Deep Learning and Practice
Lab2 : back-propagation

310605009
吳公耀

一、 Introduction

A. Lab Objective

- 實作一個包含兩個 hidden layers 的 neural network
- 使用Numpy和其他Python函式庫，不可使用其他深度學習相關框架(e.g. temsorflow , pytorch)
- 透過forward propagation 得到預測答案，將預測答案和真實答案之間的差距經過 backpropagation 回傳，藉此來更新layer 的 weights，

B. Nations

- data、label：neural network inputs

```
data, label = GenData.fetch_data('Linear', 70)
```

```
data, label = GenData.fetch_data('XOR', 70)
```

- y : neural network outputs(predict)
- y_had : ground truth
- L(Θ) : loss function (MSE $\frac{1}{n}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$)

```
def loss(self, y, y_hat):
    #MSE
    return np.mean((y - y_hat)**2)

def derivative_loss(self, y, y_hat):
    # Differentiate with respect to N (total number of datasets)
    return (y - y_hat)*(2/y.shape[0])
```

- Weight : weight matrix for each networks layers

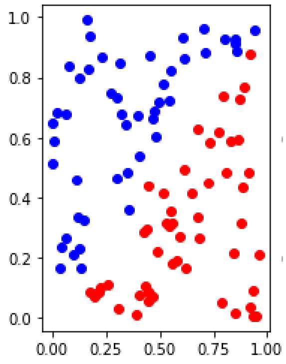
```
def __init__(self , input_size , output_size,learning_rate = 0.1 , optimizer ="default")

    # init the weights
    self.weights = np.random.normal(0, 1, (input_size+1, output_size))
```

```
def update(self):
    """
    update the weight
    default : by multiplication the learning rate
    adam : by adam optimizer
    """
    #calculates the gradient by multiplication output of the second hidden layer and backpropagated gradient
    self.gradient = np.matmul(self.forward_gradient.T, self.backward_gradient)

    if(self.optimizer  == 'adam'):
        self.moving_average_m = 0.9 * self.moving_average_m + 0.1 *  self.gradient #updates the moving averages of the gradie
        self.moving_average_v = 0.999 * self.moving_average_v + 0.001 * np.square( self.gradient) #updates the moving average
        bias_correction_m = self.moving_average_m / (1.0 - 0.9 ** self.update_times) #calculates the bias-corrected estimates
        bias_correction_v = self.moving_average_v / (1.0 - 0.999 ** self.update_times) #calculates the bias-corrected estimat
        self.update_times += 1
        delta_weight = -self.learning_rate * bias_correction_m / (np.sqrt(bias_correction_v) + 1e-8)
    else :
        delta_weight = -(self.learning_rate*self.gradient)
    self.weights += delta_weight
    return self.gradient
```

C. Dataset

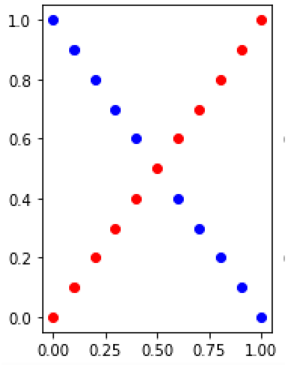
- Linear

```python
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

```python
def generate_xor(n=100):
    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)
```



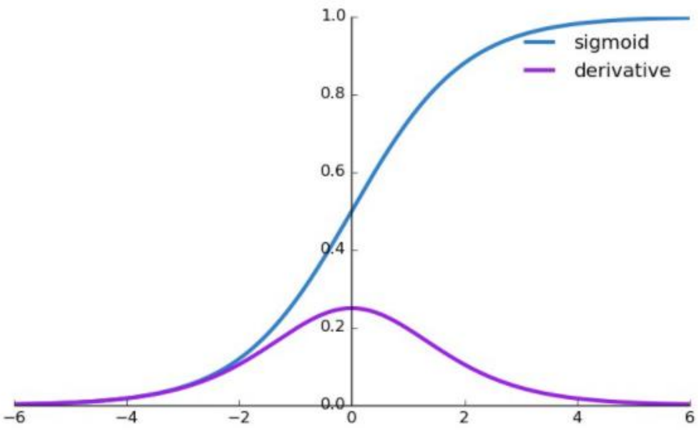
二、 Experiment setups

A. Sigmoid Functions

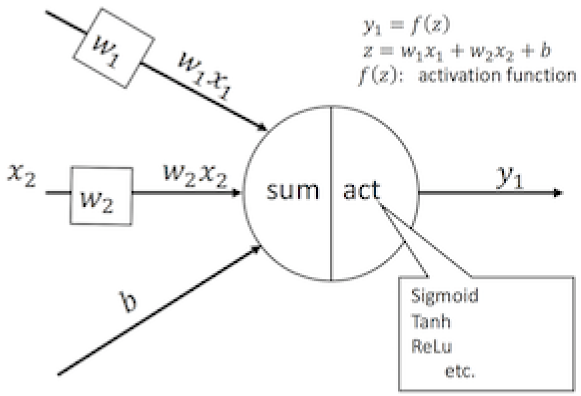
Sigmoid Functions = $\sigma(x) = \frac{1}{1+e^{-x}}$

$Z_1 = \sigma(XW_1)$, $Z_2 = \sigma(XW_2)$, $Z_3 = \sigma(XW_3)$

