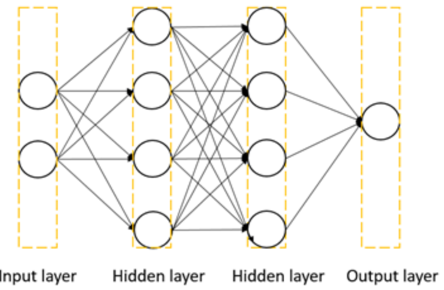
DL LAB1 Report - Back propagation

匯出pdf之後排版變得有點醜 QQ
助教不介意的話可以到HackMD網址閱讀～～～謝謝
https://hackmd.io/UsGKuF5yTVmfr3PGU0BKsw

(https://hackmd.io/UsGKuF5yTVmfr3PGU0BKsw)

1. Introduction

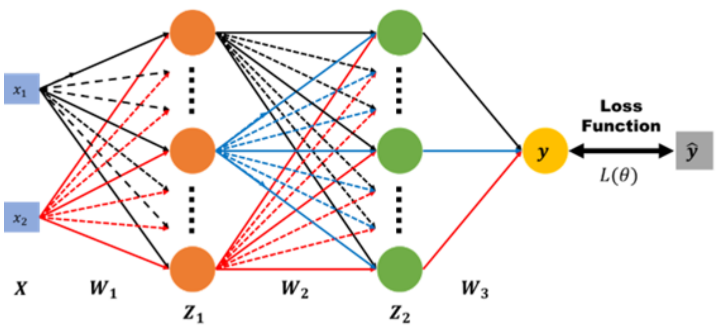
In this lab, we need to understand and implement simple neural networks with forwarding pass and backpropagation.



Request

- Implement simple neural networks with two hidden layers.
- You must use backpropagation in this neural network and can only use Numpy and other python standard libraries to implement.
- Plot your comparison figure that show the predicted results and the ground-truth.

Details



1. x_1, x_2 : *nerual network inputs*
2. X : $[x_1, x_2]$
3. y : *nerual network outputs*
4. $\hat{y}$: *ground truth*
5. $L(\theta)$: *loss function*
6. W_1, W_2, W_3 : *weight matrix of network layers*

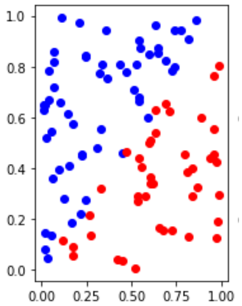
$Z_1 = \sigma(XW_1)$ $\quad Z_2 = \sigma(Z_1W_2)$ $\quad y = \sigma(Z_2W_3)$ $\quad \sigma(x) = \frac{1}{1+e^{-x}}$

Dataset

- Linear

```
def generate_linear(n=100):
    pts = np.random.uniform(0, 1, (n, 2))
    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)

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



- XOR

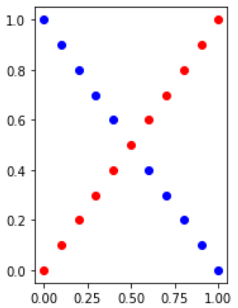
```
def generate_XOR_easy():
    inputs = []
    labels = []

    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)
```

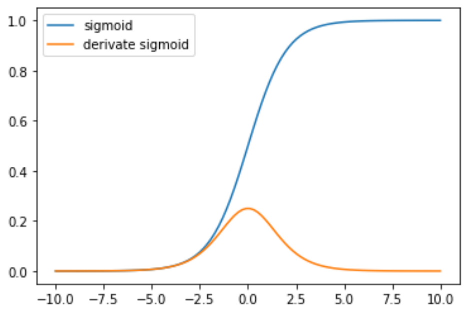


2. Experiment setups

A. Sigmoid functions

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

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



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

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

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

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

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

Sigmoid是一個映射函數，把變量映射到[0, 1]之間，通常被用來當作neural networks的activation function。
最常使用的情形，就是做binary classification的時候。將model的最後一層layer設定為只有一個neural unit，再將最後輸出的值傳入sigmoid function，就會得到一個介於[0, 1]之間的數值。最後只需要設定threshold，例如將小於0.5的值判斷為0、大於0.5的值判斷為1，就可以做出binary classification的預測。

B. Neural network

