Deep Learning HW1

Introduction

Lab1，在不使用 pytorch 的前提下實作神經網路，其中至少包含

- Implement simple neural network with two hidden layers
- Each hidden layer needs to contain at least one transformation and one activate function.
- Must use backpropagation in this neural network and can only use `Numpy` and other python standard libraries to implement.
- Plot the comparsion figure that shows the predicted results and the ground-truth.
- Be sure that output is meet the requirement.

使用的 dataset 為從函數

- generate_linear
- generate_XOR_easy

回傳的結果

Experiment Setups

A. Sigmoid functions

```
### implememt the activation function
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def derivative_sigmoid(x):
    return sigmoid(x) * (1-sigmoid(x))
```

B.Neural network

In this part, I divide the neural network in two classes: `layers` and `network`.

Class `layers` 中實作了 forward propagation, backward propagation, compute gradient, update weight 四個函數。

Class `network` 主要是向前或向後傳遞 `layer` 計算的結果

C. Backward Propagation

用於計算或更新每個 `layer` 的梯度，或更準確地說， `layer` 中每個weight (神經元) 的梯度，其方法為：
用預測值與真實值之間的 error 作為輸入。主要目的在找每個 weight 對 output 產生多少影響
Suppose we use the sigmoid function as the activation function while forward propagation, then we need to use the first direvative of it to update the gradient. And output the result to the preivous layer.

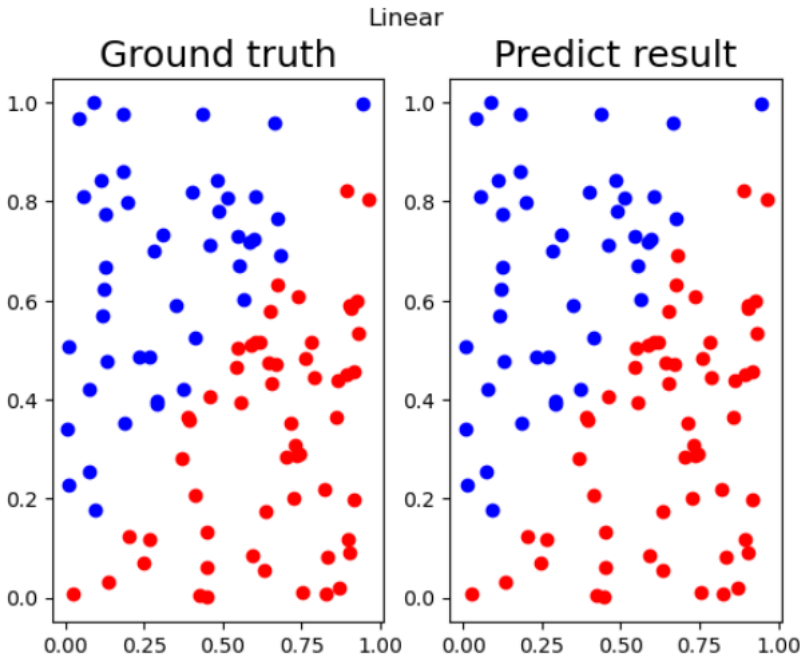
```
class Layer():
...
    def backward(self, x):
        self.up_grad = x    #將loss 作為 input
        if self.activate is not None:
            if self.activate == 'sigmoid':
                self.up_grad *= derivative_sigmoid(self.z)   #計算誤差影響
            if self.activate == 'tanh':
                self.up_grad *= derivative_tanh(self.z)

        return self.up_grad @ self.w.T    #傳遞誤差給前一層

    def step(self, lr, optim):
        ### update the weight
        if optim == 'Adam':
            ...
        else:
        ### gradient decent
            grad_w = self.local_grad.T @ self.up_grad    #計算梯度下降的值
            self.w -= lr * grad_w      #更新權重
            if self.bias:
                grad_b = self.local_grad_b.T @ self.up_grad
                self.b -= lr * grad_b.squeeze()

...

class Network():
...
    def backward(self, loss):
        x = loss
        for i in reversed(range(self.num_layer)):
            x = self.layers[i].backward(x)    #將loss 向前傳遞

    def step(self,):
        for i in range(self.num_layer):
            self.layers[i].step(self.lr, self.optim)    #每一層更新自己的權重
```

