Deep Learning Lab 1

311552004 李明倫

1. Introduction

In lab1, you need to implement a simple neural network from scratch using only Numpy and python standard libraries, any other frameworks (ex: Tensorflow, PyTorch) are not allowed.

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. Must use backpropagation in this neural network and can only use Numpy and other python standard libraries to implement.
4. Plot the comparison figure that shows the predicted results and the ground-truth.
5. Print the training loss and testing result.

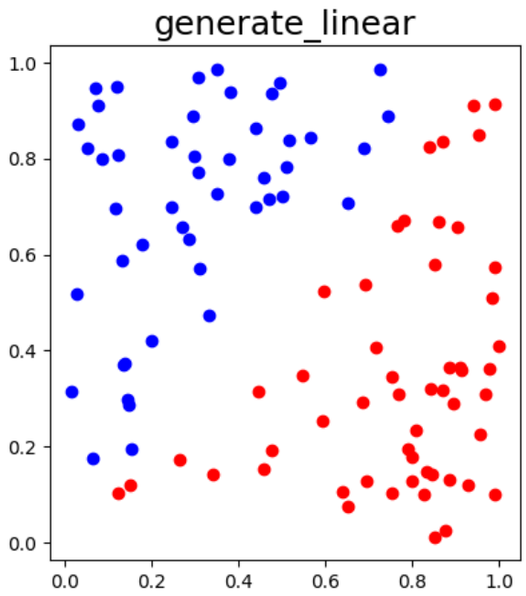
Dataset

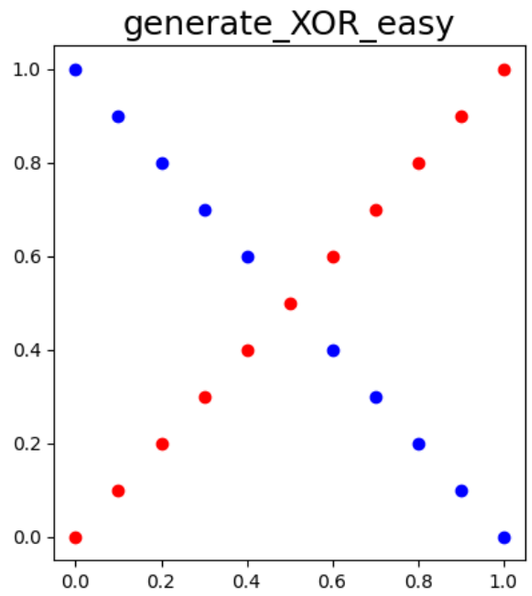
這次作業題提供兩種資料集，實作出的 neural network 要能夠正確的把資料作分類

- generate_linear:
- generate_XOR_easy:

```
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)
```





2. Experiment Setups

Sigmoid functions

按照 spec 中提及，定義 sigmoid function，用來作 forward pass 與 derivative_sigmoid 做 backward pass

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

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

Neural Network

在 neural network 的實作上，我定義了兩個 class: Dense, SimpleNet

Dense

- 單一層的神經網路，主要內容包括 weight, bias, gradient, activation function, z (線性組合完的結果), a (過完 activation function 的結果)
- 另外定義了一個 update() function，對 weight 跟 bias 做 gradient update

```
class Dense():
    def __init__(self, in_dim, out_dim, activation="sigmoid"):
        self.layer = self.linear(in_dim, out_dim)
        self.grad = [np.zeros(self.layer[0].shape), np.zeros(self.layer[1].shape)]
        self.m = [np.zeros(self.layer[0].shape), np.zeros(self.layer[1].shape)]     # momentum
        self.v = [np.zeros(self.layer[0].shape), np.zeros(self.layer[1].shape)]     # RMSprop
        self.t = 0

        if activation == "sigmoid":
            self.activation = sigmoid
        elif activation == "relu":
            self.activation = relu
        elif activation == "tanh":
            self.activation = tanh
        else:
            self.activation = identity

        self.Z = np.zeros((out_dim, 1))
        self.A = np.zeros((out_dim, 1))
        self.dc_dz = np.zeros((out_dim, 1))

    def linear(self, input_dim, output_dim):
        w = np.random.normal(0, 1, (output_dim, input_dim))
        b = np.random.normal(0, 1, (output_dim, 1))
        return [w, b]

    def update(self, new_grad):
        self.layer[0] -= new_grad[0]
        self.layer[1] -= new_grad[1]
```