```
class myNet():
    def __init__(self, sizes, learning_rate, activation):
        self.learning_rate = learning_rate
        sizes_out = sizes[1:] + [0]
        self.layer = []
        for a, b in zip(sizes, sizes_out):
            if b == 0:
                continue
            elif b == 1:
                self.layer += [myLayer(a, b, 'Sigmoid')]
            else:
                self.layer += [myLayer(a, b, activation)]

    def forward(self, x):
        for l in self.layer:
            x = l.forward(x)
        return x

    def backward(self, dC):
        for l in self.layer[::-1]:
            dC = l.backward(dC)
        return dC

    def update(self):
        gradients = []
        for l in self.layer:
            gradients += [l.update(self.learning_rate)]
        return gradients
```

C. Backpropagation

- Chain Rule

  目標是要minimize loss function的cost C，但∂C/∂w不易計算，所以要用chain rule

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

- Forward

```python
def forward(self, x):
    self.forward_pass = x
    self.y = np.matmul(x, self.w)
    if self.activation == 'Sigmoid':
        self.y = sigmoid(self.y)

    return self.y
```

算式：$\frac{\partial z}{\partial w} = \frac{\partial x' w}{\partial w} = x'$

- Backward

```python
def backward(self, derivative_C):
    if self.activation == 'Sigmoid':
        self.backward_pass = np.multiply(
        derivative_sigmoid(self.y),
        derivative_C
    )
    elif self.activation == 'None':
        self.backward_pass = derivative_C

    return np.matmul(self.backward_pass, self.w.T)
```

算式：$\frac{\partial C}{\partial z} = \frac{\partial y}{\partial z}\frac{\partial C}{\partial y} \qquad y = \sigma(z), \frac{\partial y}{\partial z} = \sigma'(z)$

- Gradient descent

```python
def update(self, learning_rate):
    self.gradient = np.matmul(
        self.forward_pass.T,
        self.backward_pass
    )
    self.w -= learning_rate * self.gradient
    return self.gradient
```

算式：$w = w - \eta \Delta w$

3. Result of your testing

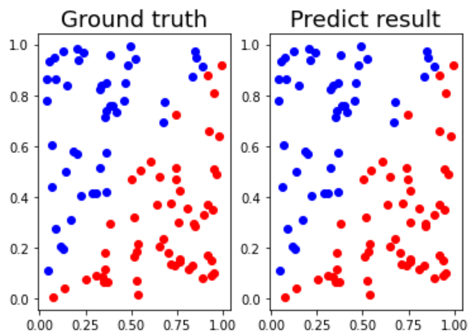
With 4 hidden units and sigmoid function.

A. Screenshot and comparision figure

- Linear