Results of my testing

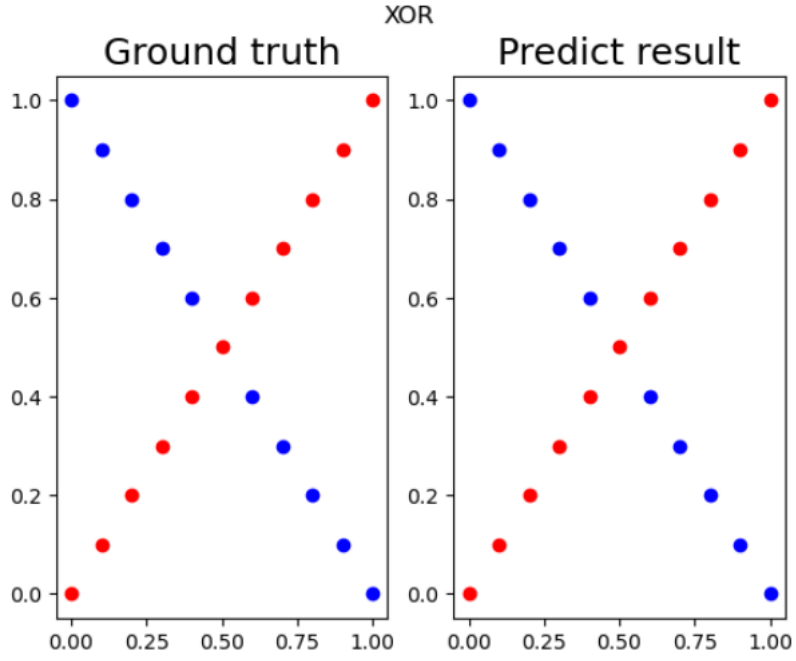
實驗設定：

```
    learning rate = 1.0
    optimizer = Gradient Decent
    batch size = 3, 4 (linear and XOR, respectively)
    hidden layer size： 8  (2 hidden layers)
    training epsilon = 0.005
    maximum epoch = 100000
    1 epoch = 500 step
```

A. Comparison Figure and Accuracy




```
Test Loss:  0.006055979122183502
acc: 99.0 %
```




```
Test Loss:  0.0048136804045673795
acc: 100.0 %
```

B. Accuracy

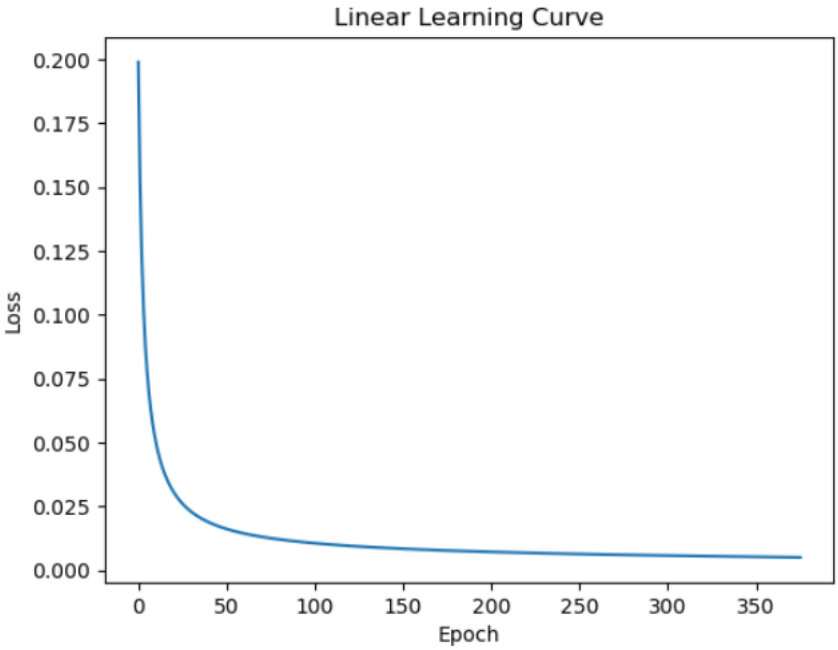
Linear data:

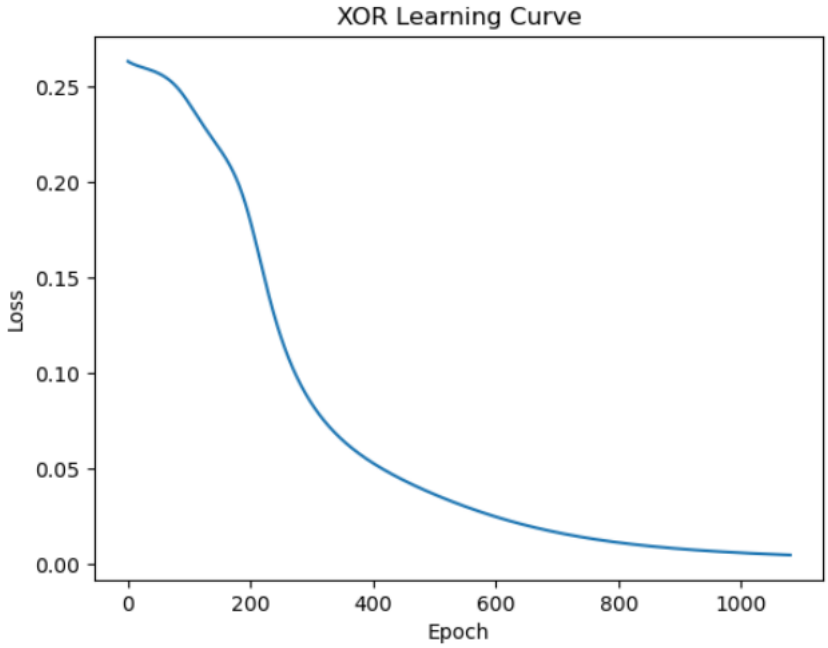
```
Test Loss:  0.006055979122183502
acc: 99.0 %
```

