Lab 1 report

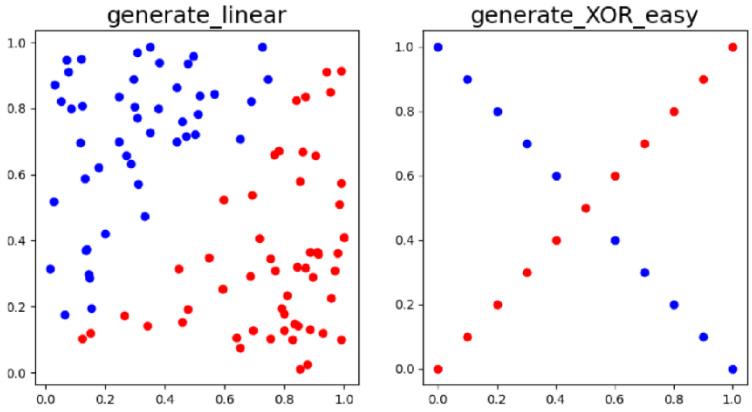
1. Introduction

這次 lab 要實作一個有兩個中間層的神經網路，透過 Backpropagation 的 forward pass 來得到預測的答案，並且利用 backward pass 來算出 gradient，再利用 optimizer 來更新 weight，讓預測的答案更準確，loss 降更低。

Dataset 是由以下的函式產生的。

```
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)

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)
```

下圖為兩個 dataset 中的每個點



以下為 Lab Requirements

1. Implement simple neural networks with two hidden layers.
2. Each hidden layer needs to contain at least one transformation (CNN, Linear …)

and one activate function (Sigmoid, tanh….).
3. You must use backpropagation in this neural network and can only use Numpy and other python standard libraries to implement.
4. Plot your comparison figure that shows the predicted results and the ground-truth.
5. Print the training loss and testing result

2. Experiment setups

A. Sigmoid functions

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

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

```
def sigmoid(a, derivative=False):
    if not derivative:
        return 1.0 / (1.0 + np.exp(-a))
    else:
        return np.multiply(a, 1.0 - a)
```

我將具有微分和無微分兩種形式的 sigmoid function 編寫成同一函式，其預設設定為無微分形式，但若需要微分，則設定 derivative 為 True。在上述程式碼中，np.multiply(a, 1.0 - a)這一段，依照 sigmoid 的微分公式，此處的 a 應為 sigmoid(a)，但由於在我的 Backpropagation 程式碼中，所有需要用到微分的 activation function 的輸入 a 都已經過一次 activation function 的處理，因此已經調整為此方式。無微分的 sigmoid function 用於神經網路的每一層的 activation function，而具有微分的 sigmoid function 則用於 backward pass 的計算。

B. Neural network

(1) Layer:

```python
#定義全連接層
class FCLayer:
    def __init__(self, input_nodes, output_nodes, activation = "sigmoid"):
        self.weight = np.random.normal(0, 1, (input_nodes, output_nodes))
        self.activation = activation
        self.F = np.zeros((output_nodes, 1)) #input和weight相乘後的矩陣
        self.Z = np.zeros((output_nodes, 1)) #F經過activation後的矩陣
        self.dC_dF = np.zeros((output_nodes, 1)) #C是loss function
        self.gradient = np.zeros((output_nodes, input_nodes))
        self.movement = np.zeros((input_nodes, output_nodes))
        self.movement_hat = np.zeros((input_nodes, output_nodes))
        self.v = np.zeros((input_nodes, output_nodes))
        self.v_hat = np.zeros((input_nodes, output_nodes))
        self.t = 1
        self.sum_square_gradient = np.zeros((output_nodes, input_nodes))
        if activation == "sigmoid":
            self.activation = sigmoid
        elif activation == "relu":
            self.activation = relu
        elif activation == "tanh":
            self.activation = tanh
        else:
            self.activation = no_activation
```

上圖為 class FCLayer，我利用它去定義神經網路的每一個全連接層。以下解釋上圖的各項參數意義。

input_nodes: 輸入的維度(節點數)
output_nodes: 輸出的維度(節點數)
activation: activation function
self.weight: 層與層間的權重矩陣
self.F : input 和 weight 相乘後的矩陣
self.Z : F 經過 activation function 後的矩陣
self.dC_df: loss function 對 F 做偏微分後的結果
self.gradient: loss function 對每一個 weight 做偏微分後的結果
self.movement: optimizer 用 momentum 時要用的參數
self.movement_hat: optimizer 用 adam 時要用的參數
self.v: optimizer 用 adam 時要用的參數
self.v_hat: optimizer 用 adam 時要用的參數
self.t: optimizer 用 adam 時要用的參數，作為 weight 更新次數
self.sum_square_gradient: optimizer 用 adagrad 時要用的參數