```
epoch   500 loss : 0.5085
epoch  1000 loss : 0.2066
epoch  1500 loss : 0.1214
epoch  2000 loss : 0.0828
epoch  2500 loss : 0.0615
epoch  3000 loss : 0.0482
epoch  3500 loss : 0.0393
epoch  4000 loss : 0.0329
epoch  4500 loss : 0.0282
epoch  5000 loss : 0.0245
epoch  5500 loss : 0.0217
epoch  6000 loss : 0.0193
epoch  6500 loss : 0.0174
epoch  7000 loss : 0.0158
epoch  7500 loss : 0.0145
epoch  8000 loss : 0.0133
epoch  8500 loss : 0.0123
epoch  9000 loss : 0.0114
epoch  9500 loss : 0.0107
epoch 10000 loss : 0.0100
epoch 10500 loss : 0.0094
epoch 11000 loss : 0.0088
epoch 11500 loss : 0.0084
epoch 12000 loss : 0.0079
epoch 12500 loss : 0.0075
epoch 13000 loss : 0.0071
epoch 13500 loss : 0.0068
epoch 14000 loss : 0.0065
epoch 14500 loss : 0.0062
epoch 15000 loss : 0.0060
epoch 15500 loss : 0.0057
epoch 16000 loss : 0.0055
epoch 16500 loss : 0.0053
epoch 17000 loss : 0.0051
```

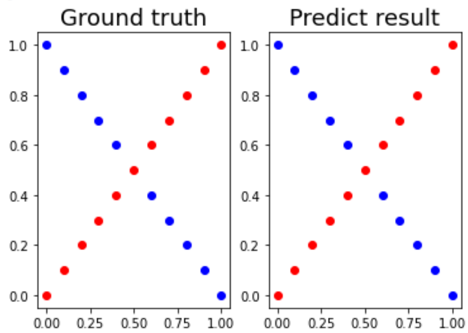
```
[[9.99999940e-01]
 [3.72272842e-06]
 [9.99973649e-01]
 [9.99999931e-01]
 [3.51397975e-05]
 [9.99999961e-01]
 [4.96985701e-06]
 [9.99997386e-01]
 [4.35247145e-06]
 [9.99999937e-01]
 [3.70567958e-06]
 [9.99999938e-01]
 [9.99491535e-01]
 [3.61019610e-06]
 [3.65872731e-06]
 [9.99999962e-01]
 [3.87776966e-06]
 [3.57111566e-06]
 [9.99734086e-01]
 [9.99993623e-01]
 [5.08162284e-05]
 [9.99999624e-01]
 [9.99999962e-01]
 [5.90316637e-06]
 [9.99999954e-01]
 [9.99999932e-01]
 [9.99999540e-01]
 [4.44856100e-06]
 [9.99999962e-01]
 [9.99999934e-01]
 [9.99999955e-01]
 [1.78554628e-03]
 [9.99999933e-01]
 [9.99779406e-01]
```



- XOR

```
epoch 1000 loss : 0.3908
epoch 1500 loss : 0.1575
epoch 2000 loss : 0.0966
epoch 2500 loss : 0.0693
epoch 3000 loss : 0.0539
epoch 3500 loss : 0.0441
epoch 4000 loss : 0.0372
epoch 4500 loss : 0.0322
epoch 5000 loss : 0.0284
epoch 5500 loss : 0.0254
epoch 6000 loss : 0.0229
epoch 6500 loss : 0.0209
epoch 7000 loss : 0.0192
epoch 7500 loss : 0.0178
epoch 8000 loss : 0.0165
epoch 8500 loss : 0.0155
epoch 9000 loss : 0.0145
epoch 9500 loss : 0.0137
epoch 10000 loss : 0.0129
epoch 10500 loss : 0.0123
epoch 11000 loss : 0.0116
epoch 11500 loss : 0.0111
epoch 12000 loss : 0.0106
epoch 12500 loss : 0.0101
epoch 13000 loss : 0.0097
epoch 13500 loss : 0.0093
epoch 14000 loss : 0.0090
epoch 14500 loss : 0.0087
epoch 15000 loss : 0.0084
epoch 15500 loss : 0.0081
epoch 16000 loss : 0.0078
epoch 16500 loss : 0.0075
epoch 17000 loss : 0.0073
epoch 17500 loss : 0.0071
epoch 18000 loss : 0.0069
epoch 18500 loss : 0.0067
epoch 19000 loss : 0.0065
epoch 19500 loss : 0.0063
epoch 20000 loss : 0.0062
epoch 20500 loss : 0.0060
epoch 21000 loss : 0.0059
epoch 21500 loss : 0.0057
epoch 22000 loss : 0.0056
epoch 22500 loss : 0.0055
epoch 23000 loss : 0.0053
epoch 23500 loss : 0.0052
epoch 24000 loss : 0.0051
```

