1. Introduction

在這次的 lab 中，我實作了一個有兩層 hidden layer 的簡單的 neural network。將 input 餵進網路中後，可以利用 forward propagation 算出此 input 預測的分類值(正確來說是算出一個數字，此數字大於 0.5 的話就將此 input 預測為 label 1，反之則將此 input 預測為 label 0)。為了使 neural network 中的參數們(weight)能夠足夠合適(能使 loss function 有最小值的 weight 們)，將 input 預測為正確的分類，就必須在訓練過程中一步步調整參數的值，直到訓練完成。神經網路預測出的數值和真實答案的差距(loss)經過 back propagation 的向後計算之後，就可以得到這些參數更新的方向，使得預測結果可以越來越接近真實答案。

除了實作出神經網路之外，在此 lab 中也會透過調整各種不同的 hyper parameter(如網路的 hidden width、learning rate 等等)，來觀察它們所造成的各種影響。

2. Experiment setups

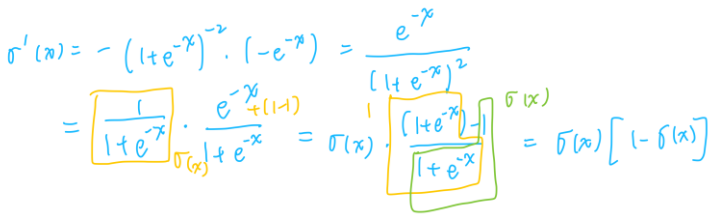
A. Sigmoid functions

在神經網路中，前一層 input 和 weight 做內積之後，會再通過一層非線性的 activation function，才正式變成那一層的 output。在這裡 sigmoid function 就是做為這個非線性的 activation function。

下圖為 sigmoid function 的定義

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

下圖則為計算出的 sigmoid function 的微分

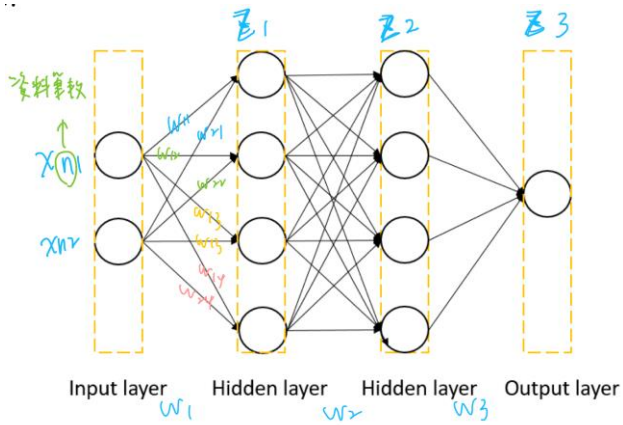


因此參考了助教的 hint 之後，code 中的 derivative_sigmoid 的 input 值應該為要微分位置的 sigmoid output 值，而不是要微分的位置。

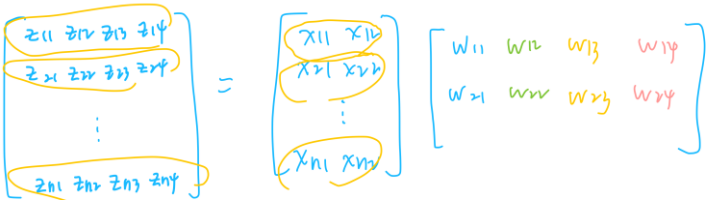
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def derivative_sigmoid(x):
    # the input of this function should be the output of the sigmoid function
    return np.multiply(x, 1 - x)    # multiply arguments element-wise -> not matrix multiplication
```

B. Neural network

參考 spec 中所附的網路架構圖，我的網路中儲存了各個 weight 以及各個 hidden layer 的 output 值，才能在 forward propagation 以及 back propagation 時都將他們儲存起來。



這是當所有 input data 一起和第一層 layer weight 做內積的矩陣乘法。在這次時做的神經網路中，每次只會有一筆 data 被送進來，因此我的各個神經網路輸入輸出都是以列向量的方式儲存，weight 則是以矩陣方式儲存，wij 代表的是要和前一層第 i 個 neuron 值做相乘，成為此層第 j 個 neuron 值一部份的 weight。