(2) NeuralNet:

```
class NeuralNet:
    def __init__(self, input_dim = 2, hidden1_dim = 3, hidden2_dim = 3, output_dim = 1
                 , num_weightlayers = 3, activation = 'sigmoid', learning_rate = 0.01, optimizer = 'gd'):
        self.num_weightlayers = num_weightlayers
        self.learning_rate = learning_rate
        self.optimizer = optimizer
        self.activation = activation

        #設定input層
        self.layers = [FCLayer(input_dim, hidden1_dim, activation)]

        #設定中間層
        self.layers.append(FCLayer(hidden1_dim, hidden2_dim, activation))

        #設定output層
        self.layers.append(FCLayer(hidden2_dim, output_dim, activation))

    def forward_pass(self, x):...

    def backward_pass(self, y, y_hat):...

    def compute_grad(self, x):...
    def optimize(self):...
```

上圖為 class NeuralNet 我利用它去定義整個神經網路，這個 class 也包含了 Backpropagation 的過程、每一層神經網路的維度、learning rate 為多少、optimizer 跟 activation function 要用什麼。以下解釋上圖參數意義。

input_dim: 輸入層的維度，預設 2

hidden1_dim: 第一個中間層的維度，預設 3

hidden2_dim: 第二個中間層的維度，預設 3

output_dim: 輸出層的維度，預設 1

self.layers: 引用 class FCLayer 建立每一層神經網路

activation: activation function 預設為 sigmoid

learning_rate: learning rate 預設為 0.01

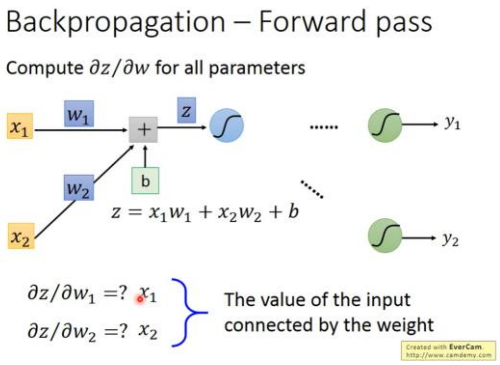
optimizer: optimizer 預設為 gradient descent

C. Backpropagation

(1) Forward pass

```
def forward_pass(self, x):
    z = x
    for i in range(self.num_weightlayers):
        f = np.matmul(z, self.layers[i].weight)
        self.layers[i].F = f
        z = self.layers[i].activation(f)
        self.layers[i].Z = z
    return z
```

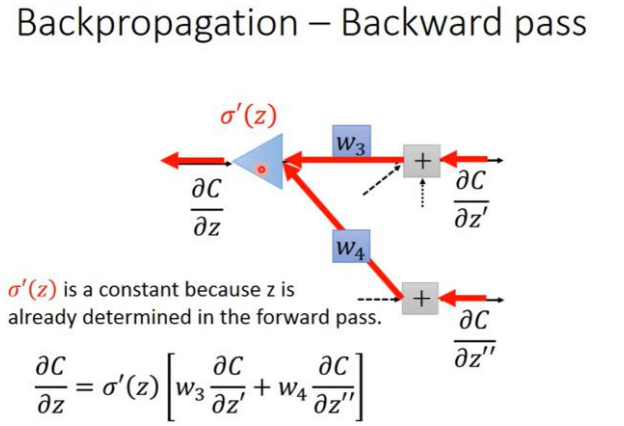
我們可以透過上圖的 forward pass 得到我們神經網路的預測值，也可以從下圖知道每層神經網路的 input 就是我們要的 df/dw。下圖的 z 就是我上圖程式的 f



(2) Backward pass

```
def backward_pass(self, y, y_hat):
    self.layers[2].dC_dF = mse(y, y_hat, derivative=True) * self.layers[2].activation(self.layers[2].Z, derivative=True)
    for i in range(1, -1, -1):
        self.layers[i].dC_dF = self.layers[i].activation(self.layers[i].Z, derivative = True) * np.matmul(self.layers[i+1].weight, self.layers[i+1].dC_dF )
```