```
[[3.32711933e-04]
 [9.99673376e-01]
 [3.31094709e-04]
 [9.99672752e-01]
 [3.29511369e-04]
 [9.99675201e-01]
 [3.27961683e-04]
 [9.99672727e-01]
 [3.26445505e-04]
 [9.99077987e-01]
 [3.24962776e-04]
 [3.23513524e-04]
 [9.99139958e-01]
 [3.22097878e-04]
 [9.99752070e-01]
 [3.20716076e-04]
 [9.99768377e-01]
 [3.19368471e-04]
 [9.99771663e-01]
 [3.18055542e-04]
 [9.99771045e-01]]
```



B. Show the accuracy of your prediction

```python
print(f'Accuracy : {float(np.sum(y == pred_y)) * 100 / len(y)} %')
```

- Linear

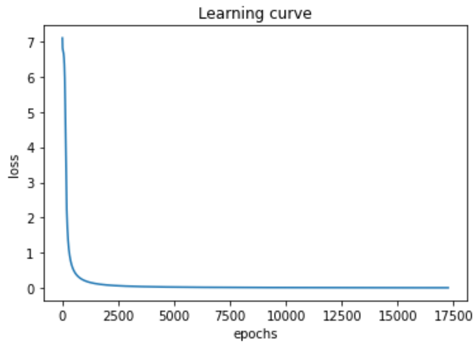
```
Accuracy : 100.0 %
```

- XOR

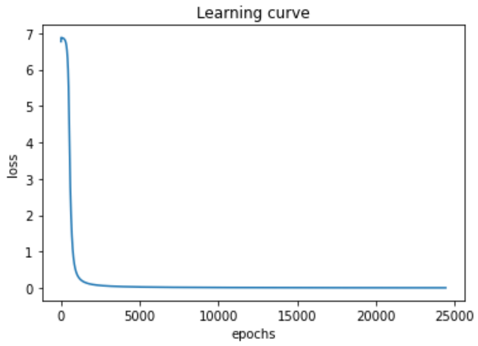
```
Accuracy : 100.0 %
```

C. Learning curve (loss, epoch curve)

- Linear



- XOR

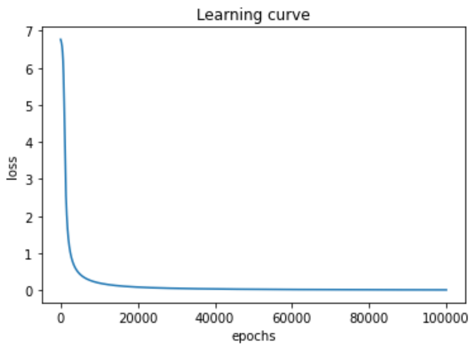


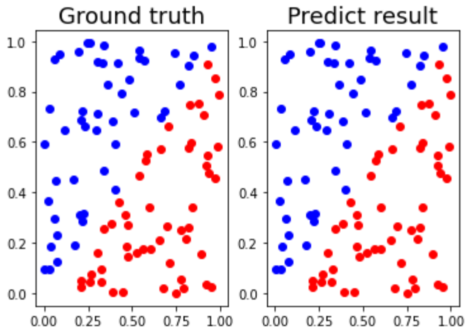
4. Discussion

A. Try different learning rates

- Learning rate = 0.1

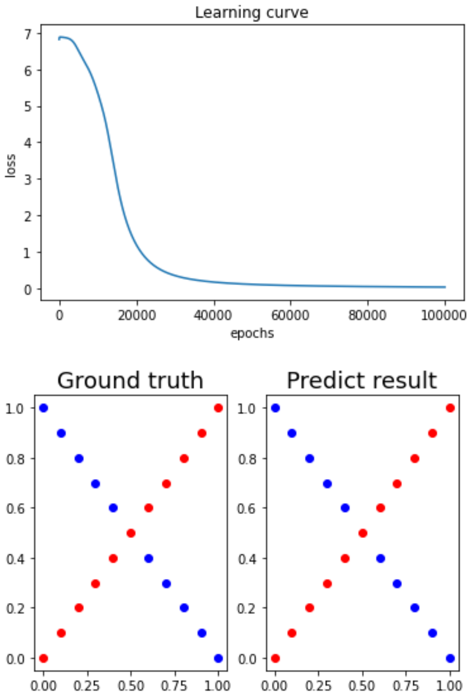
Linear