```
'''
x->(w1)->a1->(sig1)->z1->(w2)->a2->(sig2)->z2->(w3)->a3->(sig3)->out==y
'''
np.random.seed(0)
self.x = np.zeros((1, 2))
self.w1 = np.random.rand(2, hidden_width[0])
self.w2 = np.random.rand(hidden_width[0], hidden_width[1])
self.w3 = np.random.rand(hidden_width[1], 1)

self.a1 = np.zeros((1, hidden_width[0]))
self.a2 = np.zeros((1, hidden_width[1]))
self.a3 = np.zeros((1, 1))

self.z1 = np.zeros((1, hidden_width[0]))
self.z2 = np.zeros((1, hidden_width[1]))
self.z3 = np.zeros((1, 1))
self.loss = np.zeros((1, 1))
```

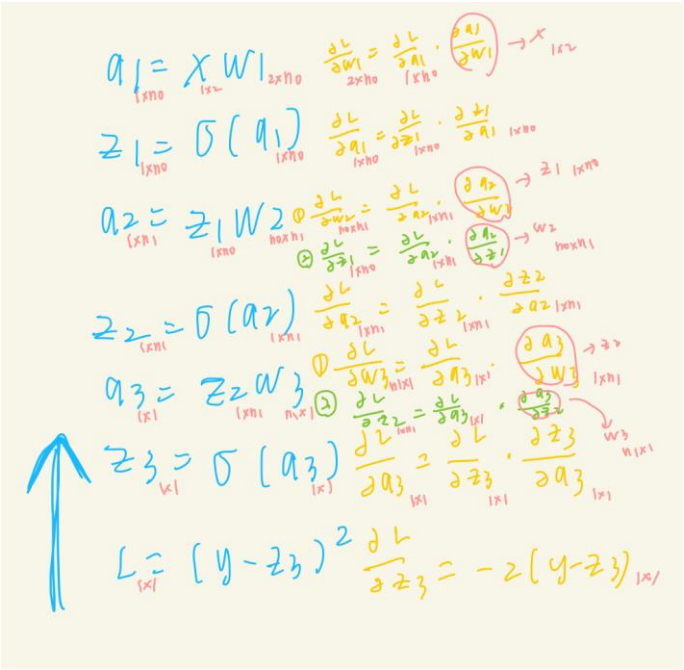
因為每次只會有一筆 data 丟進神經網路，forward 的寫法就是照順序把各層都串起來。

```
def forward(self, x):
    # x.shape = (1, 2)
    self.x[0, 0] = x[0]
    self.x[0, 1] = x[1]
    self.a1 = self.x@self.w1
    self.z1 = sigmoid(self.a1)
    self.a2 = self.z1@self.w2
    self.z2 = sigmoid(self.a2)
    self.a3 = self.z2@self.w3
    self.z3 = sigmoid(self.a3)
    out = self.z3
    return out
```

C. Backpropagation

在 backpropagation 的一開始，我先計算 min square error 來作為 loss function。

接著再將下圖寫出來的微分式子由下往上一步步算出來，前面算出來的結果可以給後面要算微分的式子用的。比較要注意的是計算微分時的矩陣維度，以及相乘所用的符號。'*'是 element-wise 的乘法，'@'則是一般的矩陣乘法。像是 sigmoid 的微分時，因為原本 sigmoid function 的 input 和 output 維度就相同，所以在微分的 chain rule 相乘時，我就會使用 element-wise 的'*'。像是計算 weight 的微分時，用的就會是矩陣乘法的'@'，並且也要注意何時要轉置矩陣。



```
def backward(self, pred_y, y):
    self.loss[0] = (pred_y - y) * (pred_y - y)                  # min square error

    # pred_y == self.z3

    grad_loss_z3 =  -2 * (y - self.z3)                          # shape = (1, 1)

    grad_z3_a3 = derivative_sigmoid(self.z3)                    # shape = (1, 1)
    grad_loss_a3 = grad_loss_z3 * grad_z3_a3                    # shape = (1, 1)

    grad_a3_w3 = self.z2                                        # shape = (1, h1)
    grad_loss_w3 = grad_a3_w3.T @ grad_loss_a3                  # shape = (h1, 1)

    grad_a3_z2 = self.w3                                        # shape = (h1, 1)
    grad_loss_z2 = grad_loss_a3 @ grad_a3_z2.T                  # shape = (1, h1)

    grad_loss_a2 = grad_loss_z2 * derivative_sigmoid(self.z2)   # shape = (1, h1)

    grad_a2_w2 = self.z1                                        # shape = (1, h0)
    grad_loss_w2 = grad_a2_w2.T @ grad_loss_a2                  # shape = (h0, h1)