SimpleNet

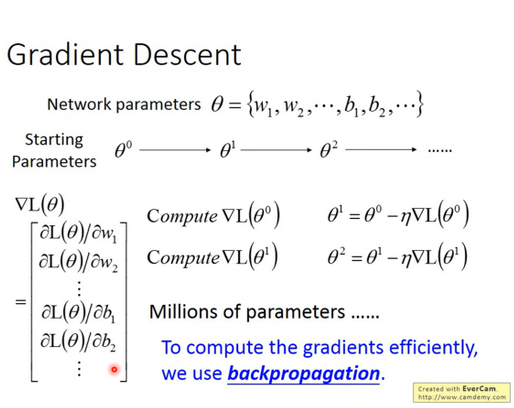
- 整個 neural network 的 class，一開始會先用給定的 input_size, hidden_size 與 output_size，去建立整個神經網路，每一層 layer 則透過上面的 Dense 去建立

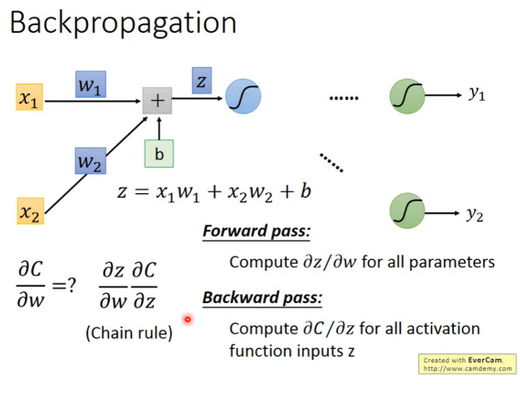
- 為了彈性起見，我的 network 可以有超過兩層的 hidden layer (透過給定 dimsension 去指定)，但在作業當中還是維持兩層 hidden layer 的設定

```
class SimpleNet():
    def __init__(self, dim=[2, 1], lr=0.001, activation="sigmoid", optimizer="sgd"):
        self.layers = self.build_layer(dim, activation)
        self.num_layers = len(self.layers)
        self.lr = lr
        self.optimizer = optimizer

    # 建每一層 network 中間的 weight 跟 bias (training 要 optimize 的對象)
    def build_layer(self, dim, activation):
        io_dims = list(zip(dim[:-1], dim[1:]))        # [(32, 64), (64, 128)]
        layers = []
        for idx, (in_dim, out_dim) in enumerate(io_dims):
            layers.append(Dense(in_dim, out_dim, activation))
        return layers

    # forward pass (把 activation 的 output 算出來，之後 backpropagation 要用到)
    def forward_pass(self, x):
        a = x.reshape(-1, 1)
        for i in range(self.num_layers):
            w, b = self.layers[i].layer
            z = np.dot(w, a) + b
            a = self.layers[i].activation(z)
            self.layers[i].Z = z
            self.layers[i].A = a
        return a

    # backward pass (把每一層的 dC/dz 算出來，然後算 gradient)
    def backward_pass(self, x, y, y_hat):
        # 計算 dC/dz
        self.layers[-1].dc_dz = mse(y, y_hat, derivative=True) * self.layers[-1].activation(self.layers[-1].A, derivative=True)
        for i in range(self.num_layers-2, -1, -1):
            self.layers[i].dc_dz = self.layers[i].activation(self.layers[i].A, derivative=True)
                                                * np.dot(self.layers[i+1].layer[0].T, self.layers[i+1].dc_dz)

        # 計算 gradient = a (前一層的 activation output) * dc_dz (當前這一層的微分)
        for i in range(self.num_layers):
            self.layers[i].grad[0] = x.T * self.layers[i].dc_dz
            self.layers[i].grad[1] = self.layers[i].dc_dz
            x = self.layers[i].A

        # 用 optimizer 去優化 gradient descent
        self.step()

    def step(self):
        for i in range(self.num_layers):
            new_grad = [0, 0]
            for j in range(2):                      # iterate for [weight, bias]
                if self.optimizer == "momentum":
                    self.layers[i].m[j] = (0.9 * self.layers[i].m[j]) - (self.lr * self.layers[i].grad[j])
                    new_grad[j] = -self.layers[i].m[j]
                elif self.optimizer == "adam":
                    self.layers[i].t += 1
                    self.layers[i].m[j] = (0.9 * self.layers[i].m[j]) + (1 - 0.9) * self.layers[i].grad[j]
                    self.layers[i].v[j] = (0.999 * self.layers[i].v[j]) + (1 - 0.999) * self.layers[i].grad[j]**2
                    m, v = [0, 0], [0, 0]
                    m[j] = self.layers[i].m[j] / (1 - (0.9)**self.layers[i].t)
                    v[j] = self.layers[i].v[j] / (1 - (0.999)**self.layers[i].t)
                    new_grad[j] = self.lr * m[j] / (np.sqrt(v[j]) + 1e-8)
                elif self.optimizer == "sgd":
                    new_grad[j] = self.lr * self.layers[i].grad[j]
            self.layers[i].update(new_grad)
```