XOR data:

```
Test Loss:  0.0048136804045673795
acc: 100.0 %
```

C. Learning Curve

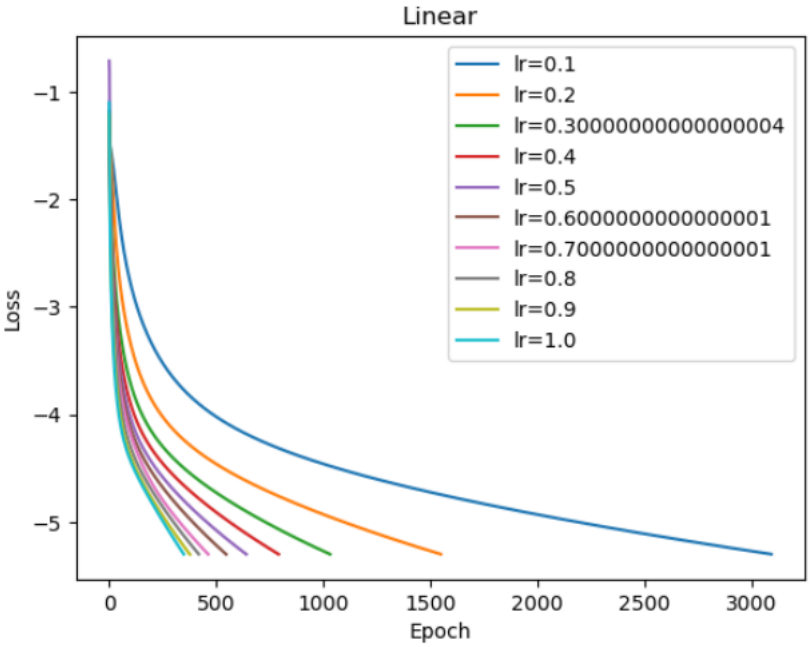


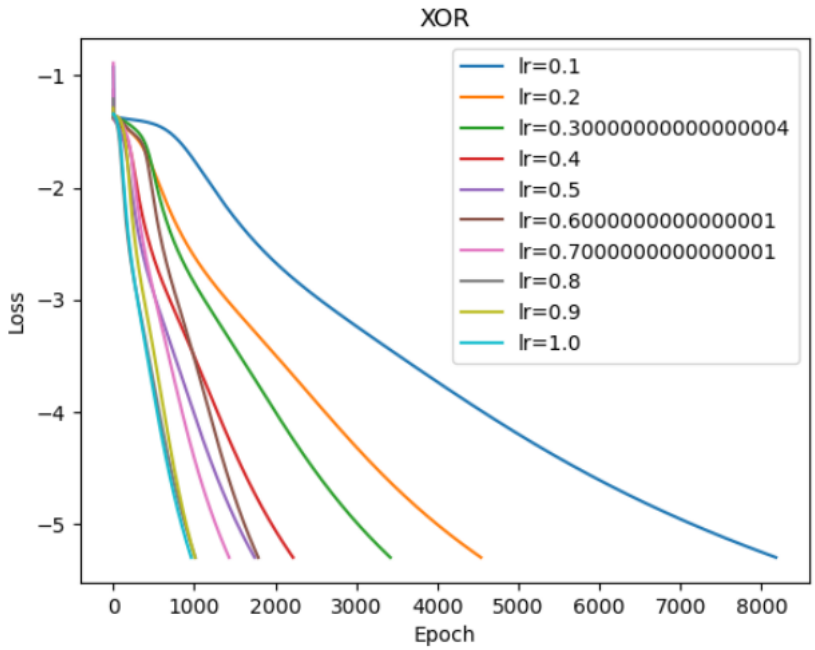


Discussion

A. Different learning rates

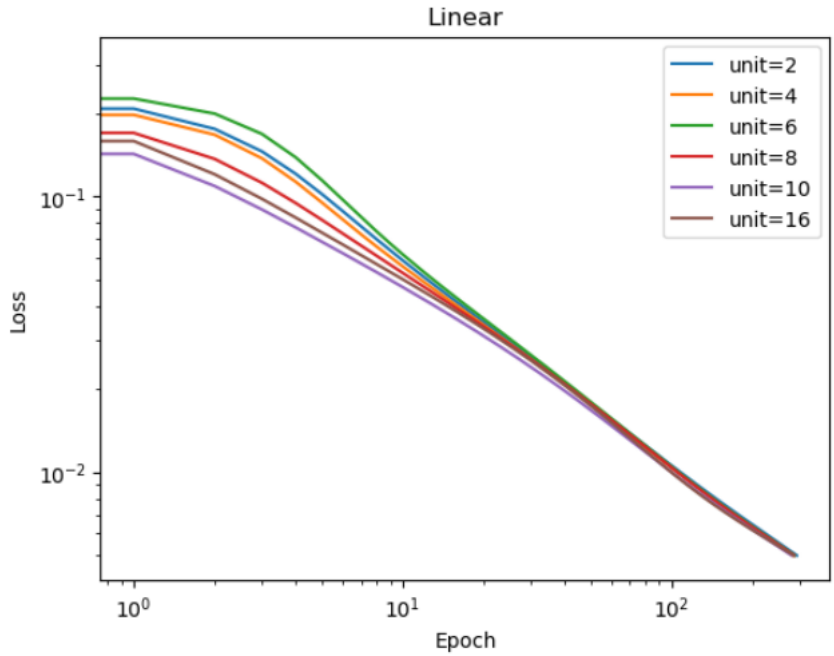
Note that these learning rates are shown in **log** scale