    grad_a2_z1 = self.w2                                        # shape = (h0, h1)
    grad_loss_z1 = grad_loss_a2 @ grad_a2_z1.T                  # shape = (1, h0)

    grad_loss_a1 = grad_loss_z1 * derivative_sigmoid(self.z1)   # shape = (1, h0)

    grad_a1_w1 = self.x                                         # shape = (1, 2)
    grad_loss_w1 = grad_a1_w1.T @ grad_loss_a1                  # shape = (2, h0)
```

算出 loss 對 weight 的微分之後，因為此 gradient 所指的方向是指向 loss 升高最多的地方，而我們所想要的是 loss 最低的地方，所以會用加負號來更新 weight，同時也會在乘上 learning rate 後再更新參數。

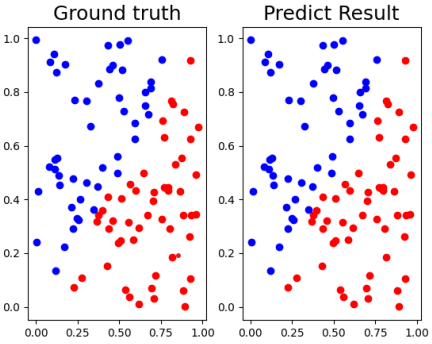
```
# update weights
self.w1 -= self.lr * grad_loss_w1
self.w2 -= self.lr * grad_loss_w2
self.w3 -= self.lr * grad_loss_w3
```

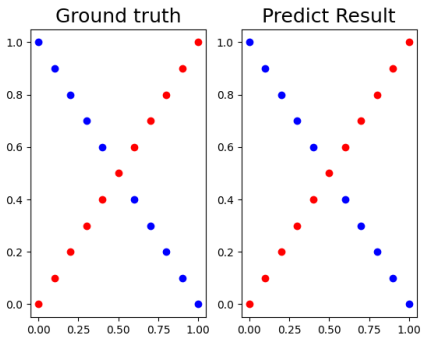
3. Results of your testing

A. Screenshot and comparison figure

兩種不同的 input 資料 train 到最後都能在 test 時獲得 100%準確率。

Linear	XOR
6000 epochs	60000 epochs





B. Show the accuracy of your prediction

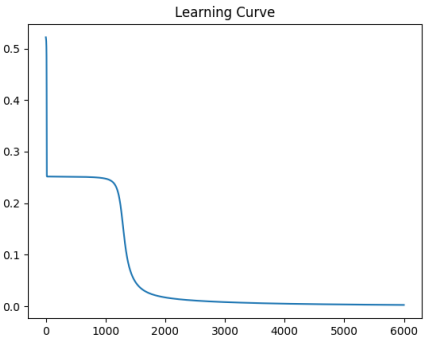
Linear	XOR

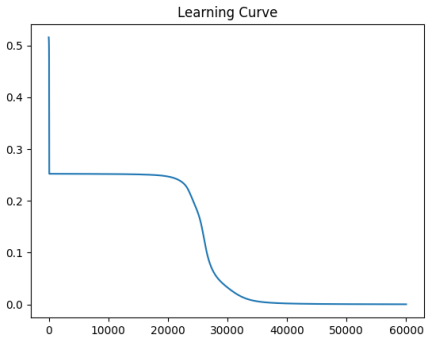
```
Data90:    Ground truth: 0, Prediction: 0.00007
Data91:    Ground truth: 0, Prediction: 0.00038
Data92:    Ground truth: 1, Prediction: 0.98827
Data93:    Ground truth: 1, Prediction: 0.99943
Data94:    Ground truth: 1, Prediction: 0.99986
Data95:    Ground truth: 1, Prediction: 0.99983
Data96:    Ground truth: 1, Prediction: 0.99984
Data97:    Ground truth: 0, Prediction: 0.00088
Data98:    Ground truth: 0, Prediction: 0.00007
Data99:    Ground truth: 1, Prediction: 0.99978
Accuracy: 1.0 (100/100), Loss: 0.24235391113722435
```