```
def activation_function(self,x,act):
    return {
    'sigmoid': lambda x: 1 / (1 + np.exp(-x)),     #calculate sigmoid function
    'tanh': lambda x: np.tanh(x) ,                 #calculate tanh function
    'relu': lambda x: np.maximum(0.0, x),          #calculate relu function
    }[act](x)
def de_activation_function(self,x,act):
     return {
    'sigmoid': lambda x: np.multiply(x, 1.0 - x), #calculate the derivative of  sigmoid function
    'tanh': lambda x: 1.0 - x ** 2,                #calculate the derivative of  tanh function
    'relu': lambda x:  1. * (x > 0),               #calculate the derivative of  relu function
    }[act](x)
```



1. **激活函數(activation function)**



在類神經網路中使用激活函數，目的為利用非線性方程式，解決非線性問題，若不使用激活函數的神經網絡本質上只是一個線性回歸模型，激活函數對輸入進行非線性變換，使其能夠學習和執行更複雜的任務。而在現實應用上，所有問題皆為非線性問題，沒有激活函數的神經網絡本質上只是一個線性回歸模型。

2. **數學推導**

(1) Derivative Sigmod 推導

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

Differentiate x

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

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

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

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

(2) 在 back propagation 進行偏微分

假設x = wz + b

對w和b進行偏微分則可由原本對x的微分做延伸，透過Chain Rule 將$-e{-x}$ 對w和b進行偏微：

$$\frac{\partial_\sigma}{\partial_w} = \frac{\partial_\sigma}{\partial_x} \times \frac{d}{d_w}e^{-x} = \frac{\partial_\sigma}{\partial_x} \times z$$

$$\frac{\partial_\sigma}{\partial_b} = \frac{\partial_\sigma}{\partial_x} \times \frac{d}{d_b}e^{-x} = \frac{\partial_\sigma}{\partial_x}$$

3. **優缺點及應用**

(1) 優點：

a. 便於求導的連續函數

b. 能壓縮數據，保證幅度不會有問題

c. 輸出在(0,1)之間，輸出範圍有限，優化穩定，可以用作輸出層

(2) 缺點：

a. 容易出現梯度消失（Vanishing Gradient）

b. Sigmoid 輸出不是 0均值（zero - centered）

c. 冪運算相對來講比較耗時且複雜度高

4. **討論**

(1) 梯度消失 Vanishing Gradient

優化神經網絡的方法是 back-propagation，即導數的後向傳遞

先計算輸出層對應的 loss，然後將 loss以導數的形式不斷向上一層網絡傳遞，修正相應的參數達到降低 loss的。在深度網絡中，常會使導數逐漸轉變為0，使得參數無法被更新 。原因在於兩點：

i. 在上圖中，當σ(x)中σ(x)]較大或較小時，導數接近0，而後向傳遞是依據Chain rule求導，當前層的導數需要之前各層導數的乘積，幾個小數的相乘，結果會很接近0

ii. Sigmoid導數的平均值是0.25，這表示導數在每一層至少會被壓縮為原來的1/4，通過兩層後被轉化為1/16，…，通過10層後為1/1048576。

(2)適合使用

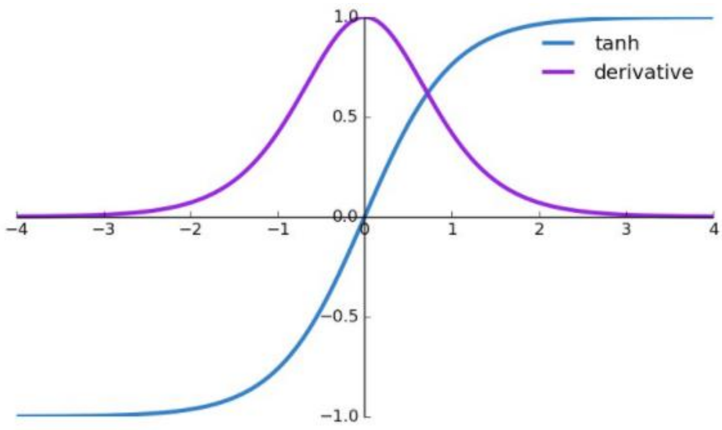
i. 如果您希望輸出值介於 0 到 1 之間，請僅在輸出層神經元使用 sigmoid

ii. 當你在做二進制分類問題時使用 sigmoid

5. 其他激活函數

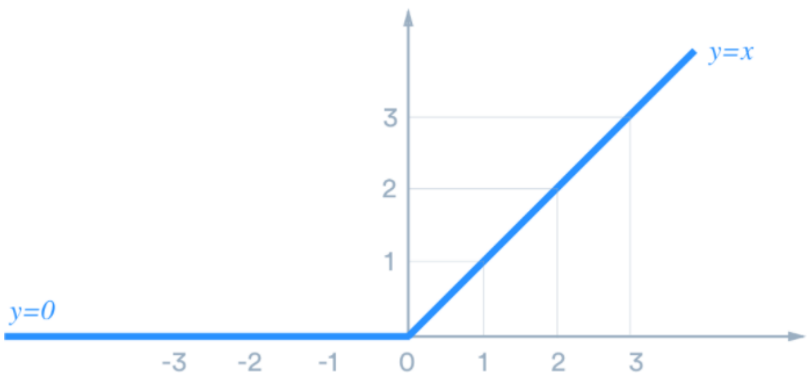
(1) tanh

值介於 -1 到 1 之間，因此隱藏層的平均值為 0 或非常接近它，因此通過使平均值接近 0 有助於使數據居中。這使得學習下一層更容易

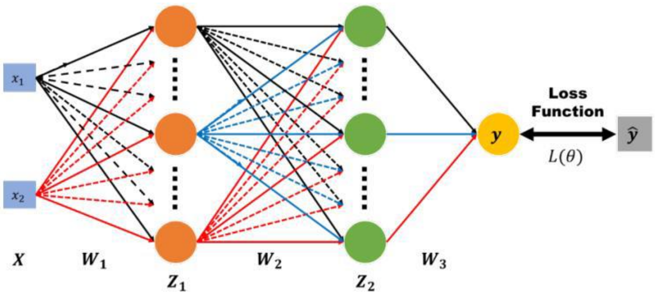


(2) ReLu

計算成本低於 tanh 和 sigmoid，因為它涉及更簡單的數學運算。一次只有少數神經元被激活，使網絡變得稀疏，從而使其高效且易於計算。



B. Neural network



神經系統的架構如上圖x(Input Layer) 就是接收信號的神經元，Z1, Z2(Hidden Layer) 就是隱藏層，而y(Output Layer) 就是做出反應的輸出層，而各神經元傳導的力量大小，稱為權重(Weight, 以W表示)，也就是模型要求解的參數，如果求算出來，我們就得到一道公式，只要輸入信號，經過層層傳導，推斷出結果

整個Neural network分成三個架構，initial，train，test(predict)