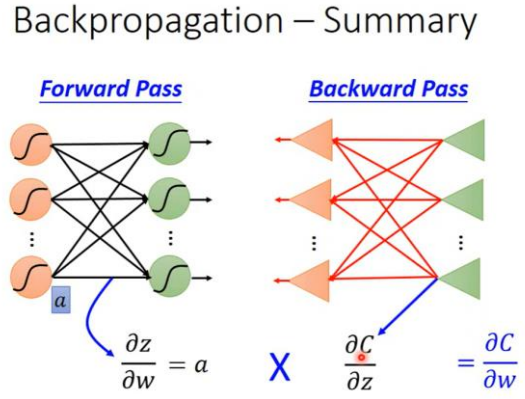
將下圖的 z 換成 f，就是上圖程式碼的運算。



(3) Compute Gradient

```
def compute_grad(self, x):
    self.layers[0].gradient = x * self.layers[0].dC_dF.reshape(-1, 1)
    for i in range(1, 3, 1):
        self.layers[i].gradient = self.layers[i-1].Z * self.layers[i].dC_dF.reshape(-1, 1)
```

將 Forward pass 乘以 Backward pass 就可以得到我們的 Gradient 了，如下圖算式，將 z 改成 f、a 改成 z，就是上圖程式碼的算式。



(4) Weight update

```
def optimize(self):
    if self.optimizer == 'gd':
        for i in range(3):
            self.layers[i].weight = self.layers[i].weight + (-self.learning_rate * self.layers[i].gradient.T)
    elif self.optimizer == 'momentum':
        for i in range(3):
            self.layers[i].movement = (0.9 * self.layers[i].movement) +  (-self.learning_rate * self.layers[i].gradient.T)
            self.layers[i].weight = self.layers[i].weight + self.layers[i].movement
    elif self.optimizer == 'adagrad':
        for i in range(3):
            self.layers[i].sum_square_gradient += np.square(self.layers[i].gradient)
            self.layers[i].weight = self.layers[i].weight + ((-self.learning_rate * self.layers[i].gradient.T) / (np.sqrt(self.layers[i].sum_square_gradient).T +1e-8))
    elif self.optimizer == 'adam':
        for i in range(3):
            self.layers[i].movement = (0.9 * self.layers[i].movement) +  (0.1 * self.layers[i].gradient.T)
            self.layers[i].movement_hat = self.layers[i].movement / (1 - (0.9 ** self.layers[i].t))
            if self.layers[i].t > 1 :
                self.layers[i].v = 0.999 * self.layers[i].v + 0.001 * np.square(self.layers[i].gradient).T
            else :
                self.layers[i].v = np.square(self.layers[i].gradient).T
            self.layers[i].v_hat = self.layers[i].v / (1 - (0.999 ** self.layers[i].t))
            self.layers[i].weight = self.layers[i].weight - (self.learning_rate * self.layers[i].movement_hat / (np.sqrt(self.layers[i].v_hat) + 1e-8))
```

Gradient Decent 公式:

$$W = W - \eta \frac{\partial L}{\partial W}$$

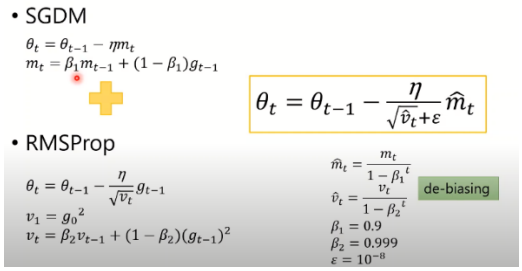
Momentum 公式:

$$\theta_t = \theta_{t-1} - \eta m_t$$
$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_{t-1}$$

Adagrad 公式:

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\sum_{i=0}^{t-1}(g_i)^2}} g_{t-1}$$

Adam 公式:



