HW1 Report 311553030 林亮丞

1. Introduction

在這次作業中，目標是使用 numpy 實作一個有 2 層 hidden layer 的 network，每層 hidden layer 包含了一層 linear 和一層 sigmoid，並使用 backpropagation 去訓練。而訓練的資料有 2 種，每種 data 有 2 個 feature(x 和 y)，x 代表的是 2 維座標，y 代表的是 label，為 0 或 1。其中一種 data 為 100 個隨機點組成，我稱其為 data_100，另一種 data 為 X 形分布的，稱其為 data_X。

在這份報告中我會用截圖展示部分實作的程式，以及展示 2 種 data 各自 ground truth 畫出來的圖與各自去訓練的 model 預測出來的圖，並討論使用不同 lr、hidden_dim，以及是否使用 sigmoid 的差異。

2. Experiment setups:

A. Sigmoid functions

```python
def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

def derivative_sigmoid(self, x):
    return self.sigmoid(x) * (1 - self.sigmoid(x))
```

B. Neural network

使用一個有 2 層 hidden layer 的 network，hidden layer 為一層 hidden dim=10 的 linear 和一層 sigmoid，loss 使用 MSE/2，除 2 是為了方便 backward 時微分去掉*2，經過 backpropagation 去得到各層的 delta 來 update weights

```python
class MyNet:
    def __init__(self):
        self.epoch = []
        self.train_loss = []
        input_dim = 2
        hidden_dim = 10
        output_dim = 1

        # 隨機初始化權重和偏差
        self.W1 = np.random.randn(input_dim, hidden_dim)
        self.b1 = np.random.randn(hidden_dim)
        self.W2 = np.random.randn(hidden_dim, hidden_dim)
        self.b2 = np.random.randn(hidden_dim)
        self.W3 = np.random.randn(hidden_dim, output_dim)
        self.b3 = np.random.randn(output_dim)
```

```python
def loss(self, y_true, y_pred):
    loss = np.mean((y_true - y_pred)**2) / 2
    return loss
```

C. Backpropagation

過程為先 forward 得到各層的結果，再透過 backward 計算各層的 delta，再由這些

delta 與前面 forward 得到的各層結果去計算出每層 weight 要更新的 gradient。

```
# forward
# 第一層隱藏層
z1 = np.dot(X, self.W1) + self.b1
a1 = self.sigmoid(z1)
# 第二層隱藏層
z2 = np.dot(a1, self.W2) + self.b2
a2 = self.sigmoid(z2)
# 輸出層
z3 = np.dot(a2, self.W3) + self.b3
y_pred = self.sigmoid(z3)
train_loss = self.loss(y, y_pred)
```

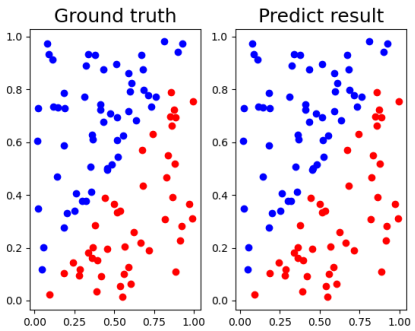
```
# backward
out_layer_error = y_pred-y
out_layer_delta = out_layer_error * self.derivative_sigmoid(z3)
hidden_layer2_error = np.dot(out_layer_delta, self.W3.T)
hidden_layer2_delta = hidden_layer2_error * self.derivative_sigmoid(z2)
hidden_layer1_error = np.dot(hidden_layer2_delta, self.W2.T)
hidden_layer1_delta = hidden_layer1_error * self.derivative_sigmoid(z1)
```

```
# update weights
N = X.shape[0]
self.W3 -= lr * (np.dot(a2.T, out_layer_delta) / N)
self.b3 -= lr * (np.sum(out_layer_delta, axis=0) / N)
self.W2 -= lr * (np.dot(a1.T, hidden_layer2_delta) / N)
self.b2 -= lr * (np.sum(hidden_layer2_delta, axis=0) / N)
self.W1 -= lr * (np.dot(X.T, hidden_layer1_delta) / N)
self.b1 -= lr * (np.sum(hidden_layer1_delta, axis=0) / N)
```

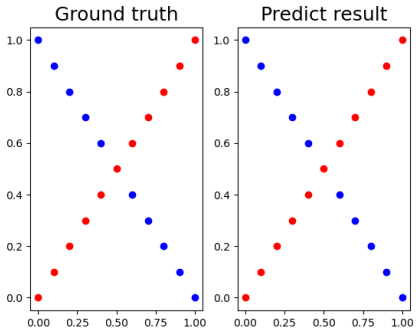
3. Results of your testing:

A. Screenshot and comparison figure

Data_100



Data_X



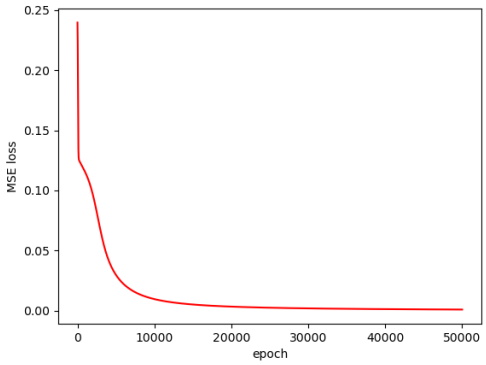
B. Show the accuracy of your prediction

Data_100: `accuracy: 100/100`

Data_X: `accuracy: 21/21`

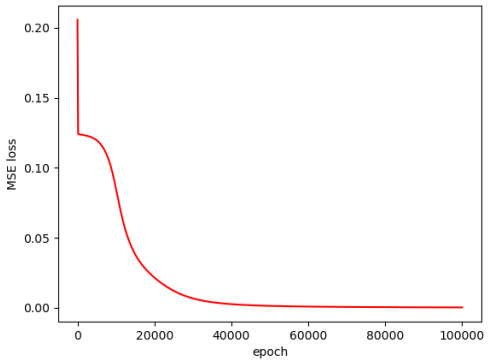
C. Learning curve (loss, epoch curve)

Data_100: lr: 0.1, epochs: 50000




```
epoch 5000, train_loss: 0.029797329203481594
epoch 10000, train_loss: 0.009651602404142057
epoch 15000, train_loss: 0.0051656513067742825
epoch 20000, train_loss: 0.003404366981752707
epoch 25000, train_loss: 0.002492604045534351
epoch 30000, train_loss: 0.0019417826321992283
epoch 35000, train_loss: 0.0015755817533634475
epoch 40000, train_loss: 0.0013158888250258778
epoch 45000, train_loss: 0.0011229831754690895
epoch 50000, train_loss: 0.0009745900760822118
```

Data_X: lr: 0.1, epochs: 100000




```
epoch 5000, train_loss: 0.1196865986809471
epoch 10000, train_loss: 0.08521368472846182
epoch 15000, train_loss: 0.038203671685889756
epoch 20000, train_loss: 0.02161258009597188
epoch 25000, train_loss: 0.012198596127480522
epoch 30000, train_loss: 0.006770540559770289
epoch 35000, train_loss: 0.004035097919940663
epoch 40000, train_loss: 0.0026429984193216912
epoch 45000, train_loss: 0.0018741434330376706
epoch 50000, train_loss: 0.0014111229636495032
epoch 55000, train_loss: 0.0011111825145001153
epoch 60000, train_loss: 0.0009052097498455649
epoch 65000, train_loss: 0.0007570450597395534
epoch 70000, train_loss: 0.0006464165679634977
epoch 75000, train_loss: 0.0005612709485190255
epoch 80000, train_loss: 0.0004940772067124177
epoch 85000, train_loss: 0.00043992864519117194
epoch 90000, train_loss: 0.00039551245265261406
epoch 95000, train_loss: 0.00035852258949808967
epoch 100000, train_loss: 0.0003273108898703084
```

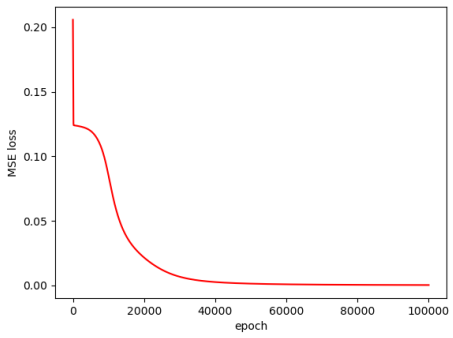
4. Discussion:

以下嘗試都使用 Data_X 訓練

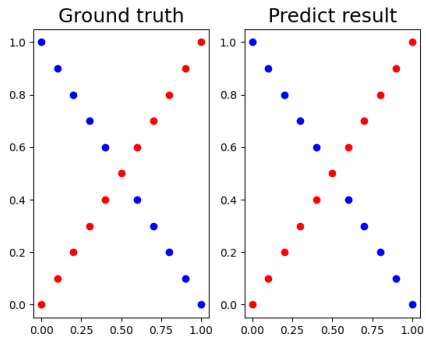
A. Try different learning rates:

從以下結果可以明顯看出在使用同樣的 epoch 下，當使用較小的 lr 時更新參數的速度會較緩慢，會需要更多 epoch 才能收斂，而較大的 lr 會更新較快，但也可能會導致無法收斂。