```
Accuracy :  100.0 %
```

XOR

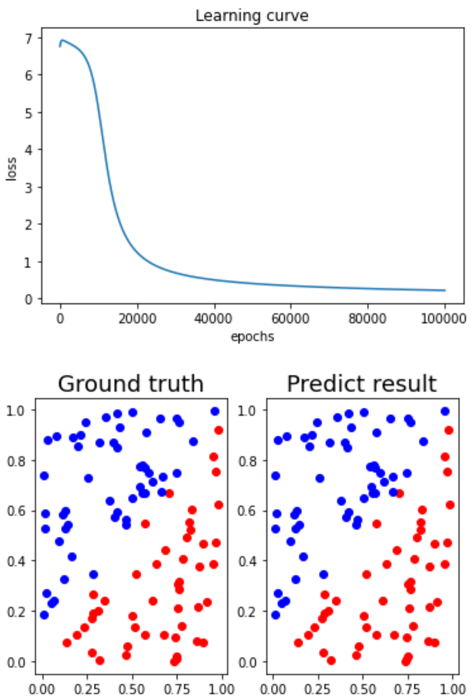


Accuracy : 100.0 %

跟learning rate = 1比起來，loss下降的較慢，在epoch 100000的loss比learning rate = 1的epoch 25000的loss還要高，但accuracy一樣可以達到100%。

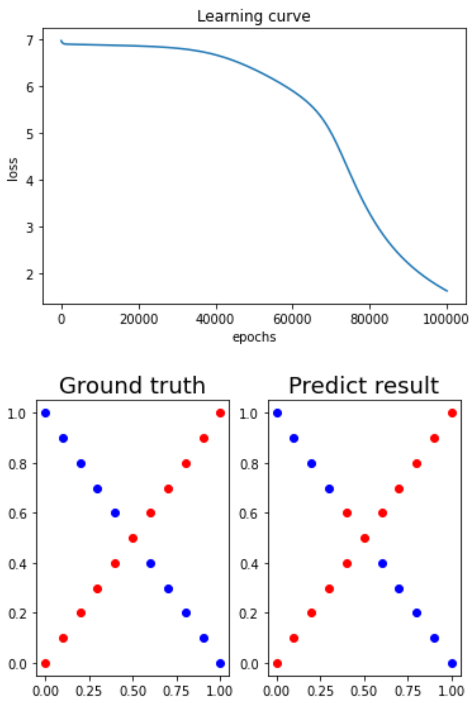
- Learning rate = 0.01

Linear



Accuracy : 100.0 %

XOR

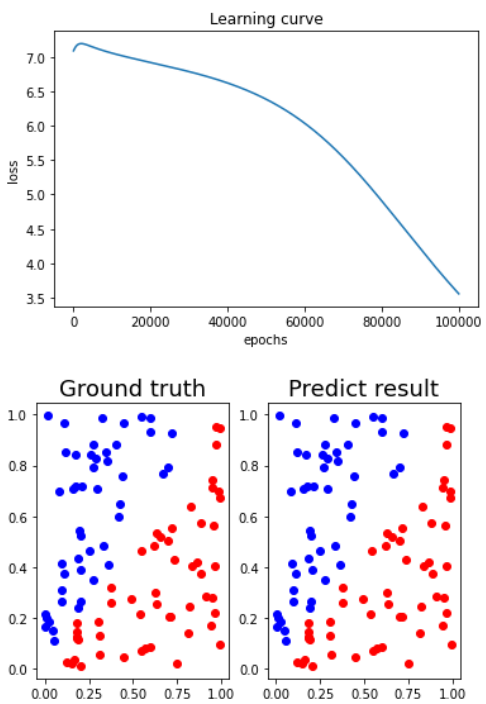


Accuracy : 95.23809523809524 %

learning rate = 0.01時，雖然在learning curve上看起來還是有持續下降的，但非常緩慢，可能需要更多的epoch才會達到更高的accuracy（若沒有early stop，最多只會跑100000個epoch）。