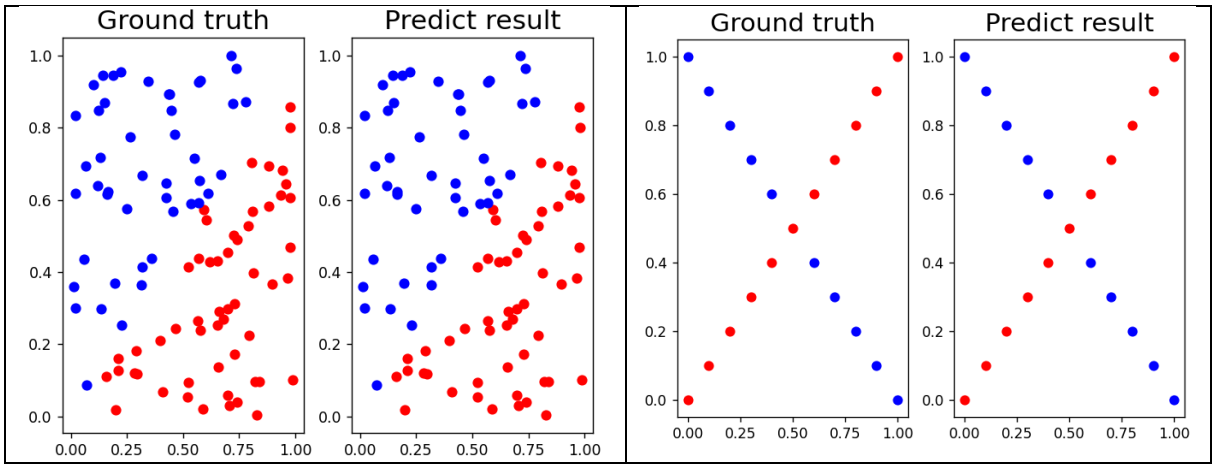
以上為各 Optimizer 的程式碼及公式

3. Results of your testing

這個 testing 的參數都是用 class NuralNet 預設的參數，且 epoch 設 10000、loss function 用 mean-square error。

A. Screenshot and comparison figure

Linear	XOR

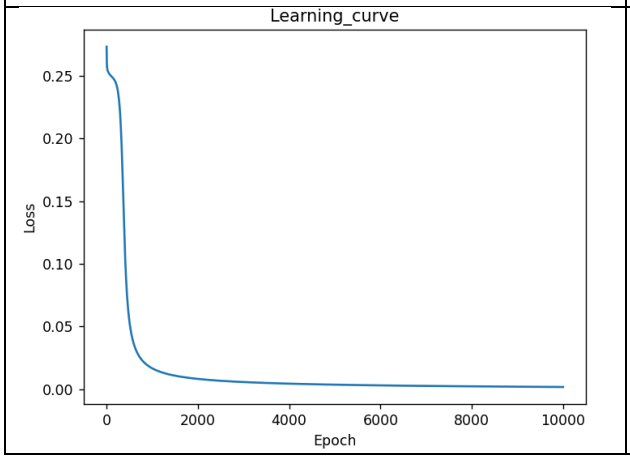
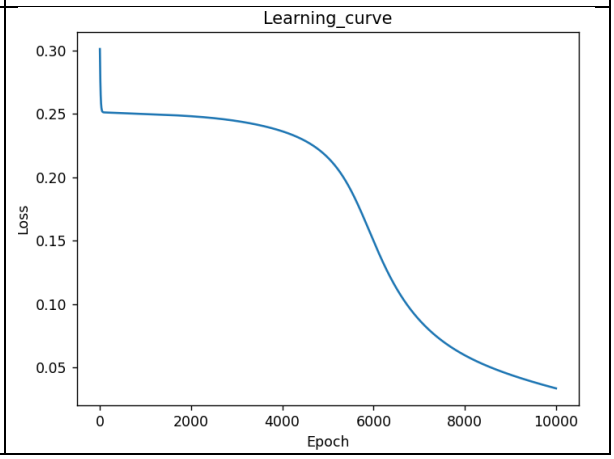


從上表格可以看出 Linear 跟 XOR 的 Predict result 跟 Ground truth 皆是一樣的。

B. Show the accuracy of your prediction

Linear	XOR
Iter95 | Ground truth:0 | prediction:0.00032682761138472504 Iter96 | Ground truth:0 | prediction:0.0005986861758303693 Iter97 | Ground truth:1 | prediction:0.9999880249878399 Iter98 | Ground truth:0 | prediction:0.00043422947213679305 Iter99 | Ground truth:1 | prediction:0.967005743544969 Iter100 | Ground truth:1 | prediction:0.999984202581091 loss=0.0017209659694868357 accuracy=100.0%	Iter1 | Ground truth:0 | prediction:0.024783259907277755 Iter2 | Ground truth:1 | prediction:0.9860438566169168 Iter3 | Ground truth:0 | prediction:0.07009391679063046 Iter4 | Ground truth:1 | prediction:0.983456823729089 Iter5 | Ground truth:0 | prediction:0.1607859911407427 Iter6 | Ground truth:1 | prediction:0.9741054203809885 Iter7 | Ground truth:0 | prediction:0.25284521202607785 Iter8 | Ground truth:1 | prediction:0.9200826754444495 Iter9 | Ground truth:0 | prediction:0.2854480809012625 Iter10 | Ground truth:1 | prediction:0.5639779902821647 Iter11 | Ground truth:0 | prediction:0.25352184934299776 Iter12 | Ground truth:0 | prediction:0.18860521385737386 Iter13 | Ground truth:1 | prediction:0.5746273950617826 Iter14 | Ground truth:0 | prediction:0.12466302096737072 Iter15 | Ground truth:1 | prediction:0.9259102790646778 Iter16 | Ground truth:0 | prediction:0.0782114090048488 Iter17 | Ground truth:1 | prediction:0.9732972261752292 Iter18 | Ground truth:0 | prediction:0.04931155233737895 Iter19 | Ground truth:1 | prediction:0.980866660365115 Iter20 | Ground truth:0 | prediction:0.032376963707094475 Iter21 | Ground truth:1 | prediction:0.9824688233222004 loss=0.03272535293466081 accuracy=100.0%