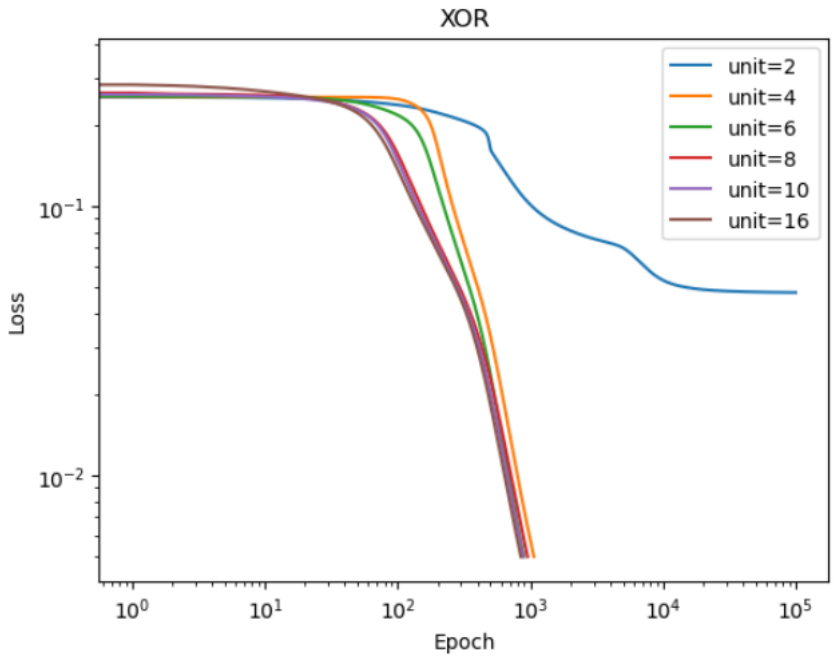

Linear
lr=0.1
lr=0.2
lr=0.30000000000000004
lr=0.4
lr=0.5
lr=0.6000000000000001
lr=0.7000000000000001
lr=0.8
lr=0.9
lr=1.0
Loss
−1
−2
−3
−4
−5
0
500
1000
1500
2000
2500
3000
Epoch


XOR
lr=0.1
lr=0.2
lr=0.30000000000000004
lr=0.4
lr=0.5
lr=0.6000000000000001
lr=0.7000000000000001
lr=0.8
lr=0.9
lr=1.0
Loss
−1
−2
−3
−4
−5
0
1000
2000
3000
4000
5000
6000
7000
8000
Epoch

B. Different numbers of hidden units

Note that these learning rates are shown in **loglog** scale





C. Without activation functions

若沒有 activation function，整個模型將會變成線性的(Linear)，這代表模型會用線性回歸(Linear regression)的方式來解決問題。但是在線性回歸中必須假設樣本為常態分布，而在我們的資料集中，樣本都是二元分布的，所以模型會訓練不起來。