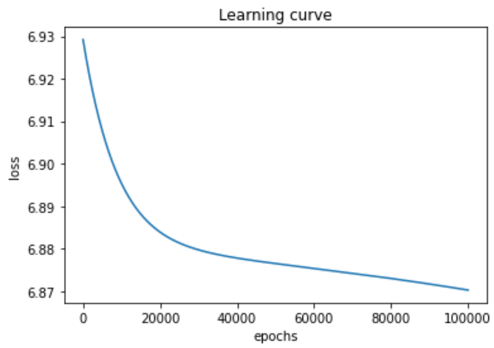
- Learning rate = 0.001

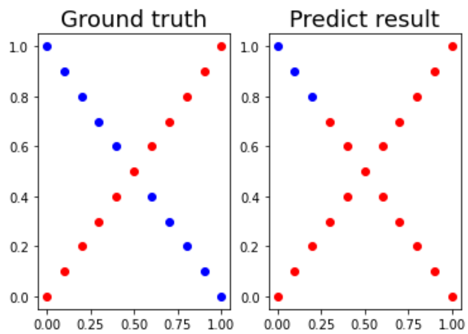
Linear



Accuracy : 100.0 %

XOR





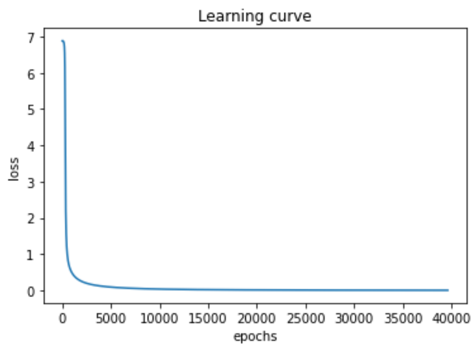
Accuracy : 66.66666666666667 %

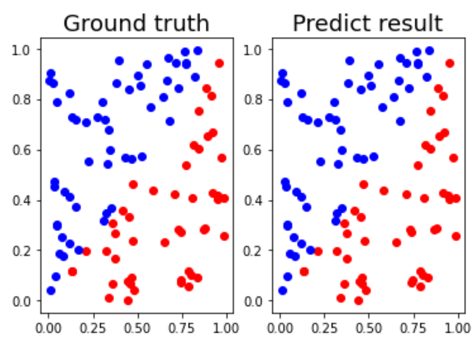
learning rate = 0.001時，loss看起來又下降得更慢了，所以在相同epoch 100000的情況，accuray就會更低。

B. Try different numbers of hidden units

- 2 hidden units

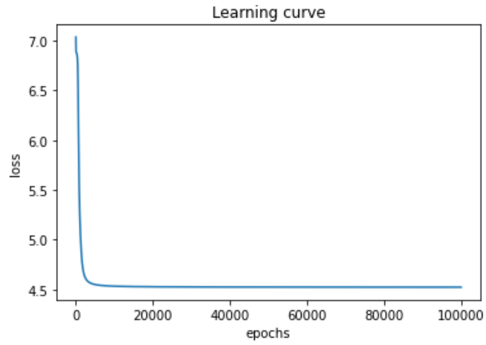
Linear

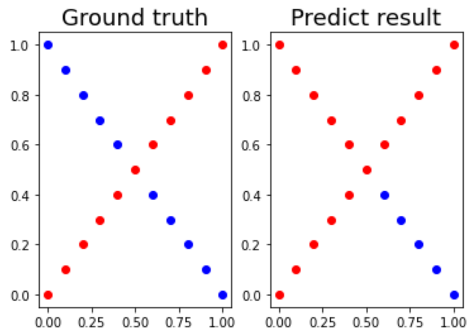




Accuracy : 100.0 %