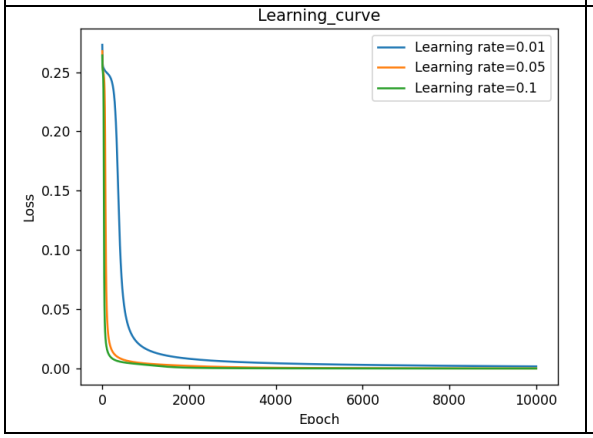
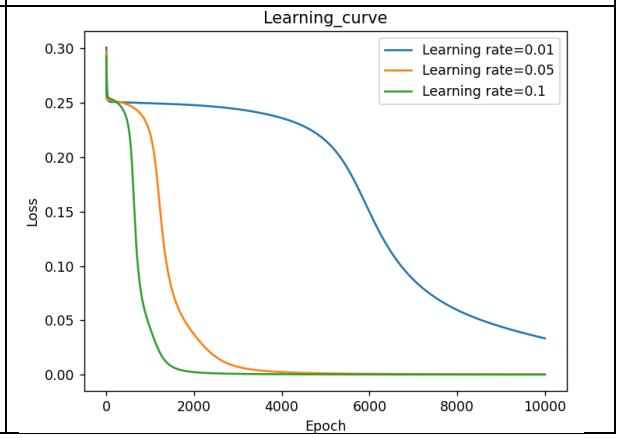
C. Learning curve (loss, epoch curve)

Linear	XOR
	

4. Discussion

A. Try different learning rates

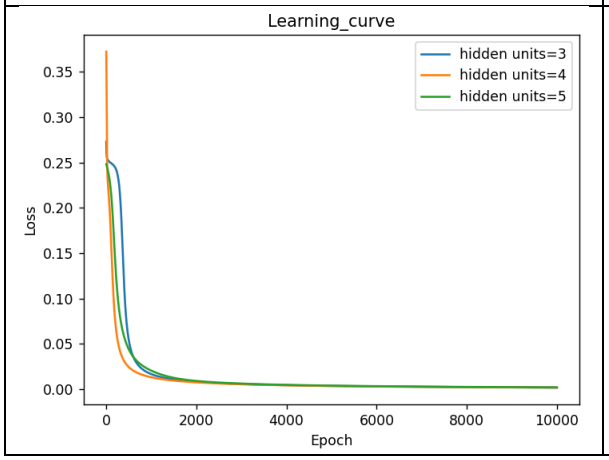
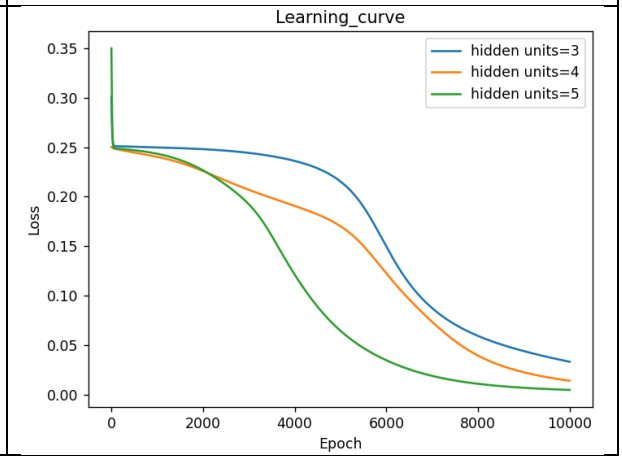
這個實驗的參數，除了 learning rate 外，其他都是用 class NuralNet 預設的參數，且 epoch 設 10000、loss function 用 mean-square error。