```
class Model :
    def __init__(self,input_size, hidden_size, output_size, learning_rate, epochs,print_interval,activa
        #defult  original parameters

        self.input_size = input_size
        self.hidden_size = hidden_size        # the number of hidden neurons used in this model.
        self.output_size=output_size
        self.learning_rate=learning_rate
        self.epochs=epochs                    # the total number of training steps.
        self.print_interval=print_interval    # the number of steps between each reported number
        self.activation = activation          # the activation type to use
        self.layers = []
        self.activations = []

        size = [input_size, hidden_size , hidden_size, output_size]

        #init each layer
        for input_nn, output_nn in list(zip(size[:-1], size[1:])):
            self.layers += [Layer(input_size=input_nn, output_size=output_nn , learning_rate = self.lea
            self.activations.append(self.activation)
```

其中Class Model 透過 Class Layer 實作各層並且定義(derivative)loss function(這邊使用MSE)

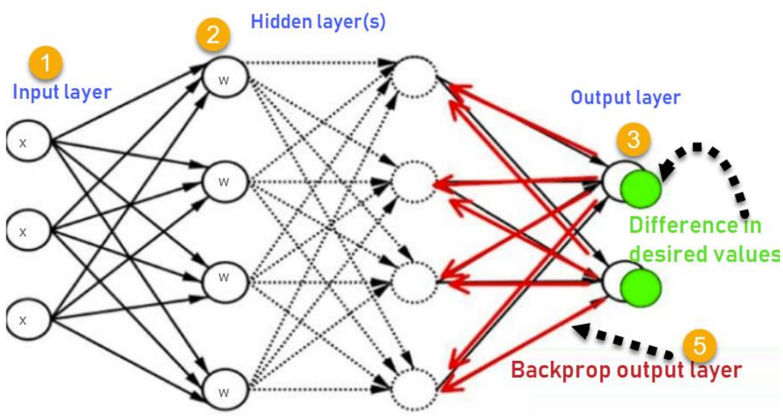
```
class Layer :
    def __init__(self , input_size , output_size,learning_rate = 0.1 , optimizer ="default") :

        # init the weights by random
        self.weights = np.random.normal(0, 1, (input_size+1, output_size))
        self.m_t = np.zeros((input_size + 1, output_size))
        self.v_t = np.zeros((input_size + 1, output_size))
        self.learning_rate = learning_rate
        self.optimizer = optimizer
        self.updatecount = 0
```

layer裡面都有定義了(derivative)activation function 及 初始化weights 且另外多加一層 bias

```python
if __name__ == '__main__':
    input_size = 2
    hidden_size = 4
    output_size = 1
    learning_rate = 0.9
    epochs = 100000
    print_interval = 5000

    optimizer = "adam"
    activation ="sigmoid"
    print("=== Linear ===")
    data, label = Data.fetch_data('Linear', 100)
    model = Model(input_size, hidden_size, output_size, learning_rate, epochs,print_interval,activatior
    model.train(data, label)

    pred_result = np.round(model.forward(data))
    print('Linear predictions:\n{}'.format(model.forward(data)))
    model.plot_result(data, label, pred_result)
```

最後透過主程式來實作 nn model

C. Backpropagation



(1)初始化神經網路所有權重

(2)將資料由input layer往output layer向前傳遞(forward pass)，
並計算出所有神經元的output

(3)誤差由output layer往input layer向後傳遞(backward propagation)，
並算出每個神經元對誤差的影響

(4)用誤差影響去更新權重(weights)

(5)重複步驟(2)~(4)直到誤差收斂夠小

```python
def backward(self,outputs):
    tmpoutput = outputs
    for layer,act in  zip(self.layers[::-1],self.activations):
        tmpoutput = layer.backward(tmpoutput,act )
```

上圖為 nn 的 backpropagation，他會從 output layer 開始往 input layer 回推，並且將計算出的 loss $L = (\frac{1}{2})(y - \hat{y})^2$ 送給上一層

```python
def backward(self, derivative, activation="sigmoid"):
    #  Calculate the back gradient by multiply gradient and weights

    self.backward_gradient = np.multiply(self.de_activation_function(self.y,activation), derivative
    return np.matmul(self.backward_gradient, self.weights[:-1].T)
```

每層layer再透過所用的 activation function 算出 backward gradient(權重對於Loss的影響程度）並且 return 這層的 loss。

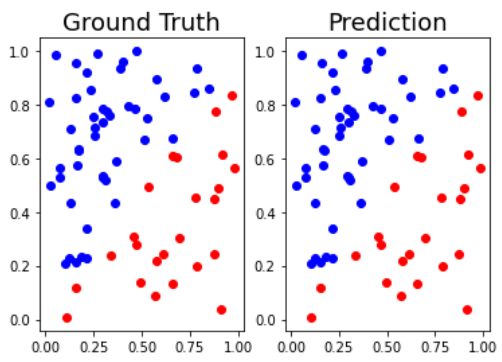
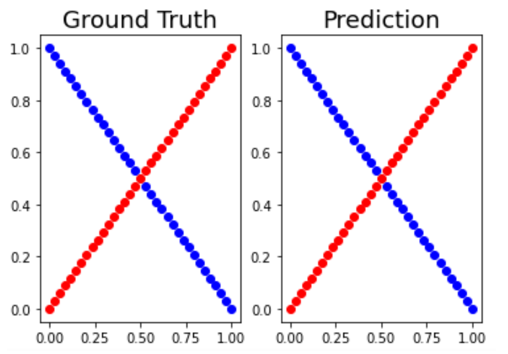
```
def update(self):
    """
    update the weight
    default : by multiplication the learning rate
    """
    #calculates the gradient by multiplication output of the second hidden layer and backpropagatec
    self.gradient = np.matmul(self.forward_gradient.T, self.backward_gradient)