XOR



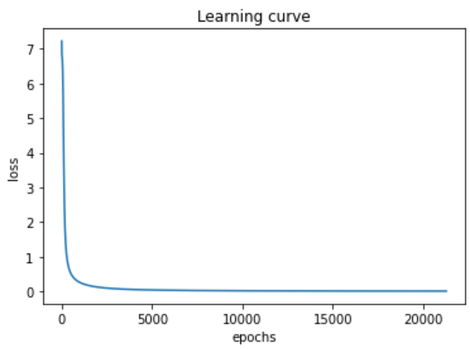


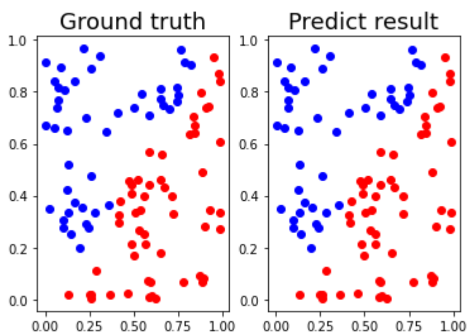
Accuracy : 76.19047619047619 %

相較於4 hidden units，2 hidden units的loss下降的較慢，並且在XOR data上沒辦法有很好的accuracy。

- 8 hidden units

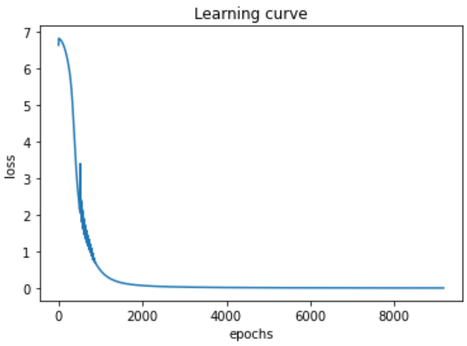
Linear

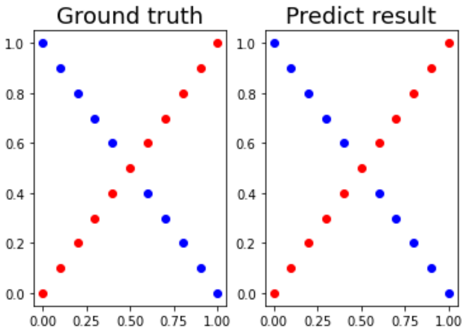




Accuracy : 100.0 %

XOR



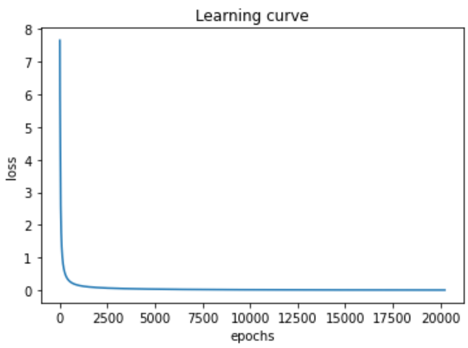


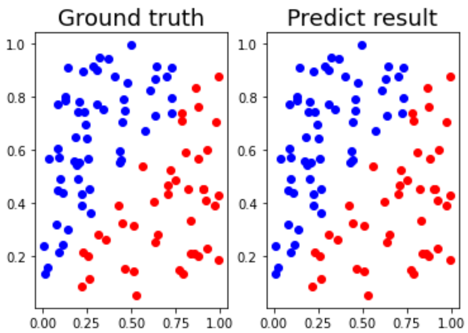
Accuracy : 100.0 %

相較於4 hidden units，8 hidden units在XOR data的loss收斂得更快。