Linear	XOR
	

實驗顯示不管是 0.01、0.05 還是 0.1 的 learning rates，對 Linear dataset、XOR dataset 作用的 accuracy 皆 100%。learning rates 越高對這兩個 dataset 來說，loss 會更快逼近 0。

B. Try different numbers of hidden units

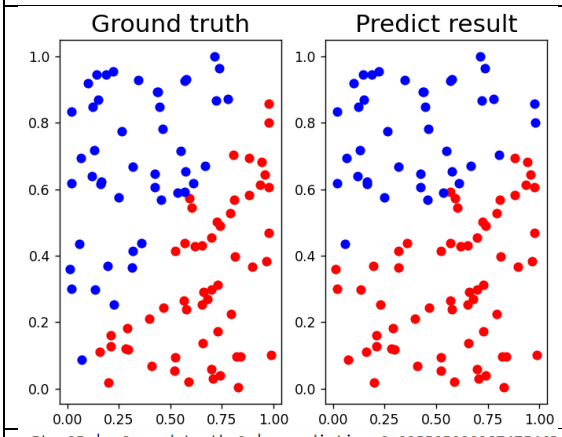
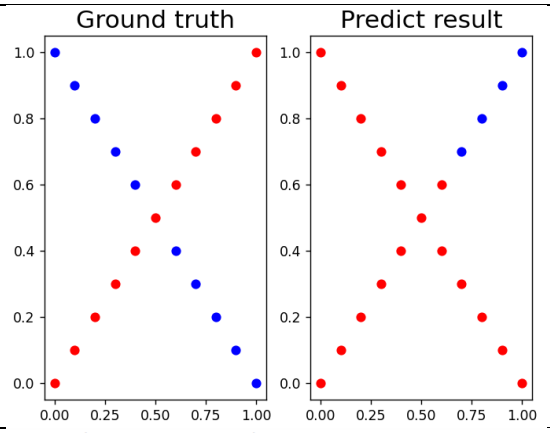
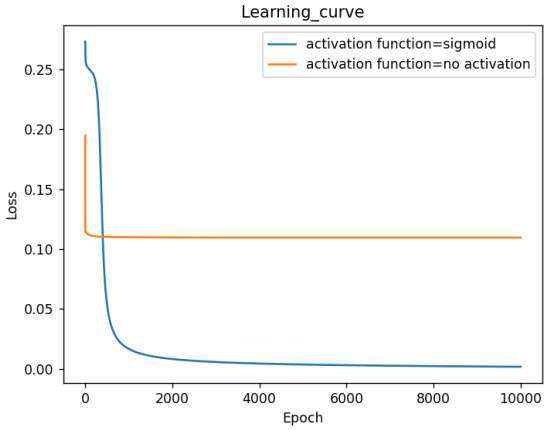
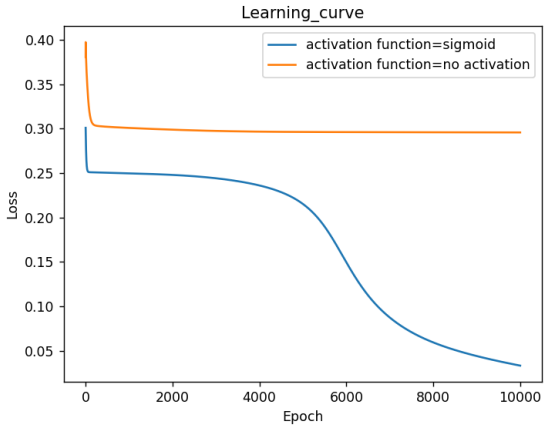
這個實驗的參數，除了 hidden units 數量外，其他都是用 class NuralNet 預設的參數，且 epoch 設 10000、loss function 用 mean-square error。

Linear	XOR
	

實驗顯示不管 hidden units 數量是 3、4 還是 5，對 Linear dataset、XOR dataset 作用的 accuracy 皆 100%。hidden units 數量越高對這 XOR dataset 來說，loss 會更快逼近 0，對 Linear dataset 來說，可能因為 hidden units 數量差距太小，loss 趨近於 0 的時間差不多。