Backpropagation

- 一個比較有效率的演算法，可以比較有效率地把 gradient 的那個 vector 計算出來

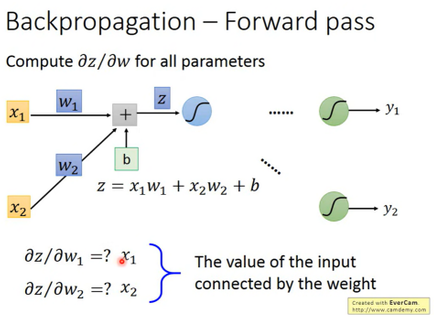




計算某一個 weight (w1, w2, b, …) 對 example 的某一個 cost 的偏微分 *∂C/∂w*，可以先把他拆成 *∂C/∂z . ∂z/∂w (chain rule)*。

- ***∂z/∂w - Forward pass***

  微分的東西就是看他前面接的東西是甚麼 (x1, x2, ...)



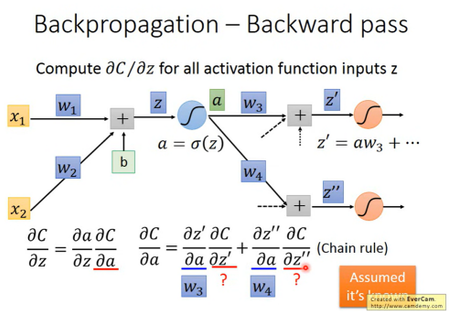
- ***∂C/∂z - Backward pass***

  z 會再經過一個 activation (sigmoid)，假設得到 a，就可以把 *∂C/∂z* 拆開成 *∂C/∂a . ∂a/∂z*

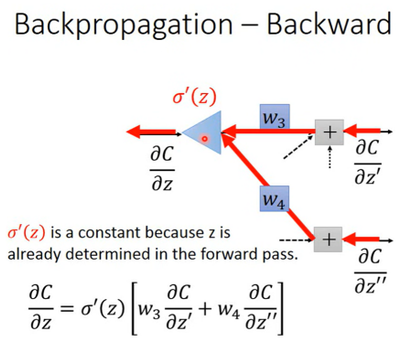
  - *∂a/∂z* 就是 sigmoid 的微分

  而因為 a 會再透過 z', z" 往後影響 C，所以一樣可以透過 chain rule，把 *∂C/∂a* 拆開成 *(∂C/∂z' . ∂z'/∂a) + (∂C/∂z" . ∂z"/∂a)*