- 16 hidden units

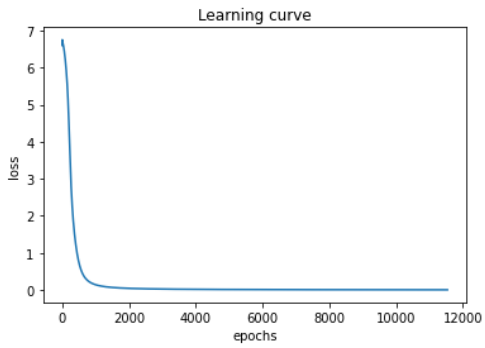
Linear

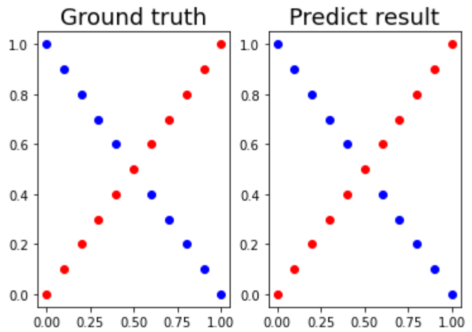




Accuracy : 100.0 %

XOR



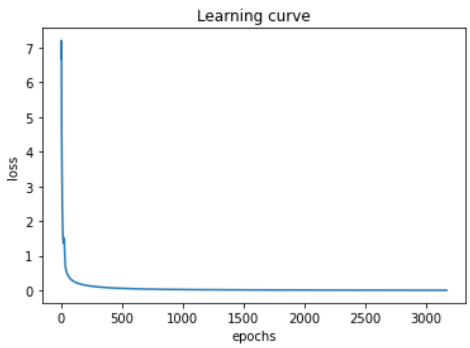


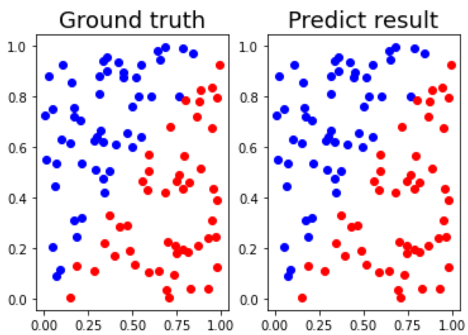
Accuracy : 100.0 %

16和4 hidden units較無差異

C. Try without activation functions

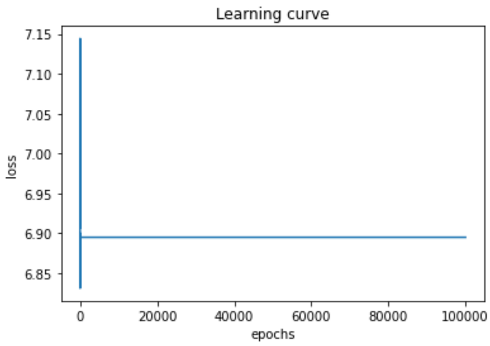
- Linear

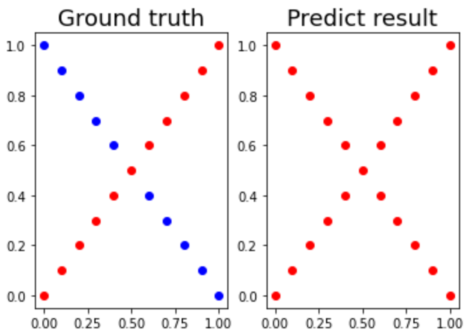




Accuracy : 100.0 %

- XOR






```
Accuracy : 52.38095238095238 %
```

因為XOR data非線性，所以拿掉sigmoid後結果會變得非常差