```
Data11:    Ground truth: 0, Prediction: 0.22585
Data12:    Ground truth: 1, Prediction: 0.60708
Data13:    Ground truth: 0, Prediction: 0.14629
Data14:    Ground truth: 1, Prediction: 0.94896
Data15:    Ground truth: 0, Prediction: 0.08693
Data16:    Ground truth: 1, Prediction: 0.98732
Data17:    Ground truth: 0, Prediction: 0.05097
Data18:    Ground truth: 1, Prediction: 0.99063
Data19:    Ground truth: 0, Prediction: 0.03087
Data20:    Ground truth: 1, Prediction: 0.98861
Accuracy: 1.0 (21/21), Loss: 0.6511811145594609
```

C. Learning curve (loss, epoch curve)

Linear	XOR





D. anything you want to present

比起 linear 的資料，XOR 的資料相對來說是比較困難的問題，不僅 training 到收斂所需的 epoch 數比 linear 多(大概是 10 倍的關係)，從 learning curve 中也可以看出，雖然兩者一開始的 loss 都會快速下降，可是在到完全收斂時，都會有一段 loss 有點停滯下不去的階段，這個停滯的階段，XOR 的也比 linear 的長很多，接著再到最後終於開始下降時，雖然相比停滯的時候已經是快速下降了，但 XOR 到最後收斂時大約也花了 10000 個 epoch。另一個造成 XOR 的 training epoch 數比 linear 多的可能的原因則是資料筆數，linear 的資料筆數有 100 筆，但 XOR 的資料筆數只有 20 筆。

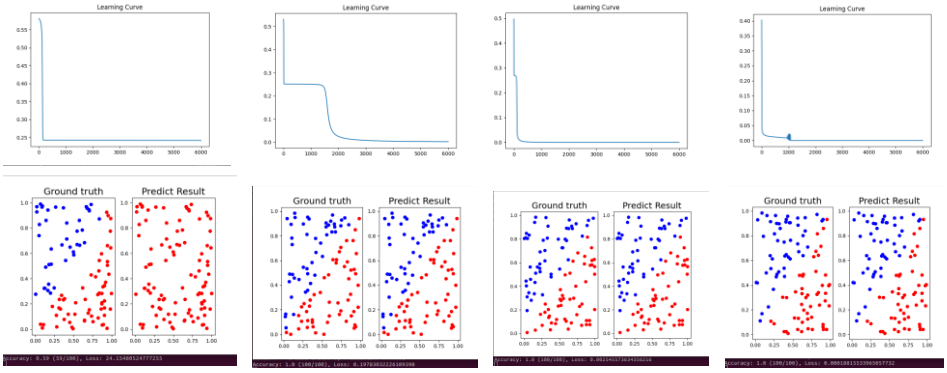
4. Discussion

A. Try different learning rates

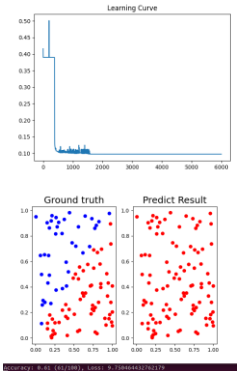
a. Linear

先固定其他參數，hidden width = (10, 10) (兩層 hidden layer 都各 10 個 neuron)、training epochs = 6000

Lr = 0.001	Lr = 0.01	Lr = 0.1	Lr = 1



Lr = 7