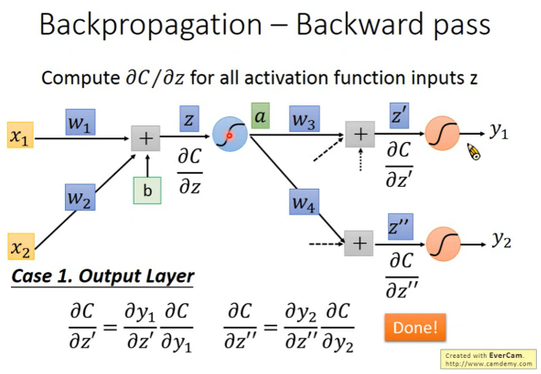
  - *∂z'/∂a* 就是 w3， *∂z"/∂a* 就是 w4

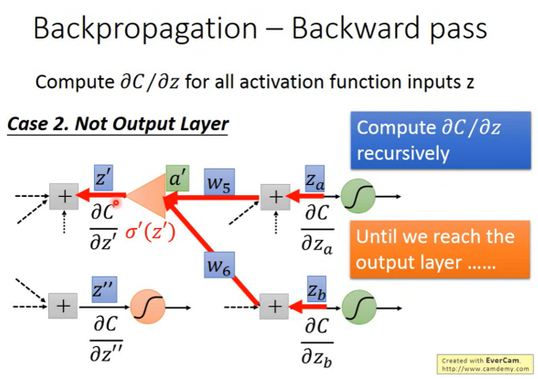


整理一下就變成下面這樣

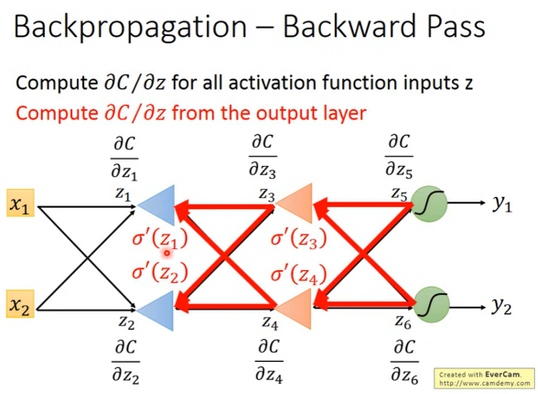


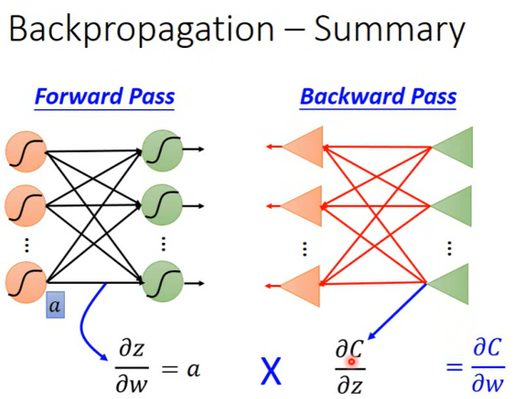
然後再分成兩種 case: 後面是 output layer / 後面不是 output layer





最後再把 forward pass 跟 backward pass 的結果相乘就結束了





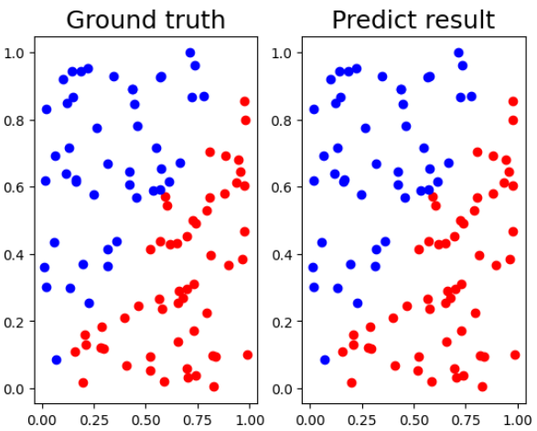
在程式中，我實作了 forward_pass, backward_pass 與 step