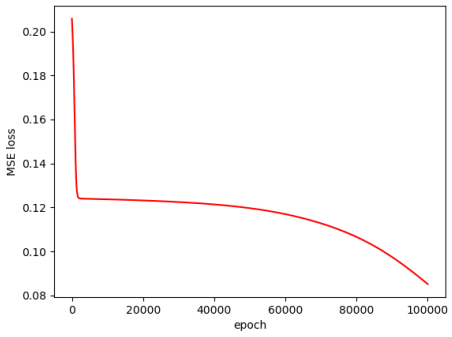
lr: 0.1



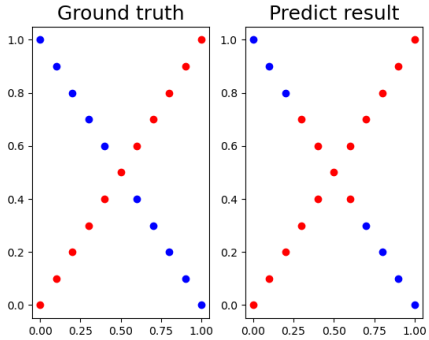

```
epoch 5000, train_loss: 0.1196865986809471
epoch 10000, train_loss: 0.08521368472846182
epoch 15000, train_loss: 0.038203671685889756
epoch 20000, train_loss: 0.02161258009597188
epoch 25000, train_loss: 0.012198596127480522
epoch 30000, train_loss: 0.006770540559770289
epoch 35000, train_loss: 0.004035097919940663
epoch 40000, train_loss: 0.0026429984193216912
epoch 45000, train_loss: 0.0018741434330376706
epoch 50000, train_loss: 0.0014111229636495032
epoch 55000, train_loss: 0.0011111825145001153
epoch 60000, train_loss: 0.0009052097498455649
epoch 65000, train_loss: 0.0007570450597395534
epoch 70000, train_loss: 0.0006464165679634977
epoch 75000, train_loss: 0.0005612709485190255
epoch 80000, train_loss: 0.0004940772067124177
epoch 85000, train_loss: 0.00043992864519117194
epoch 90000, train_loss: 0.0003955124526526140 6
epoch 95000, train_loss: 0.0003585225894980896 7
epoch 100000, train_loss: 0.0003273108898703084
```



lr: 0.01




```
epoch 5000, train_loss: 0.12392788770388667
epoch 10000, train_loss: 0.12371382130061753
epoch 15000, train_loss: 0.1234711912595314
epoch 20000, train_loss: 0.12319054893186972
epoch 25000, train_loss: 0.1228598460408952
epoch 30000, train_loss: 0.12246329931037557
epoch 35000, train_loss: 0.12197990933860828
epoch 40000, train_loss: 0.12138165261438875
epoch 45000, train_loss: 0.1206315975229848
epoch 50000, train_loss: 0.11968275668878668
epoch 55000, train_loss: 0.11847948541942561
epoch 60000, train_loss: 0.11696360845005647
epoch 65000, train_loss: 0.11508272753319894
epoch 70000, train_loss: 0.11278615568963482
epoch 75000, train_loss: 0.10999814475597665
epoch 80000, train_loss: 0.10659837586638336
epoch 85000, train_loss: 0.10244835534074138
epoch 90000, train_loss: 0.0974532930485151
epoch 95000, train_loss: 0.09163329666374327
epoch 100000, train_loss: 0.08518134089978428
```

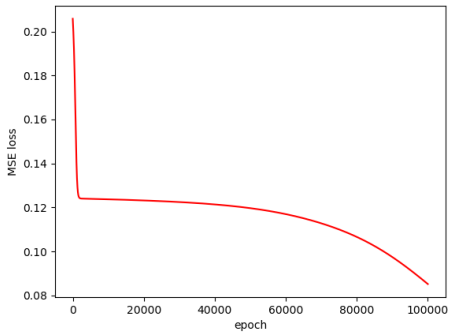


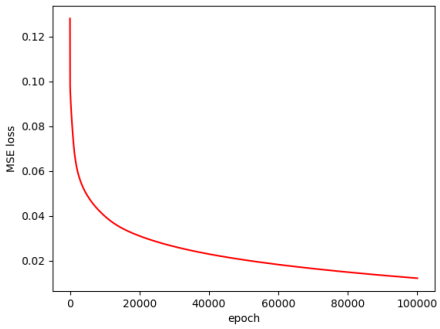
B. Try different numbers of hidden units

