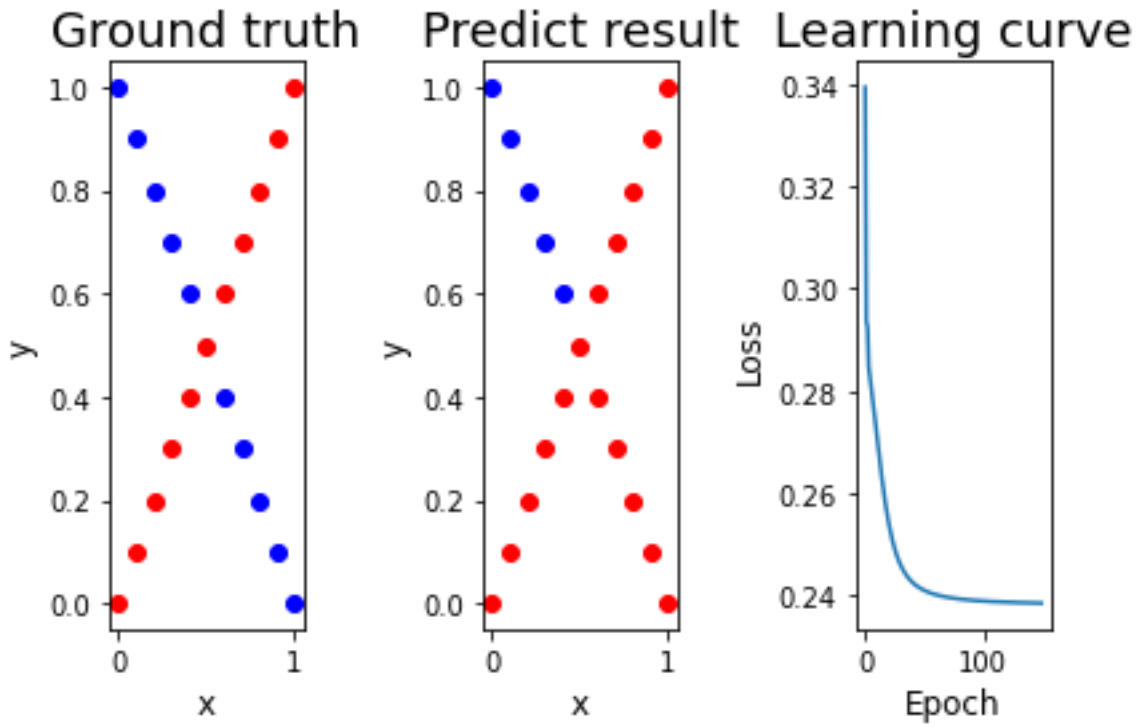


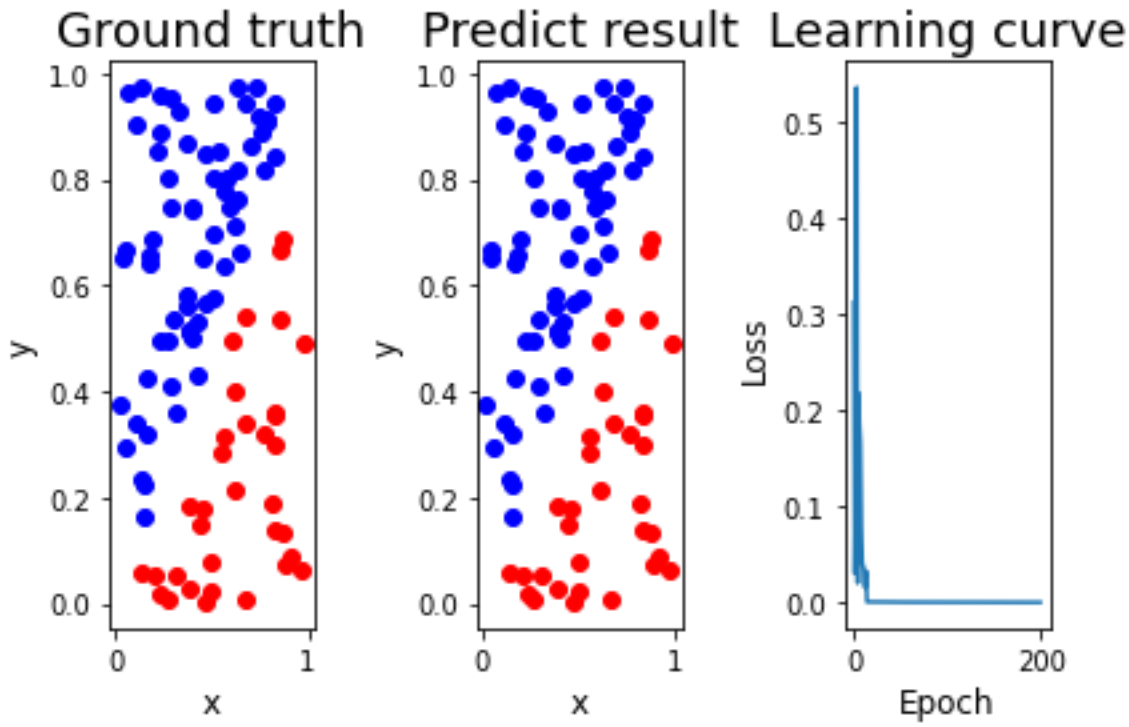
Hidden_layer1 , Hidden_layer2 = 5, 3



Hidden_layer1 , Hidden_layer2 = 50, 30

在 linear problem 中 accuracy is 0.48

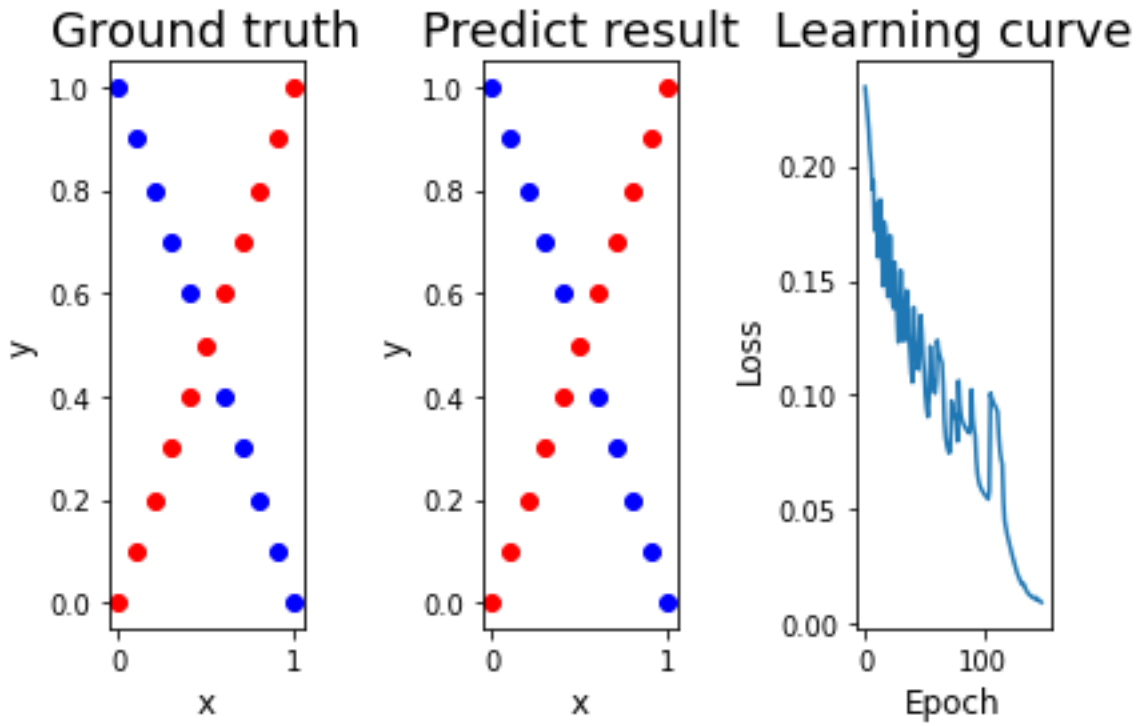




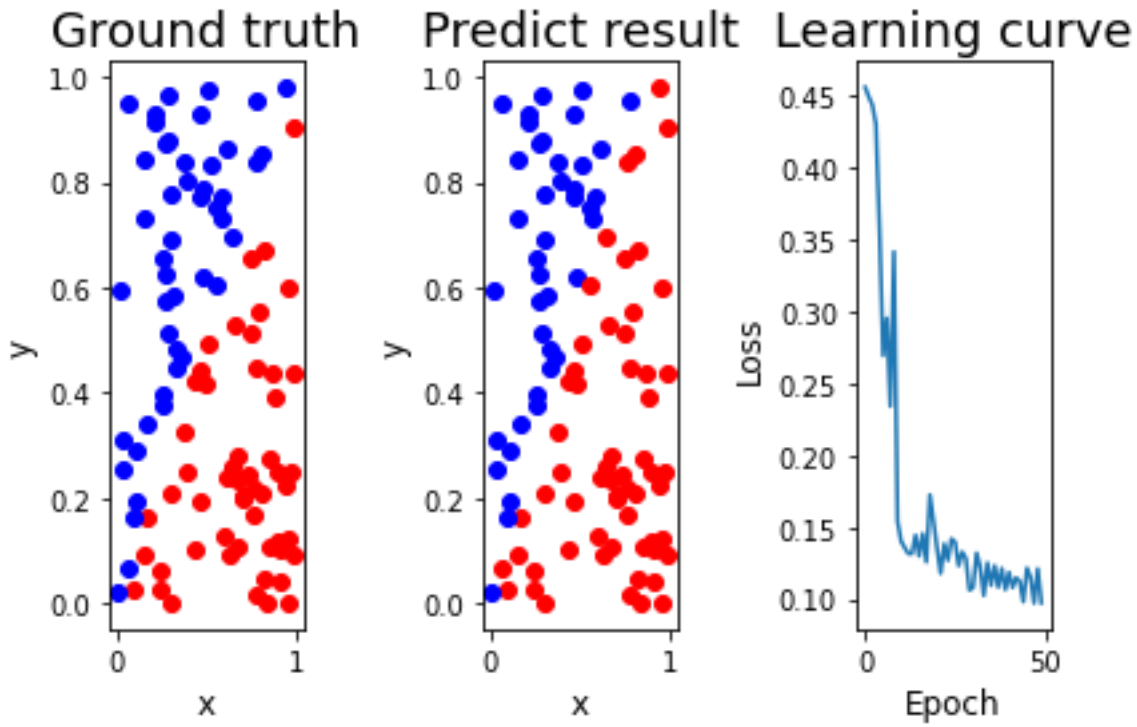
Hidden_layer without activation function case:

XOR problem accuracy is 0.7619047619047619

Linear problem accuracy is 1.0



在 Xor problem，如果將 activation function 改為 Relu 在相同架構下 sigmoid 需要近 1000 次才可以完成收斂但 Relu 只需要 150



在 linear problem 中，在相同架構下，如果將 activation function 改成 relu，訓練只要 50 次就可以有不錯的結果。

看似 Relu 在兩個問題上似乎都比 sigmoid 強

結論 Relu >>>>> Sigmoid