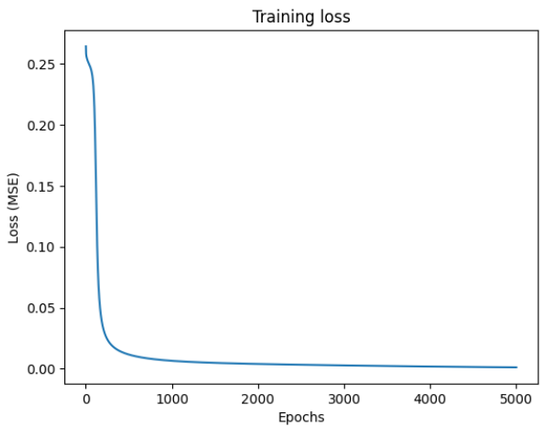
- forward_pass: 把 input data 完整的從第一層 layer，透過一連串的線性組合與 activation，一路運算到 output layer
- backward_pass: 把每一層的 dC/dz 從最後一層一路往回算到第一層，然後結合 forward_pass 的結果，計算出中間所有參數 (weight, bias) 的 gradient
- step: 得到所有參數的 gradient 之後，就可以透過不同的 optimizer 去做 gradient descent 的 optimization，這邊我實作了: sgd, momentum, adam

3. Result of your training

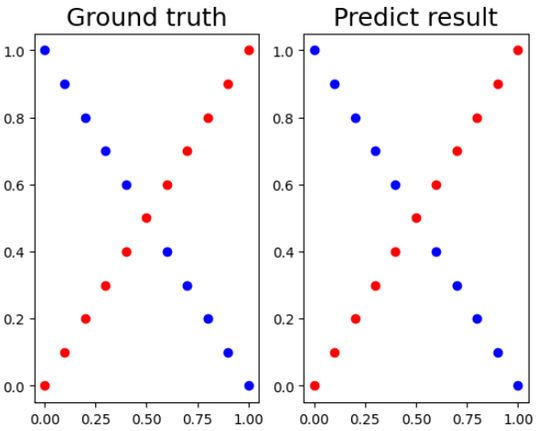
Screenshot and comparison figure / Learning curve (loss, epoch curve)

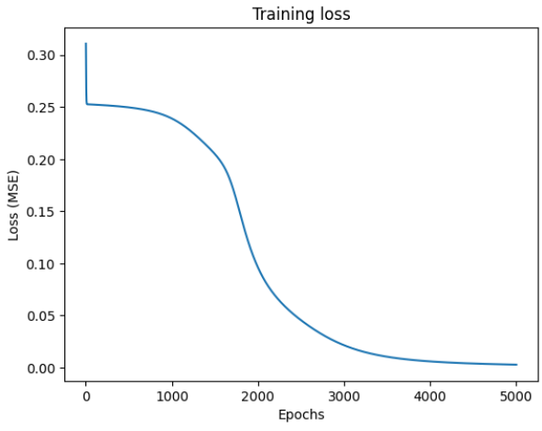
- **Linear, accuracy: 100%**





- **XOR, accuracy: 100%**





4. Discussion

Try different learning rates

在這個部分，我試了三種大小的 learning rate: 0.3, 0.03, 0.003。

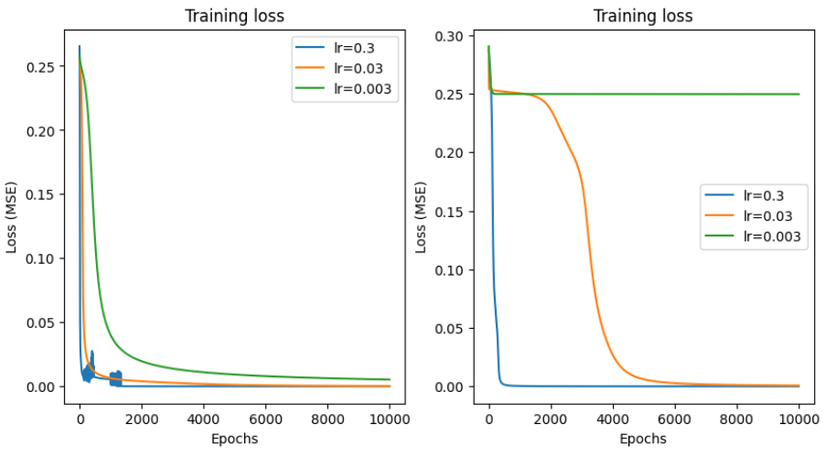
左邊是 **Linear**，右邊是 **XOR** 的結果。可以看到說 learning rate 越大，代表每一次 gradient update 的距離就越大，因此在這兩個資料集的表現上，learning rate 越大者會越快收斂，而如果 learning rate 設的太小，也有機會像右邊的 lr=0.003，走的步伐太小，造成模型還沒有走到最佳解的地方。