        delta_weight = -(self.learning_rate*self.gradient)
    self.weights += delta_weight
    return self.gradient
```

最夠再去更新每層的weight，每次只跨出learning rate(n) 的數值，反覆計算參數weight的gradient並重複更新weight，更新的公式為 $W_{ij}^{(l)} \leftarrow W_{ij}^{(l)} - \eta \times \frac{\partial L}{\partial w_{ij}^{(l)}}$

三、 Results of your testing

A. Screenshot and comparison figure

經過反覆的實驗 Sigmoid + Adam optimize + learning rate = 0.1 達到最好的結果，除了準確率100%之外也可在最短的epoch到達100%

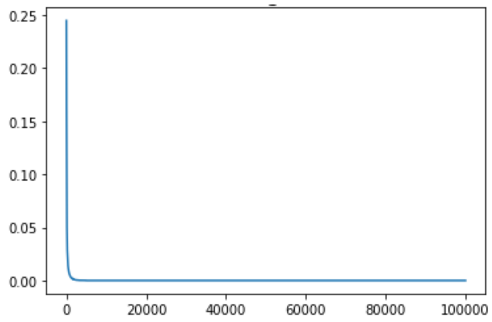
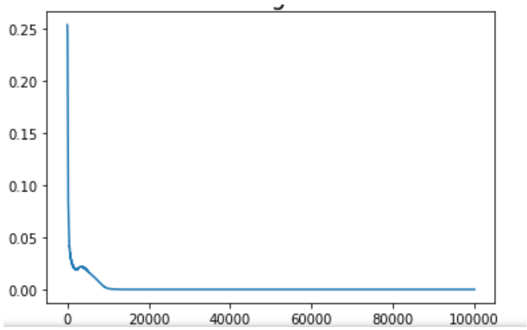
Linear	XOR
	

B. Show the accuracy of your prediction

network 對於 linear 這個比較簡單的問題會給出較接近 1 或 0 的機率。

Linear	XOR
Training finished total time:31.47303342819214 accuracy: 100.00% Linear predictions: [[0.0000e+00] [0.0000e+00] [0.0000e+00] [1.0000e+00] [1.0000e+00] [1.0000e+00] [0.0000e+00] [0.0000e+00] [1.0000e+00] Epoch 15000 loss : 0.0000 accuracy: 99.99% Epoch 20000 loss : 0.0000 accuracy: 100.00%	Training finished total time:31.51274037361145 accuracy: 100.00% XOR predictions: [[0.0000e+00] [1.0000e+00] [0.0000e+00] [1.0000e+00] [0.0000e+00] [1.0000e+00] [0.0000e+00] [1.0000e+00] [0.0000e+00] Epoch 25000 loss : 0.0000 accuracy: 99.99% Epoch 30000 loss : 0.0000 accuracy: 100.00%

C. Learning curve (loss, epoch curve)

Linear	XOR
	

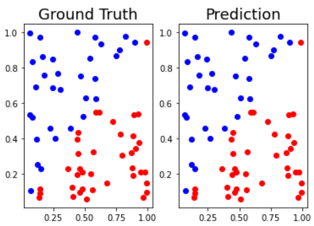
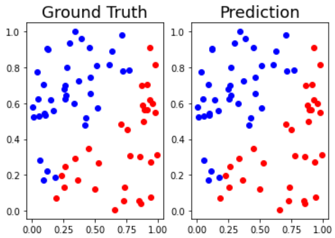
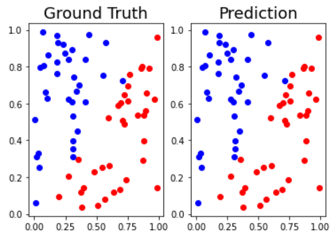
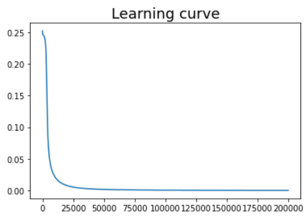
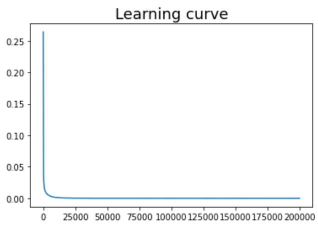
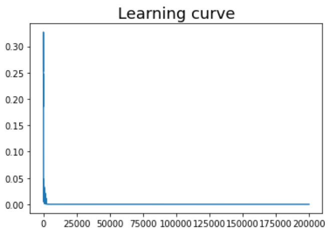
四、 Results of your testing

因為這次作業sigmoid function 所以大部分都使用sigmoid function作為default

A. Try different learning rates

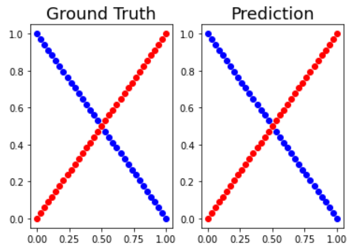
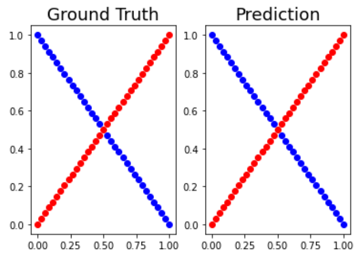
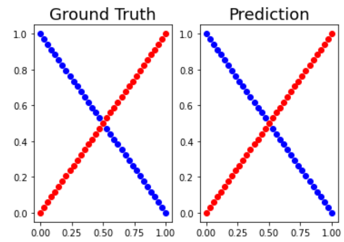
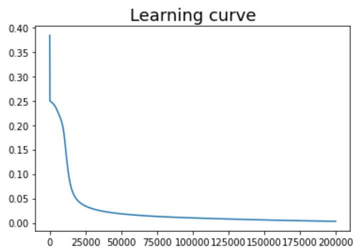
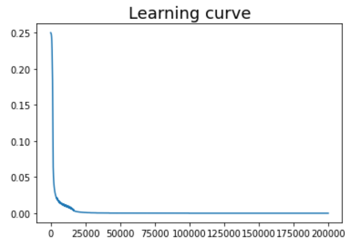
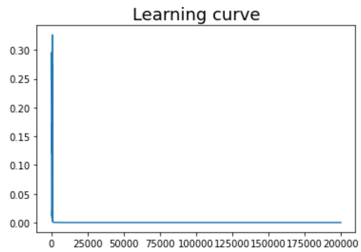
With default sigmoid + 4 unit

(1) Linear

learning rates 0.1	learning rates 1	learning rates 10