從以下結果可以看出在 hidden dim 為 10 的情況下，lr: 0.01 還沒到收斂的地步，但將 hidden dim 設為 100 時則接近收斂，且訓練結果也變完全正確，雖然使用較大 hidden dim 可以使精確度上升，但缺點是訓練時間會變長，因為一個 epoch 需要花更多時間。

lr: 0.01, hidden dim: 10

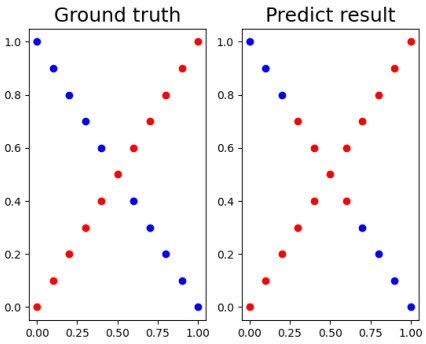
lr: 0.01, hidden dim: 100

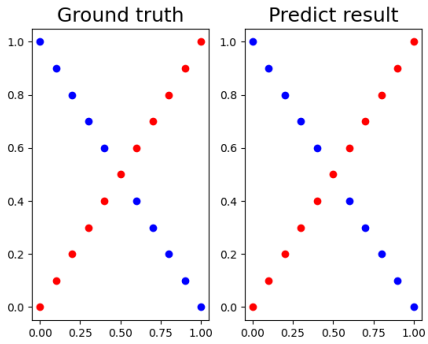





```
epoch 5000, train_loss: 0.12392788770388667
epoch 10000, train_loss: 0.12371382130061753
epoch 15000, train_loss: 0.1234711912595314
epoch 20000, train_loss: 0.12319054893186972
epoch 25000, train_loss: 0.1228598460408952
epoch 30000, train_loss: 0.12246329931037557
epoch 35000, train_loss: 0.12197990933860828
epoch 40000, train_loss: 0.12138165261438875
epoch 45000, train_loss: 0.1206315975229848
epoch 50000, train_loss: 0.11968275668878668
epoch 55000, train_loss: 0.11847948541942561
epoch 60000, train_loss: 0.11696360845005647
epoch 65000, train_loss: 0.11508272753319894
epoch 70000, train_loss: 0.11278615568963482
epoch 75000, train_loss: 0.10999814475597665
epoch 80000, train_loss: 0.10659837586638336
epoch 85000, train_loss: 0.10244835534074138
epoch 90000, train_loss: 0.0974532930485151
epoch 95000, train_loss: 0.09163329666374327
epoch 100000, train_loss: 0.08518134089978428
```

```
epoch 5000, train_loss: 0.04908565322758605
epoch 10000, train_loss: 0.03994592057224979
epoch 15000, train_loss: 0.03453864766649514
epoch 20000, train_loss: 0.03110347670469567
epoch 25000, train_loss: 0.028490165876346694
epoch 30000, train_loss: 0.026361618303696314
epoch 35000, train_loss: 0.024567299251023827
epoch 40000, train_loss: 0.02301768308588579
epoch 45000, train_loss: 0.021654297483129987
epoch 50000, train_loss: 0.020436510518649892
epoch 55000, train_loss: 0.019334813669565083
epoch 60000, train_loss: 0.018327126202651898
epoch 65000, train_loss: 0.0173966137690491
epoch 70000, train_loss: 0.01653030886836746
epoch 75000, train_loss: 0.01571817965319562
epoch 80000, train_loss: 0.014952465249694439
epoch 85000, train_loss: 0.014227180283760995
epoch 90000, train_loss: 0.013537733629373536
epoch 95000, train_loss: 0.012880628213752824
epoch 100000, train_loss: 0.012253220599309823
```





C. Try without activation functions

如果 network 不使用 activation functions 的話，會只能計算到線性的關係，且容易會使計算 loss 過程產生 overflow，以下為 lr 為 0.1, hidden_dim=10 情況下不使用任何 activation function 的結果。

```
hw1.py:142: RuntimeWarning: overflow encountered in square
  loss = np.mean((y_true - y_pred)**2) / 2
100%|
epoch 5000, train_loss: nan
epoch 10000, train_loss: nan
epoch 15000, train_loss: nan
epoch 20000, train_loss: nan
epoch 25000, train_loss: nan
epoch 30000, train_loss: nan
epoch 35000, train_loss: nan
epoch 40000, train_loss: nan
epoch 45000, train_loss: nan
epoch 50000, train_loss: nan
epoch 55000, train_loss: nan
epoch 60000, train_loss: nan
epoch 65000, train_loss: nan
epoch 70000, train_loss: nan
epoch 75000, train_loss: nan
epoch 80000, train_loss: nan
epoch 85000, train_loss: nan
epoch 90000, train_loss: nan
epoch 95000, train_loss: nan
epoch 100000, train_loss: nan
```