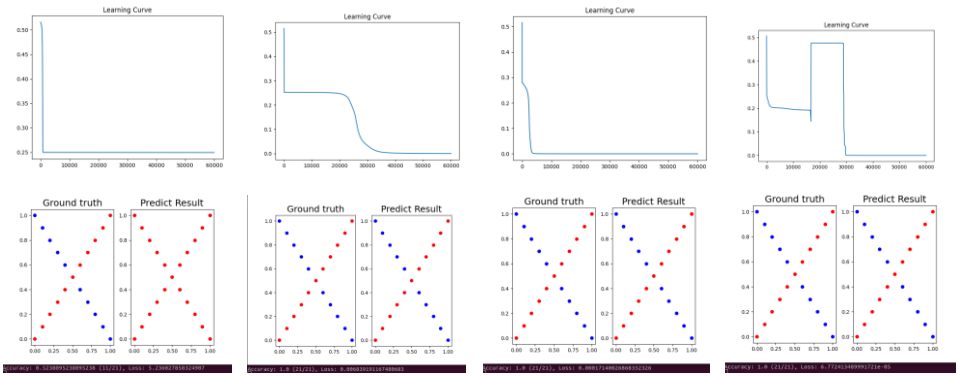
由圖中可以看到，當 lr 很小時(0.001)，train 到 6000 個 epoch 的時候他其實還沒收斂，準確率只有近六成左右。而當 lr 變大，lr=0.01~1 左右時，到 6000 個 epoch 時雖然都有收斂，不過可以觀察出這三個在收斂過程中的分別。Lr=0.01 時，其實 lr 還是偏小，因此在 loss curve 中有一段明顯 loss 停滯不往下的階段，這可能就是因為 lr 太小，所以對參數的改變不大造成的。Lr=0.1 則相對是一個不錯的 lr。到了 Lr=1 時，雖然有順利收斂，不過因為 lr 偏大的關係，在收斂到最低點時會有震盪的現象。Lr=7 則過大，震盪更明顯且無法收斂至 loss=0。

b. Xor

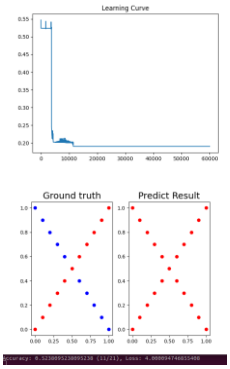
一樣先固定其他參數，hidden width = (10, 10) (兩層 hidden layer 都各

10 個 neuron)、training epochs = 60000

Lr = 0.001	Lr = 0.01	Lr = 0.1	Lr = 0.7



Lr = 10



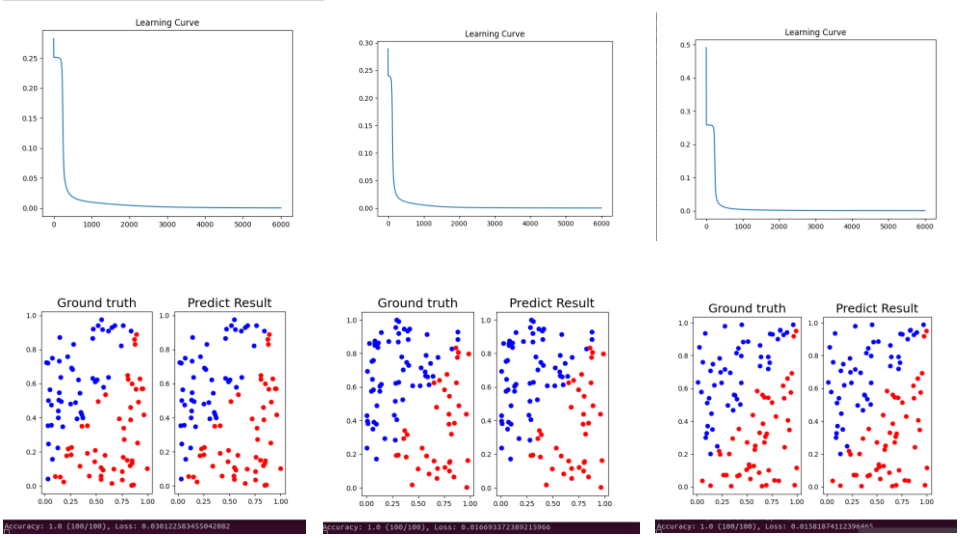
XOR 的結果和 linear 的也類似。不過在 lr=0.7 時，可能剛好那時參數的所在位置在陡峭的地方，因此更新太多時，讓他的 loss 變得非常大，雖然最後順利收斂了。

B. Try different numbers of hidden units

a. Linear

固定其他參數，lr=0.05、training epochs = 6000

Hidden_width = (2, 2)	Hidden_width = (6, 6)	Hidden_width = (10, 10)



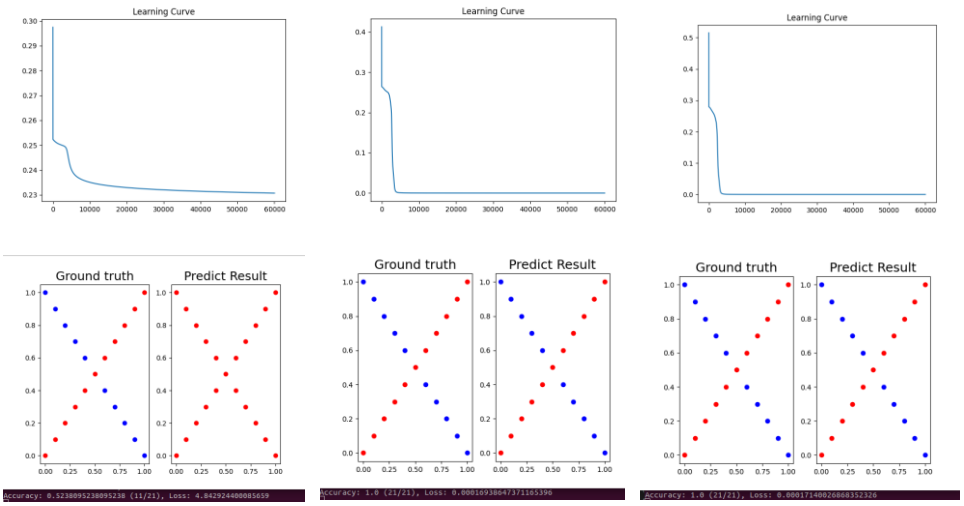
我覺得他們的結果主要差別可能是 loss 的初始值，因為 hidden unit 多的網路可以模擬比較複雜的函數，但可能這個 linear data 的複雜度並

沒有很高，所以在一開始 train 的初始 loss 時，由左到右 hidden unit 多的網路反而 loss 最多。

b. XOR

固定其他參數，lr=0.1、training epochs = 60000

Hidden_width = (2, 2)	Hidden_width = (6, 6)	Hidden_width = (10, 10)



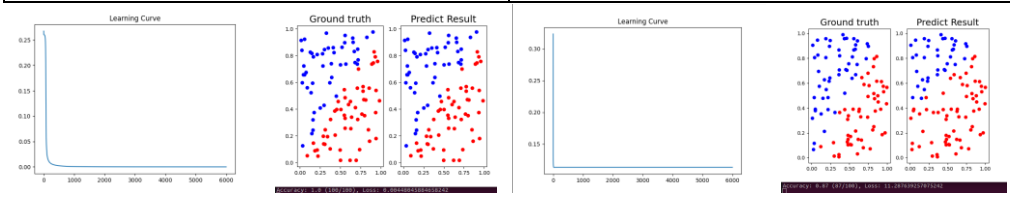