Linear

- lr: 0.3, Accuracy: 100%
- lr: 0.03, Accuracy: 100%
- lr: 0.003, Accuracy: 100%

XOR

- lr: 0.3, Accuracy: 100%
- lr: 0.03, Accuracy: 100%
- lr: 0.003, Accuracy: 52.39%



Try different numbers of hidden units

在這個部分，我試了三種大小的 hidden size: 4, 16, 64。

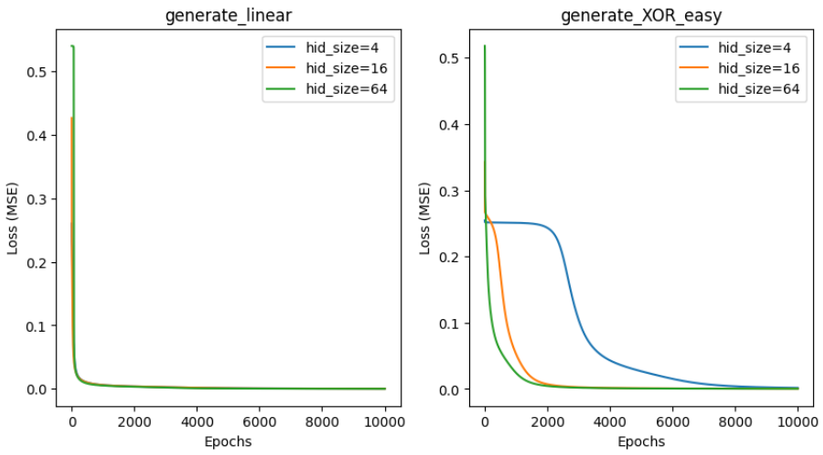
可以看到說 hidden size 越大，代表模型的能力越強，因此在這兩個資料集的表現上，hidden size 越大者會越快收斂。

Linear

- hidden: 4, Accuracy: 100%
- hidden: 16, Accuracy: 100%
- hidden: 64, Accuracy: 100%

XOR

- hidden: 4, Accuracy: 100%
- hidden: 16, Accuracy: 100%
- hidden: 64, Accuracy: 100%



Try without activation functions

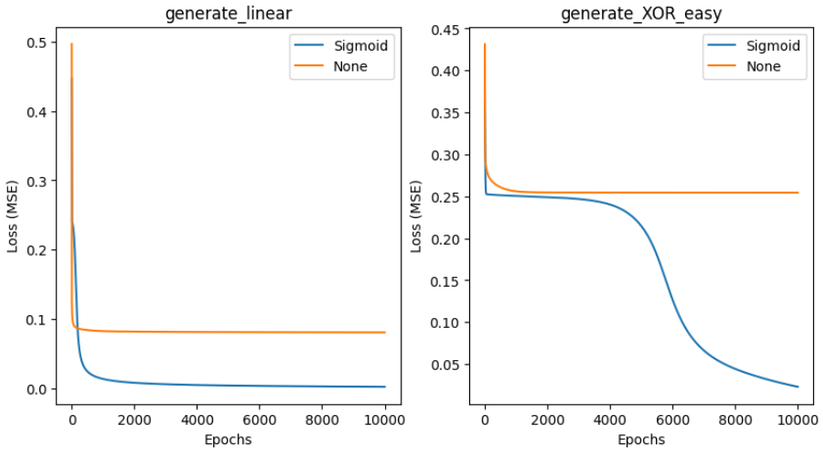
可以看到說少了 activation function 的加入，會讓模型在訓練過程中都是重複的在做線性組合，而沒有辦法做到一些 non-linear 的轉換，一方面讓模型的能力比較弱，另一方面也會像 XOR 的結果一樣，沒有辦法成功的學會分類。