 total time:53.36312532424927 accuracy: 99.58% Linear predictions: [[3.0000e-05] [4.0000e-05] [9.9998e-01] [9.9999e-01] [9.9999e-01] [2.7430e-02] [8.9000e-03]	 total time:54.32226014137268 accuracy: 99.92% Linear predictions: [[1.0000e+00] [9.9999e-01] [1.3000e-03] [0.0000e+00] [1.0000e+00] [1.0000e+00] [9.9008e-01]	 total time:54.467650413513184 accuracy: 99.98% Linear predictions: [[4.0000e-05] [4.0000e-05] [9.9999e-01] [9.9999e-01] [4.0000e-05] [4.0000e-05] [5.0000e-05]
		

由上列結果發現我的model在learning rate = 10 有最好結果，但是這不包含任何優化以及其他activation function但在learning rate=1時候訓練初期learning curve 下向平滑且快，learning rate=10的時候前期反而動盪較大。

(2) XOR

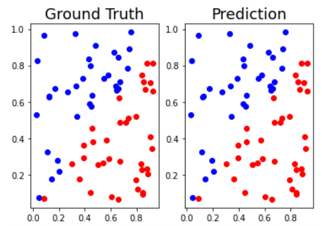
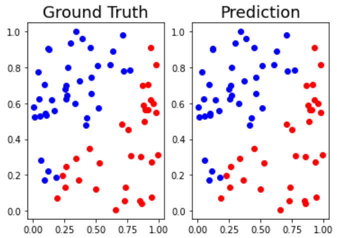
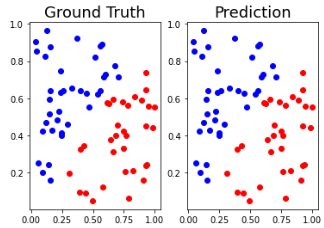
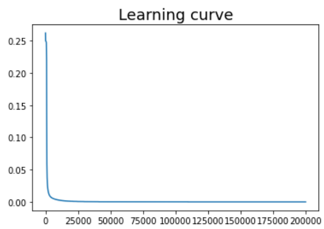
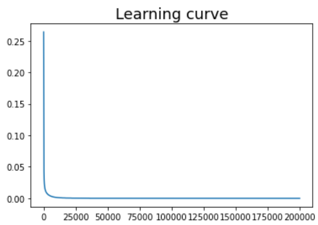
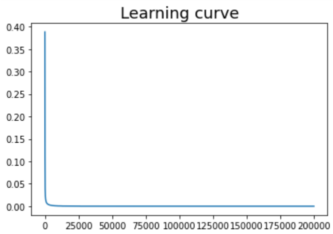
learning rates 0.1	learning rates 1	learning rates 10
 total time:53.72257137298584 accuracy: 98.44% XOR predictions: [[1.2380e-02] [1.0000e+00] [1.6800e-03] [1.0000e+00] [2.2000e-04] [1.0000e+00] [3.0000e-05] [1.0000e+00] [0.0000e+00]	 total time:54.36573028564453 accuracy: 99.88% XOR predictions: [[1.4100e-03] [1.0000e+00] [8.0000e-05] [1.0000e+00] [1.0000e-05] [1.0000e+00] [0.0000e+00] [1.0000e+00]	 total time:54.517961740493774 accuracy: 99.93% XOR predictions: [[2.0000e-04] [9.9956e-01] [1.9000e-04] [9.9956e-01] [1.8000e-04] [9.9956e-01] [1.8000e-04]
 	 	

由上列結果發現我的model在learning rate = 10 有最好結果，但是這不包含任何優化以及其他activation function 但在100時會無法收斂可能不太好細微的調整weight。

B. Try different numbers of hidden units

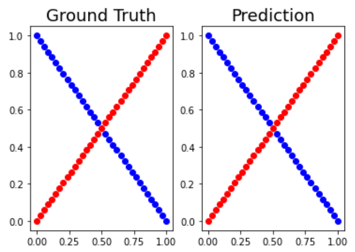
With default sigmoid + learning rates = 1

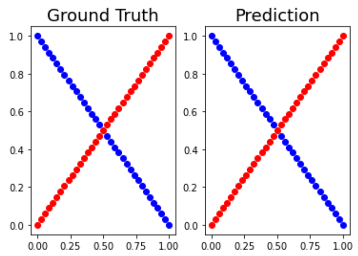
(1) Linear

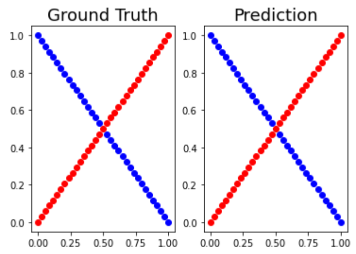
2 units	4 units	10 units