C. Try without activation functions

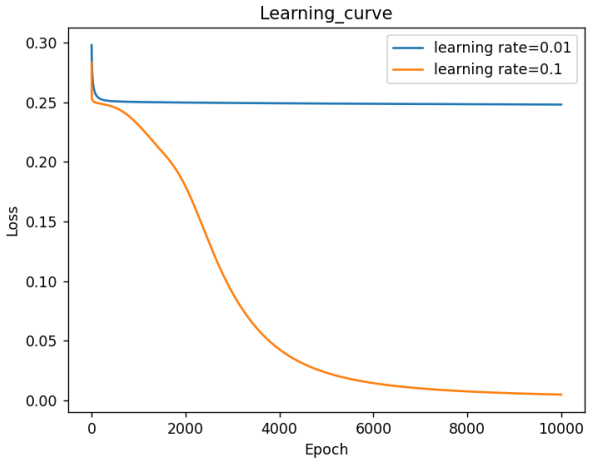
這個實驗的參數，除了沒有用 activation function 以外，其他都是用 class NuralNet 預設的參數，且 epoch 設 10000、loss function 用 mean-square error。

Linear	XOR
	
Iter95 | Ground truth:0 | prediction:0.025505990287475463 Iter96 | Ground truth:0 | prediction:0.06872604434142043 Iter97 | Ground truth:1 | prediction:1.2004733756012822 Iter98 | Ground truth:0 | prediction:0.266729186288339 Iter99 | Ground truth:1 | prediction:0.22496516002122546 Iter100 | Ground truth:1 | prediction:0.5675519656790436 loss=0.107879282203635 1 accuracy=87.0%	Iter16 | Ground truth:0 | prediction:0.5933645744760478 Iter17 | Ground truth:1 | prediction:0.37511597881411785 Iter18 | Ground truth:0 | prediction:0.6675351462855539 Iter19 | Ground truth:1 | prediction:0.3765370187363139 Iter20 | Ground truth:0 | prediction:0.7417057180950597 Iter21 | Ground truth:1 | prediction:0.3779580586585099 loss=0.2893558934218439 accuracy=33.33333333333333%
	

上表格顯示沒有 activation function，會使 Linear dataset、XOR dataset 的 accuracy 降低非常多，loss 也一直無法降到趨近 0。

D. Anything you want to share

下面 5. Extra 的 A 部分，我的 optimizer 用 adagrad 在 XOR dataset 上 accuracy 只有 52.38%，所以我想嘗試改變 learning rate，從預設的 0.01 改成 0.1，下圖為在 XOR dataset 上對比的 learning curve。



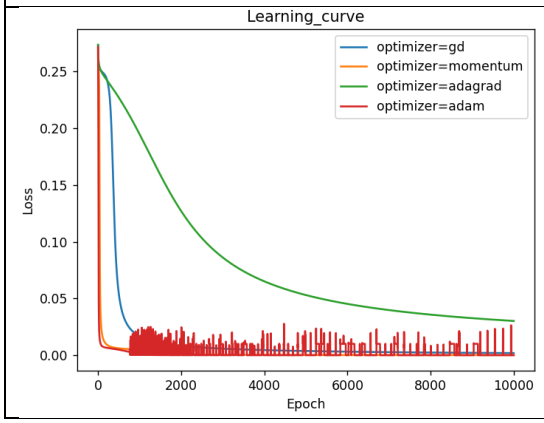
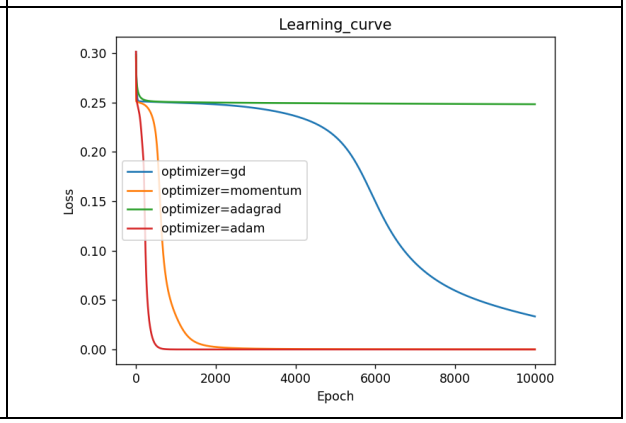
可以明顯看到 loss 有持續下降了，然後 accuracy 也改善變成 100%，這可能是因為 adagrad 的公式是要把 learning rate 除以之前的 gradient 總和，這導致一開始 learning rate 設太低的話，weight 每次要更新的幅度就會很小，導致 loss 不太能下降。