D. Others

Extra

A. Different optimizers

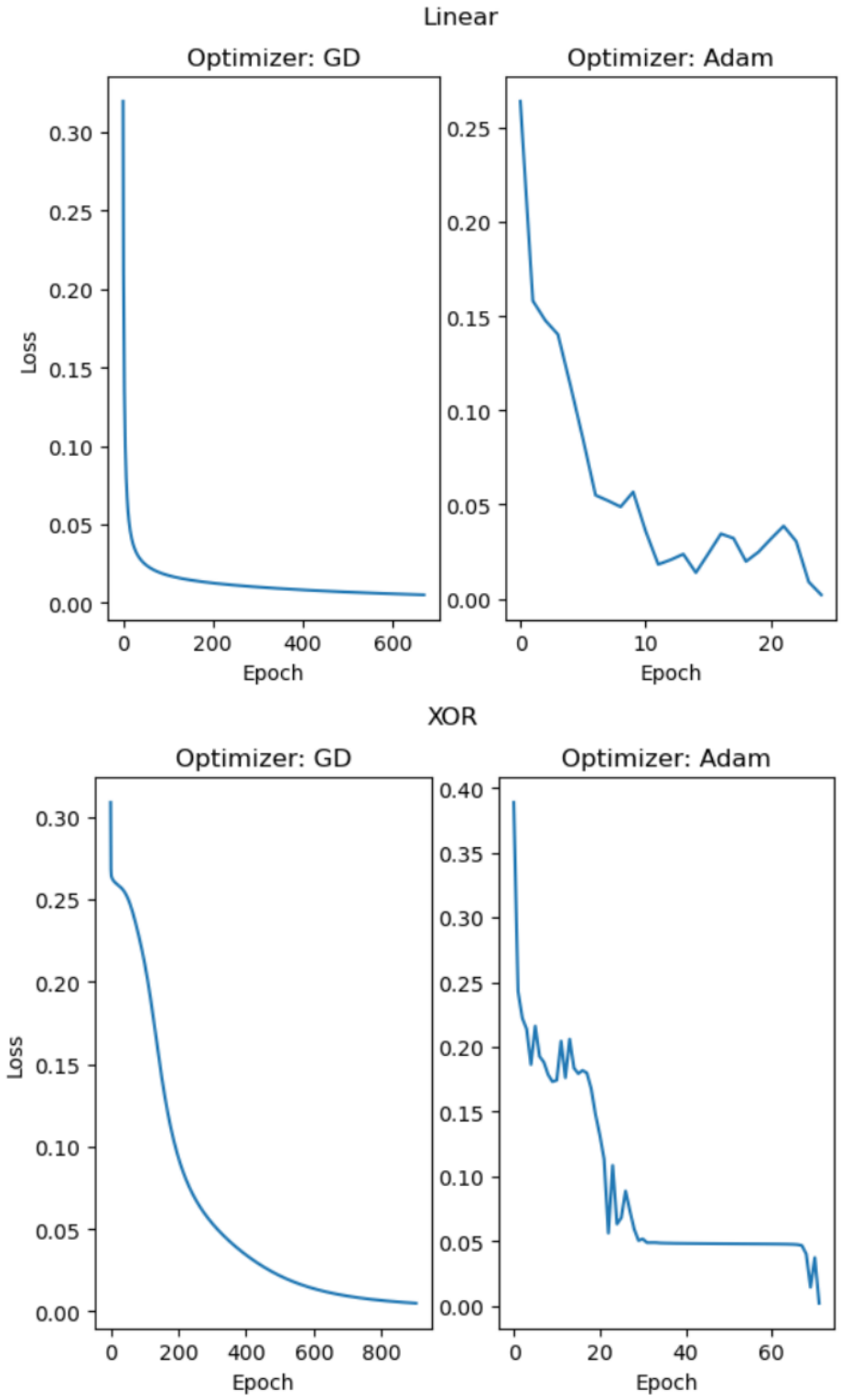
機器學習中有名的 optimizer 就屬 Adams 了，下面附上演算法：

$$
\begin{aligned}
&\textbf{input}: \gamma \text{ (lr)}, \beta_1, \beta_2 \text{ (betas)}, \theta_0 \text{ (params)}, f(\theta) \text{ (objective)} \\
&\qquad \lambda \text{ (weight decay)}, \ amsgrad, \ maximize \\
&\textbf{initialize}: m_0 \leftarrow 0 \text{ (first moment)}, v_0 \leftarrow 0 \text{ (second moment)}, \widehat{v_0}^{max} \leftarrow 0 \\
&\textbf{for } t = 1 \textbf{ to } \ldots \textbf{ do} \\
&\quad \textbf{if } maximize: \\
&\qquad g_t \leftarrow -\nabla_\theta f_t(\theta_{t-1}) \\
&\quad \textbf{else} \\
&\qquad g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \\
&\quad \textbf{if } \lambda \neq 0 \\
&\qquad g_t \leftarrow g_t + \lambda \theta_{t-1} \\
&\quad m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
&\quad v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\
&\quad \widehat{m_t} \leftarrow m_t / \big(1 - \beta_1^t\big) \\
&\quad \widehat{v_t} \leftarrow v_t / \big(1 - \beta_2^t\big) \\
&\quad \textbf{if } amsgrad \\
&\qquad \widehat{v_t}^{max} \leftarrow \max(\widehat{v_t}^{max}, \widehat{v_t}) \\
&\qquad \theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}^{max}} + \epsilon\big) \\
&\quad \textbf{else} \\
&\qquad \theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m_t} / \big(\sqrt{\widehat{v_t}} + \epsilon\big) \\
&\textbf{return } \theta_t
\end{aligned}
$$

(來源：**pytorch 官網** (https://pytorch.org/docs/stable/generated/torch.optim.Adam.html))