Linear

- Sigmoid, Accuracy: 100%
- None, Accuracy: 95%

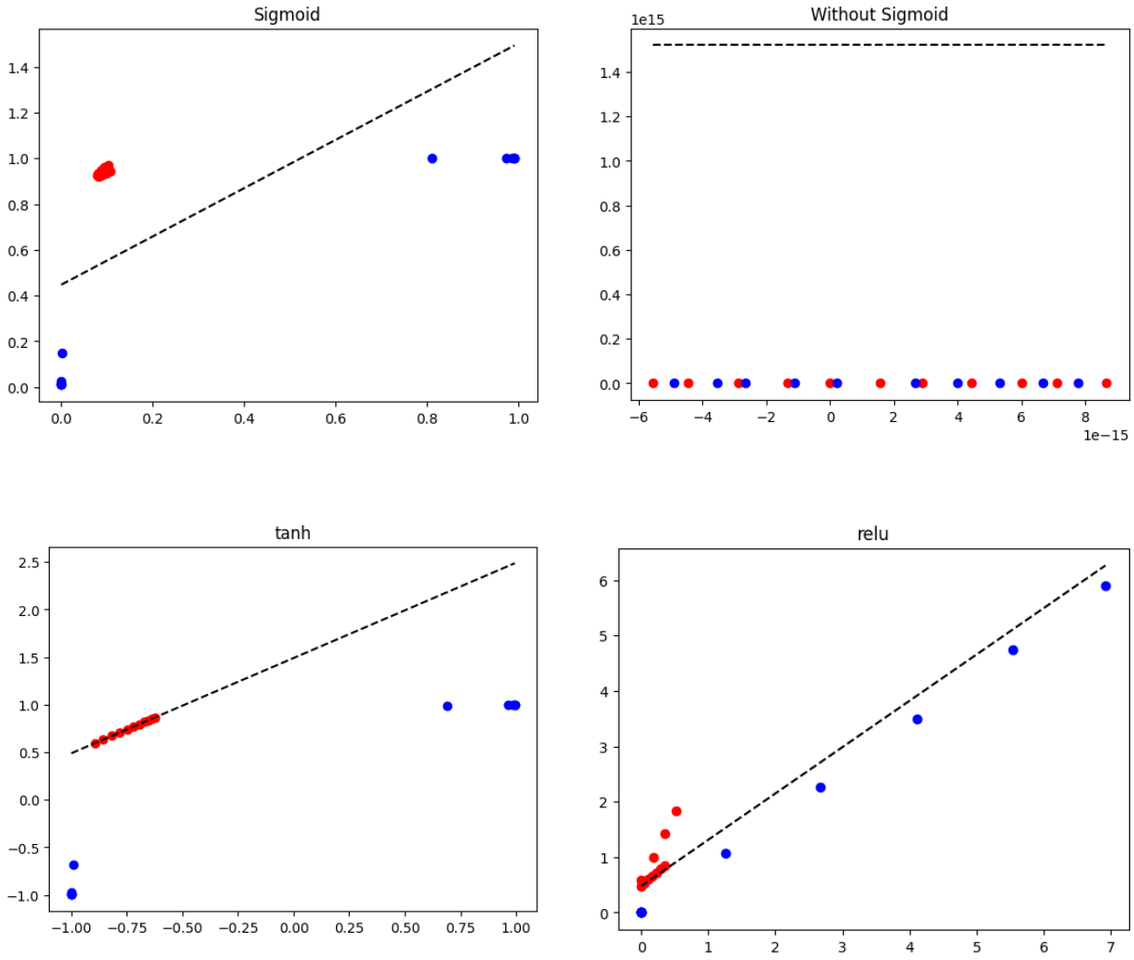
XOR

- Sigmoid, Accuracy: 100%
- None, Accuracy: 52.39%



Anything you want to share

我有另外將有 activation 與沒有 activation 進行比較，目的是要看說能做到 non-linear transformation 的意義是甚麼。在這裡我把最後一層 hidden layer 的 output 拿出來看，可以看到說 sigmoid, tanh, relu 都有成功的把原本 XOR 的 data 做空間上的轉換，找到不同 class 之間的 decision boundary；而沒有用任何 activation function 的結果，則沒有辦法成功讓資料被分開來。因此可以看到說為甚麼 neural network 裡面需要加入 activation function 幫我們做一些 non-linear 的轉換。



5. Extra

Implement different optimizers

在這個部分中，我另外實作了 Momentum 與 Adam 兩個 optimizer。從結果來看，因為資料集比較簡單，所以當 training 時間夠久的時候，三個 optimizer 都能夠有效的做好 gradient descent，而另外也可以看到 Adam 因為有考慮到比較多優化 gradient 的特性，所以在收斂速度上會比較快。

```
if self.optimizer == "momentum":
    self.layers[i].m[j] = (0.9 * self.layers[i].v[j]) - (self.lr * self.layers[i].grad[j])
    new_grad[j] = -self.layers[i].m[j]
elif self.optimizer == "adam":
```

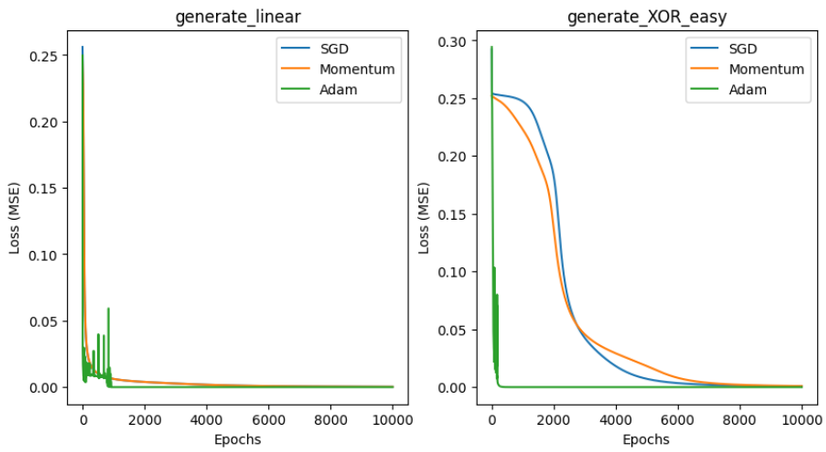
```
    self.layers[i].t += 1