5. Extra

A. Implement different optimizers

這個實驗的參數，除了 optimizer 不一樣外，其他都是用 class NuralNet 預設的參數，且 epoch 設 10000、loss function 用 mean-square error。

```
def optimize(self):
    if self.optimizer == 'gd':
        for i in range(3):
            self.layers[i].weight = self.layers[i].weight + (-self.learning_rate * self.layers[i].gradient.T)
    elif self.optimizer == 'momentum':
        for i in range(3):
            self.layers[i].movement = (0.9 * self.layers[i].movement) +  (-self.learning_rate * self.layers[i].gradient.T)
            self.layers[i].weight = self.layers[i].weight + self.layers[i].movement
    elif self.optimizer == 'adagrad':
        for i in range(3):
            self.layers[i].sum_square_gradient += np.square(self.layers[i].gradient)
            self.layers[i].weight = self.layers[i].weight + ((-self.learning_rate * self.layers[i].gradient.T) / (np.sqrt(self.layers[i].sum_square_gradient).T +1e-8)
    elif self.optimizer == 'adam':
        for i in range(3):
            self.layers[i].movement = (0.9 * self.layers[i].movement) +  (0.1 * self.layers[i].gradient.T)
            self.layers[i].movement_hat = self.layers[i].movement / (1 - (0.9 ** self.layers[i].t))
            if self.layers[i].t > 1 :
                self.layers[i].v = 0.999 * self.layers[i].v + 0.001 * np.square(self.layers[i].gradient).T
            else :
                self.layers[i].v = np.square(self.layers[i].gradient).T
            self.layers[i].v_hat = self.layers[i].v / (1 - (0.999 ** self.layers[i].t))
            self.layers[i].weight = self.layers[i].weight - (self.learning_rate * self.layers[i].movement_hat / (np.sqrt(self.layers[i].v_hat) + 1e-8))
            self.layers[i].t += 1
```

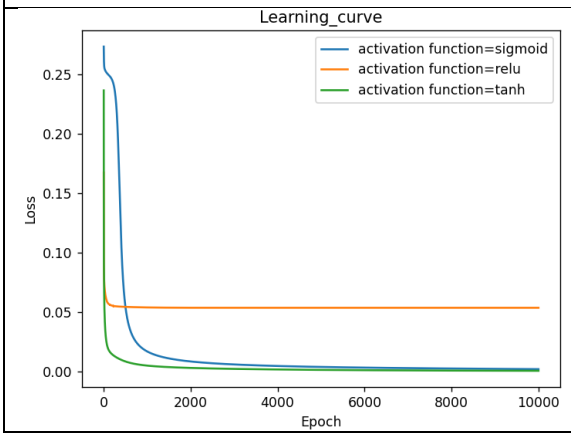
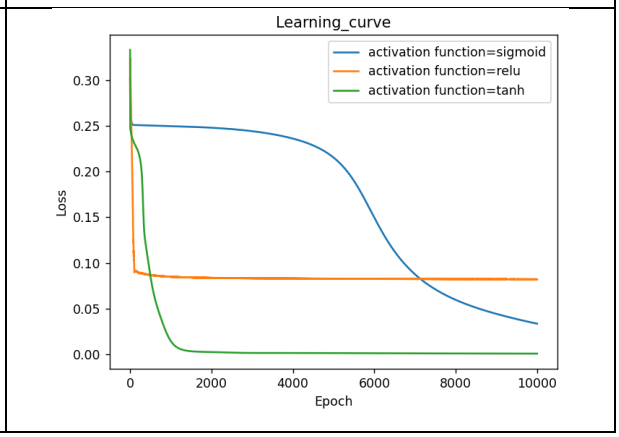
Linear	XOR
	

根據上表我們可以看到在 XOR dataset 上，Adam optimizer 的表現最佳。相較之下，Adagrad 在兩個 dataset 上的表現都不甚理想，尤其在 XOR 數據集上，accuracy 甚至只有 52.38%。觀察圖上的曲線趨勢，有可能需要增加訓練的 epoch 數量來改善效果。至於其他的 optimizer，他們在 accuracy 上皆達到了 100%。

B. Implement different activation functions

這個實驗的參數，除了 activation function 不一樣外，其他都是用 class NuralNet 預設的參數，且 epoch 設 10000、loss function 用 mean-square error。

```
def sigmoid(a, derivative=False):
    if not derivative:
        return 1.0 / (1.0 + np.exp(-a))
    else:
        return np.multiply(a, 1.0 - a)

def relu(a, derivative=False):
    if not derivative:
        return np.maximum(0.0, a)
    else:
        return np.heaviside(a, 0.001)

def tanh(a, derivative=False):
    if not derivative:
        return np.tanh(a)
    else:
        return 1.0 - a**2
```

Linear	XOR
	

根據上述表格，我們可以觀察到 tanh 在兩個 dataset 上均有最優秀的表現。反觀 relu 在兩個 dataset 上的表現並不佳，其中在 Linear dataset 上的 accuracy 只有 96%，而在 XORdataset 上的 accuracy 僅為 90.48%。其他的 activation function 在 accuracy 上均達到了 100%。