考慮簡單實作內容，將一些參數設為定值：

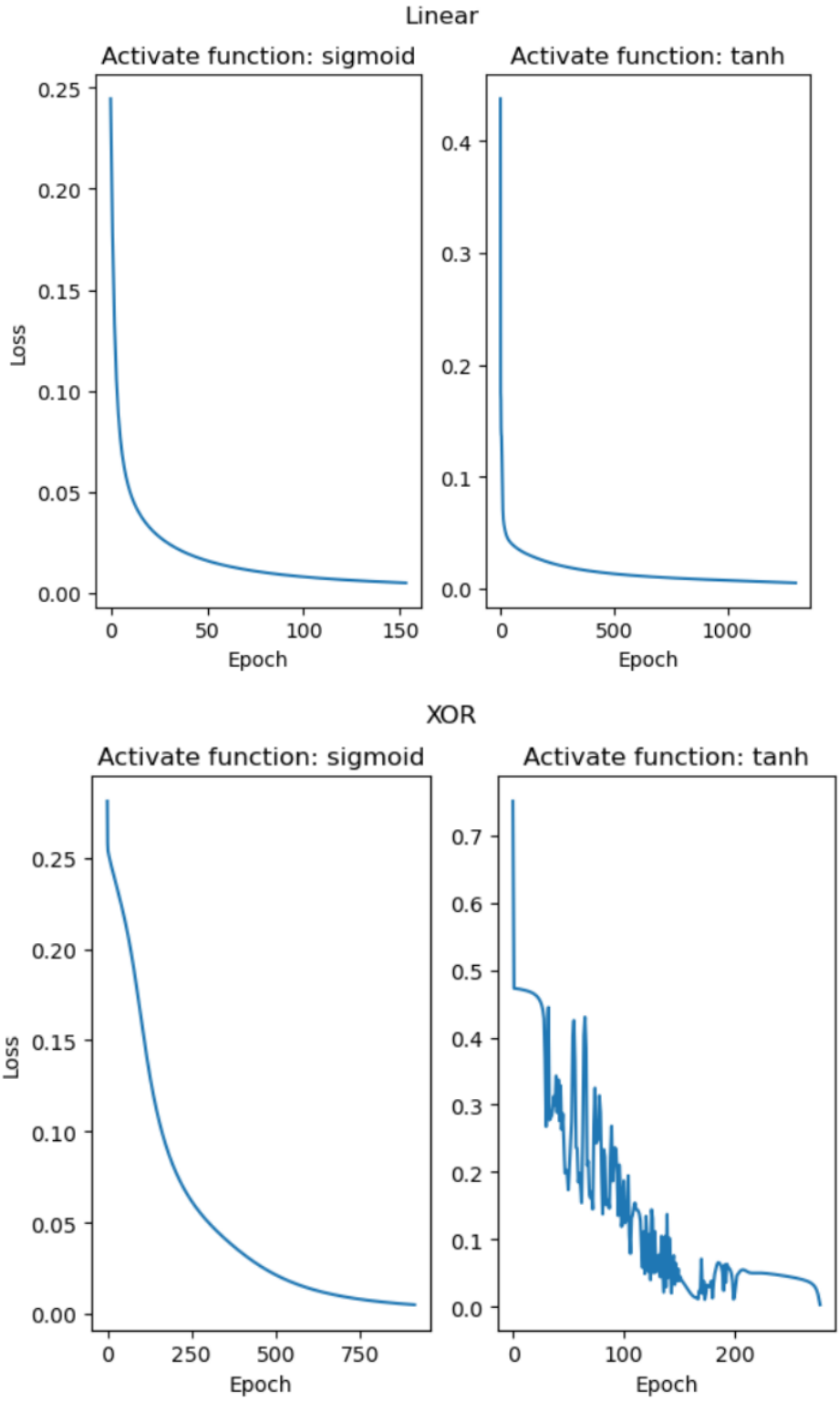
```
1. weight_decay = 0
2. this optimizer is use for minimize the loss
3. betas = [0.9, 0.999]
4. amsgrad = False
5. epsilon = 1e-8
```



B. Different activation functions

由於是二元分類，這裡選擇實作 tanh function

```
def tanh(x):
    return (1.0 - np.exp(-2 * x)) / (1.0 + np.exp(-2 * x))

def derivative_tanh(x):
    return 1 - tanh(x) ** 2
```



C. Convolutional layers

None