 total time:52.700193643569946 accuracy: 99.90% Linear predictions: [[0.0000e+00] [0.0000e+00] [9.9999e-01] [0.0000e+00] [9.8160e-01] [9.9998e-01] [0.0000e+00] [9.9999e-01]	 total time:54.32226014137268 accuracy: 99.92% Linear predictions: [[1.0000e+00] [9.9999e-01] [1.3000e-03] [0.0000e+00] [1.0000e+00] [1.0000e+00] [9.9008e-01]	 total time:61.84667158126831 accuracy: 99.93% Linear predictions: [[1.7990e-02] [1.0000e-05] [1.6000e-04] [1.0000e+00] [9.9999e-01] [1.0000e+00] [1.0000e-05]
		

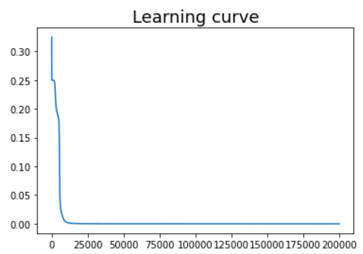
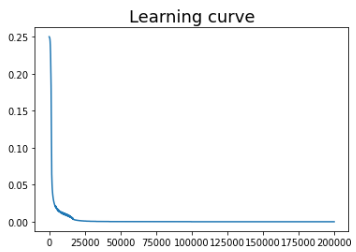
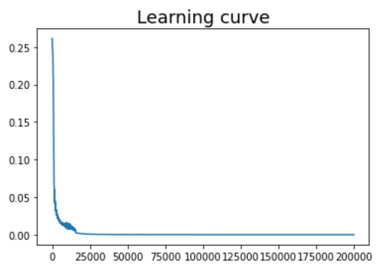
從上面的結果發現隨著增加hidden layer unit 數量可以發現時間變長之外，同樣epoch下可以有準確率些微進步，且loss下降速度也明顯提升很多。

(2) XOR

<table>
<tr><th>2 units</th><th>4 units</th><th>10 units</th></tr>
<tr>
<td>
total time:52.64521503448486
accuracy: 99.72%

XOR predictions:
[[0.00353]
[0.99863]
[0.00354]
[0.99863]
[0.00355]
[0.99863]
[0.00355]</td>
<td>
total time:54.36573028564453
accuracy: 99.88%

XOR predictions:
[[1.4100e-03]
[1.0000e+00]
[8.0000e-05]
[1.0000e+00]
[1.0000e-05]
[1.0000e+00]
[0.0000e+00]
[1.0000e+00]</td>
<td>
total time:61.48851037025452
accuracy: 99.88%

XOR predictions:
[[0.0000e+00]
[1.0000e+00]
[0.0000e+00]
[1.0000e+00]
[0.0000e+00]
[1.0000e+00]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</table>

同 unit 的收斂時間看起越大越快收斂可是較多的unit因為學習力較強learning curve 變得較為動盪。

C. Try without activation functions

Without the activation functions 沒辦法再XOR取得好的結果，因為沒有activation function 沒辦法有效的處理non linear的data也沒有找到比姣好的優方式

D. Anything you want to share

之前只有使用過一些套件來做NN，這次自己手刻NN可以更了解整個運行的流程，且透過不同方式改變NN內的參數再透過learning curve去推斷為何可能造成這樣的結果，但即使同樣的參數跑多次過程中還是有很多不一樣的learning curve，有時候有無法自己解釋為什麼這樣的改變會引響整個NN。此外也在網路上發現output layer若設定為sigmoid function 中間再透過其他activation function 來做NN可以有很好的結果，可以透過sigmoid將結果控制在0~1之間。

五、 Extra

A. Implement different optimize

Momentum Optimizer

此優化器為模擬物理動量的概念，在同方向的維度上學習速度會變快，方向改變的時候學習速度會變慢。

Adam optimizer (adaptive moment estimation):

Adam 算法和傳統的隨機梯度下降不同。隨機梯度下降保持單一的學習率（即 alpha）更新所有的權重，學習率在訓練過程中並不會改變。而 Adam 通過計算梯度的一階矩估計和二階矩估計而為不同的參數設計獨立的自適應性學習率。

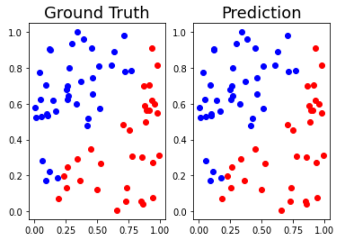
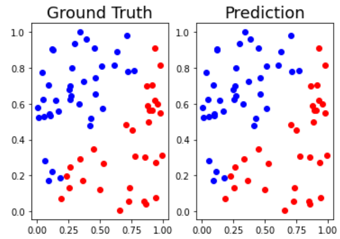
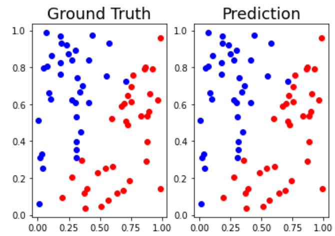
Adam 算法的提出者描述其為兩種隨機梯度下降擴展式的優點集合，即：