使用 XOR 作為資料，比較能看出 hidden unit 數所造成的影響。在 hidden unit 數只有 2*2 時，因為 model 能力較弱的關係，到了 epoch 60000 時，都無法收斂至 loss 為 0。當 hidden unit 數變多時，其他兩個結果都能順利收斂至 loss 約為 0 的地方。而且 hidden unit 數 10*10 的收現情形，稍微比 6*6 的再平滑一點點，可見在資料分布較難的情況下，hidden unit 數變多，有助於學習。

C. Try without activation functions

a. linear

固定其他參數，lr=0.1、training epochs = 6000、hidden units = 6*6

有 activation function	沒有 activation function

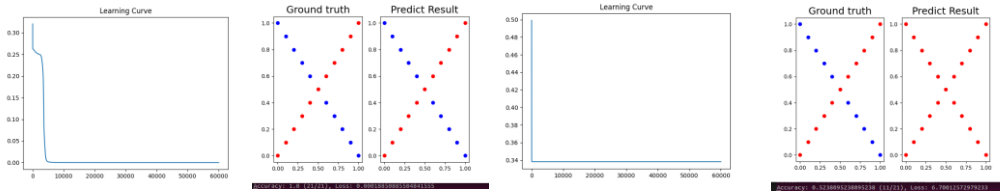


沒有加 activation function 的情況，因為 model 少了非線性元素在，能力變弱了，變得比較難 train。雖然理論上線性的 input 應該也能在少了 activation fuction 的 model 上 train 到 100%準確率，不過我調整了幾個參數後，都沒能達到 100%準確率，我在猜會不會是因為我實做的 model 並沒有加上 bias 有關。

b. XOR

固定其他參數，lr=0.1、training epochs = 60000、hidden units = 6*6

有 activation function	沒有 activation function



因為 xor 是非線性的關係，所以在沒有加 activation function 的線性 model 中，應該是沒有辦法 train 起來的，而且到最後的情況應該會趨近於亂猜(50%準確率)。跑出來的結果也和我的猜測很符合。

D. Anything you want to share

我在寫作業的時候，原本有碰到下面這種維度的問題，本來搞不清楚覺得下面例子中的 a 和 b[0]明明都是 row vector 為什麼 shape 卻一個是(1, 4)一個是(4,)，結果在矩陣乘法那裡花了一點時間。不過現在我搞懂了，雖然表面上他們感覺都是 row vector，不過 a 的寫法實際上是一個”二維”的一列四行的矩陣。

```
a = np.zeros((1, 4))
b = np.array([[0, 0, 0, 0], [0, 0, 0, 0]])
print(a, type(a), a.shape)
print(b[0], type(b[0]), b[0].shape)
```

```
wenxuan@wenxuan-System-Product-Name: ~/D
[[0. 0. 0. 0.]] <class 'numpy.ndarray'> (1, 4)
[0 0 0 0] <class 'numpy.ndarray'> (4,)
```