    self.layers[i].m[j] = (0.9 * self.layers[i].m[j]) + (1 - 0.9) * self.layers[i].grad[j]
    self.layers[i].v[j] = (0.999 * self.layers[i].v[j]) + (1 - 0.999) * self.layers[i].grad[j]**2
    m, v = [0, 0], [0, 0]
    m[j] = self.layers[i].m[j] / (1 - (0.9)**self.layers[i].t)
    v[j] = self.layers[i].v[j] / (1 - (0.999)**self.layers[i].t)
    new_grad[j] = self.lr * m[j] / (np.sqrt(v[j]) + 1e-8)
```

Linear

- SGD, Accuracy: 100%
- Momentum, Accuracy: 100%
- Adam, Accuracy: 100%

XOR

- SGD, Accuracy: 100%
- Momentum, Accuracy: 100%
- Adam, Accuracy: 100%



Implement different activation functions

在這個部分中，我另外實作了 tanh 與 relu 兩個 activation function。從結果來看，三個 activation function 都能夠有效把訓練收斂，而另外也可以看到 tanh 在處理 XOR 資料的表現上最好。

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

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

Linear

- Sigmoid, Accuracy: 100%
- tanh, Accuracy: 100%
- relu, Accuracy: 100%

XOR

- Sigmoid, Accuracy: 100%
- tanh, Accuracy: 100%
- relu, Accuracy: 100%