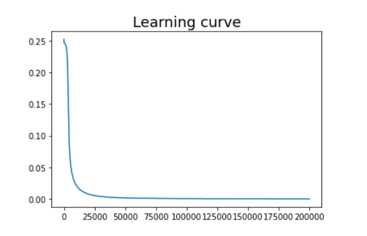
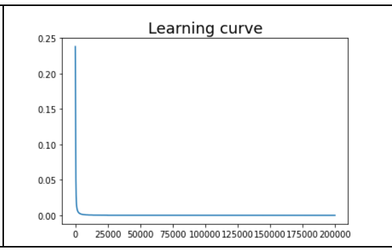
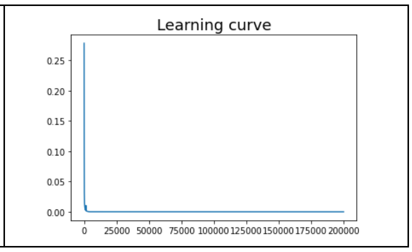
適應性梯度算法（AdaGrad）為每一個參數保留一個學習率以提升在稀疏梯度（即自然語言和計算機視覺問題）上的性能。

均方根傳播（RMSProp）基於權重梯度最近量級的均值為每一個參數適應性地保留學習率。這意味著算法在非穩態和在線問題上有很有優秀的性能。

Reference : https://github.com/sagarvegad/Adam-optimizer

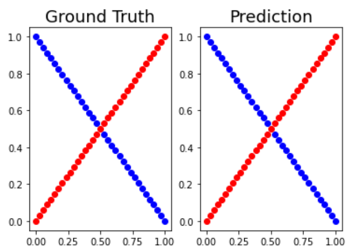
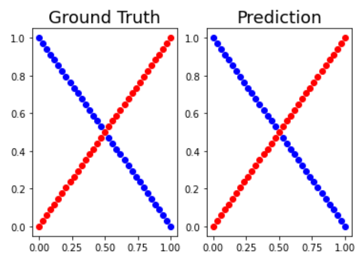
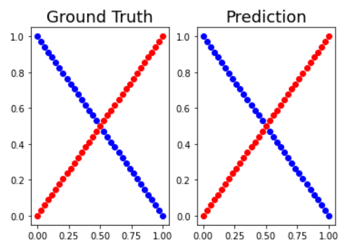
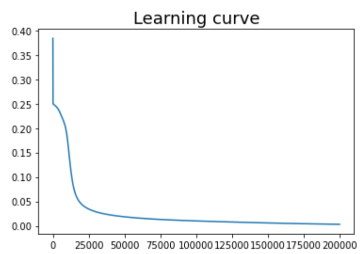
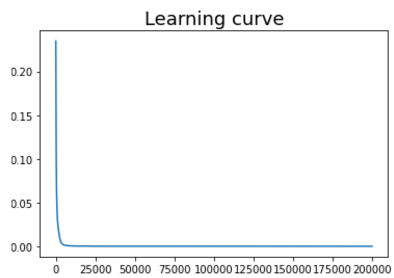
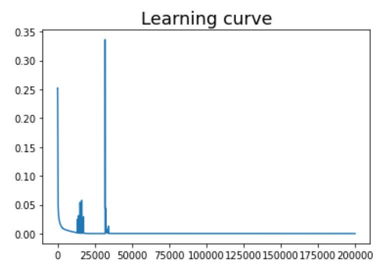
(1)Linear

Gradient descent(default)	Momentum	Adam
$W \leftarrow W - \eta \frac{\partial L}{\partial W}$	$V_t \leftarrow \beta V_{t-1} - \eta \frac{\partial L}{\partial W}$ $W \leftarrow W + V_t$	$f(x, y) = 0.5x^2 + 2.5y^2$
 total time:53.36312532424927 accuracy: 99.58% Linear predictions: [[3.0000e-05] [4.0000e-05] [9.9998e-01] [9.9999e-01] [9.9999e-01] [2.7430e-02] [8.9000e-03]	 total time:53.13893413543701 accuracy: 99.90% Linear predictions: [[1.0000e-05] [2.1200e-03] [6.0000e-05] [1.0000e-05] [1.0000e-05]	 total time:62.73565149307251 accuracy: 100.00% Linear predictions: [[1.0000e+00] [1.0000e+00] [0.0000e+00] [1.0000e+00] [1.0000e+00] [1.0000e+00] [1.0000e+00] [1.0000e+00]

這次的實驗可以發現有用其他optimaiz來調整learning rate除了loss下降得更快之外，準確率都明顯提升許多

(2) XOR

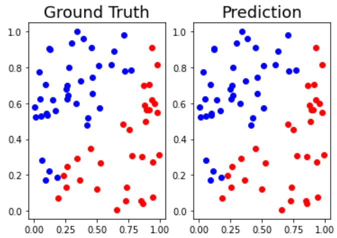
Gradient descent(default) $W \leftarrow W - \eta \frac{\partial L}{\partial W}$	Momentum $V_t \leftarrow \beta V_{t-1} - \eta \frac{\partial L}{\partial W}$ $W \leftarrow W + V_t$	Adam $f(x, y) = 0.5x^2 + 2.5y^2$
 total time:53.72257137298584 accuracy: 98.44% XOR predictions: [[1.2380e-02] [1.0000e+00] [1.6800e-03] [1.0000e+00] [2.2000e-04] [1.0000e+00] [3.0000e-05] [1.0000e+00] [0.0000e+00]	 total time:53.01536154747009 accuracy: 99.85% XOR predictions: [[4.0000e-05] [9.9737e-01] [4.0000e-05] [9.9736e-01] [4.0000e-05] [9.9736e-01]	 total time:63.4729528427124 accuracy: 100.00% XOR predictions: [[0.] [1.] [0.] [1.] [0.]
		

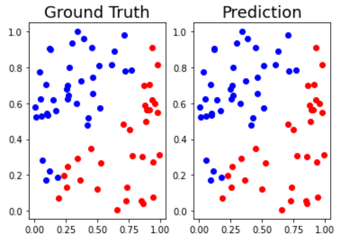
這次的實驗跟上面linear一樣準確率都明顯提升，但不太知道是甚麼原因造成Adam比較動盪，而且透過其他optimize可以使用比較高的learning rate因為也會在訓練中調整。

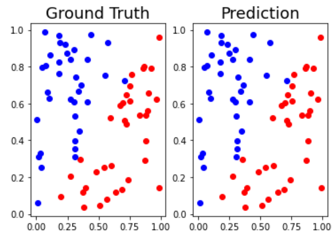
B. Implement different activation function.

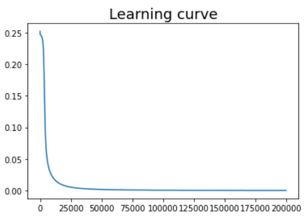
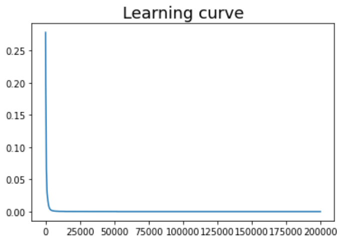
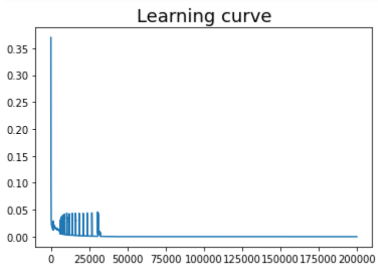
With default slearningrate=0.1 + 4 unit

(1) Linear

<table>
<tr><th>sigmoid</th><th>tanh</th><th>relu</th></tr>
<tr>
<td>
total time:53.36312532424927
accuracy: 99.58%

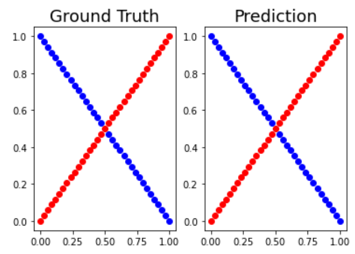
Linear predictions:
[[3.0000e-05]
 [4.0000e-05]
 [9.9998e-01]
 [9.9999e-01]
 [9.9999e-01]
 [2.7430e-02]
 [8.9000e-03]</td>
<td>
total time:51.82054591178894
accuracy: 99.84%

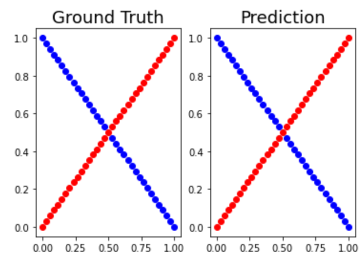
XOR predictions:
[[0.0000e+00]
 [9.9773e-01]
 [0.0000e+00]
 [9.9769e-01]
 [0.0000e+00]
 [9.9765e-01]
 [0.0000e+00]</td>
<td>
total time:53.153714418411255
accuracy: 99.99%

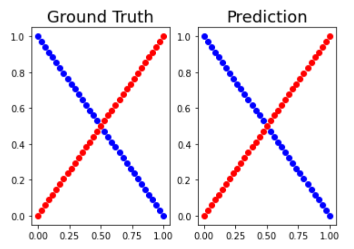
Linear predictions:
[[0.   ]
 [1.   ]
 [1.   ]
 [0.   ]
 [1.   ]
 [1.   ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</table>

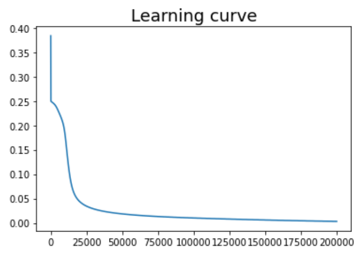
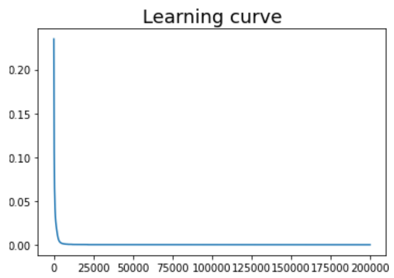
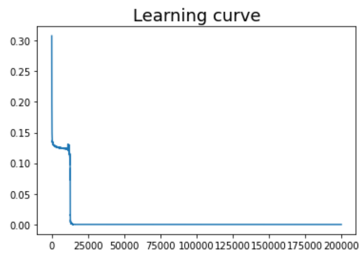
發現這次的實驗relu可以達到最好的acc 但是相對動盪也比較大，需要比較多的epoch來收斂，且兩個都有明顯進步

(2) XOR

<table>
<tr><th>sigmoid</th><th>tanh</th><th>relu</th></tr>
<tr>
<td>
total time:53.72257137298584
accuracy: 98.44%

XOR predictions:
[[1.2380e-02]
[1.0000e+00]
[1.6800e-03]
[1.0000e+00]
[2.2000e-04]
[1.0000e+00]
[3.0000e-05]
[1.0000e+00]
[0.0000e+00]</td>
<td>
total time:53.01536154747009
accuracy: 99.85%

XOR predictions:
[[4.0000e-05]
[9.9737e-01]
[4.0000e-05]
[9.9736e-01]
[4.0000e-05]
[9.9736e-01]</td>
<td>
total time:53.57631278038025
accuracy: 99.99%

XOR predictions:
[[3.0000e-05]
[9.9988e-01]
[2.0000e-05]
[9.9988e-01]
[2.0000e-05]
[9.9988e-01]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</table>

由上面兩個表格我覺得 tanh 並沒有達到比較好的結果，但是反而 learning rate 都較為平滑，反而是 relu 動盪較大但結果都有明顯進步。