Backpropagation
Lab Report # 1

By
312581020
許瀚丰

Deep Learning
Spring 2024
Date Submitted: March 26, 2024

Contents

Listings

1 Introduction

1.1 Lab Description

本次實驗是實作一個簡單的Neural Network，並包含了利用Forward來預測答案，再透過Backward來更新網路的參數。而本實驗的目標就是希望讓預測的答案與實際的答案越像越好，並透過兩個任務來驗證模型的好壞，分別為一個簡單的線性可分的Linear問題與一個線性不可分的XOR問題。

1.2 Objective

在此實驗中，我希望能夠透過嘗試撰寫自己的Package的實作方式，希望能夠做出類似於Pytorch的工具，在未來遇到簡單的分類問題就能夠直接套用。而在實作上，我分別將一些常用的功能分開撰寫，分別為以下幾種：

nn.py

包含常見的Activation Function和NN的初始化與Forward、Backward等。

loss.py

包含在分類模型中常見的loss function，如BCELoss, CrossEntropyLoss等。

optim.py

包含一些常見的optimizer與一個簡單的learning rate scheduler。

utils.py

包含save model 、load model 與一些常用的功能。

2 Experiment setups

在本實驗中，我共有兩種不同的Abstract Class，分別為給Activation Function繼承的的Activation Class與給不同Network繼承的Layer Class，分別都需要實作Forward與Backward，並紀錄從前一層來的Net Input，以利之後需要更新時能夠直接使用，如程式碼1與2，分別為Activation Class 與Layer Class。

```python
class activation(ABC):
    def __init__(self):
        self.a = None
    @abstractmethod
    def forward(self):
        pass
    @abstractmethod
    def backward(self):
        pass
```

Listing 1: Activation Class

```python
class layer(ABC):
    def __init__(self):
        self.w = None
        self.b = None
        self.a = None
    @abstractmethod
    def forward(self):
        pass
    @abstractmethod
    def backward(self):
        pass
```

Listing 2: Layer Class

2.1 Sigmoid

2.1.1 Method

sigmoid function能夠將輸入從$(-\infty, \infty)$ mapping成$(-1, 1)$，且處處可導。其公式為$\sigma(x) = \frac{1}{1+e^{-x}}$，而在計算時，我們需要對於其求導，化簡後可以得到一個簡單的結果$\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$，其證明如下：

$$\frac{d\sigma(x)}{dx} = (\frac{1}{1+e^{-x}})'(e^{-x})' = \frac{-1}{(1+e^{-x})^{-2}} * -e^{-x} = \frac{e^{-x}}{(1+e^{-x})^2} = \frac{1}{(1+e^{-x})}\frac{e^{-x}}{(1+e^{-x})}$$
$$= \sigma(x) * \frac{(1+e^{-x})-1}{(1+e^{-x})} = \sigma(x) * (1 - \frac{1}{(1+e^{-x})}) = \sigma(x)(1-\sigma(x))$$

2.1.2 Implementation

我的實作方式參考了助教給的程式碼，在forward時，就直接回傳input經過sigmoid的結果，而在backward時，由於是對於已經經過sigmoid 的結果做偏微分，因此直接對其計算即可，如程式碼3所示：

```
class sigmoid(activation):
    def forward(self, x):
        self.a = x
        return 1.0 / (1.0 + np.exp(-x))
    def backward(self, x):
        return np.multiply(x, 1 - x)
```

Listing 3: Sigmoid

2.2 Neural Network

2.2.1 Linear Layer

對於Neural Network的部分，首先我先實作出Linear Layer的架構，一開始會先初始化每層的weights與bias，而Forward就是簡單的將weights乘上x再加上bias，而內部的a則是記錄每層輸入進來的input，用以在對權重做更新時能夠使用，如程式碼4所示：

```
class Linear(layer):
    def __init__(self, in_features, out_features):
        self.w = np.random.randn(in_features, out_features).T
        self.b = np.random.randn(1, out_features).T
        self.a = None
    def forward(self, x):
        self.a = x
        return self.w @ x + self.b
    def backward(self, delta):
        return self.w.T @ delta
```

Listing 4: Linear Layer

2.2.2 Loss Function

在本次實驗中Loss Function我選擇的是Binary Cross Entropy，原因為其相比與Mean

Square Error更適合處理這種分類的問題，使用時就將預測結果$\hat{y}$與真實的Label y帶入公式$\mathcal{L}_{BCE}(\hat{y}, y) = -(y\log(\hat{y}) + (1-y)\log(1-\hat{y}))$，並回傳其對於$\hat{y}$的偏微分$\frac{\partial L}{\partial \hat{y}} = -(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}})$，如程式碼5所示。

```
class BCELoss():
    def __init__(self):
        pass
    def __call__(self, y_pred, y_target, eps = 1e-6):
        y_pred = np.clip(y_pred, eps, 1 - eps)
        loss = -np.mean(y_target * np.log(y_pred) + (1 - y_target) * np.log(1 - y_pred
    ))
        delta = -(y_target / y_pred - (1 - y_target) / (1 - y_pred))
        return loss, delta
```

Listing 5: Binary Cross Entropy Loss

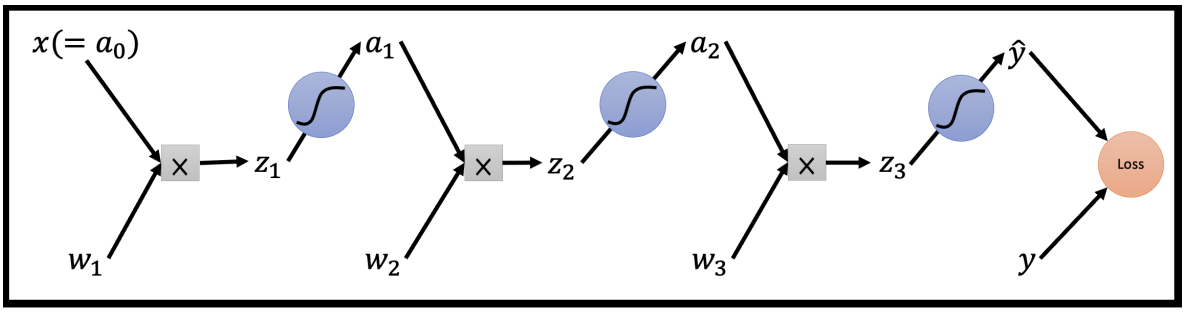


Figure 1: 此圖為一個簡單的Computational Graph範例。

2.3 Backpropagation

2.3.1 Method

在Backpropagation的部分，為了減少一些重複的計算，我使用Delta Rule的方式來更新網路。假設在模型中Loss Function為Binary Cross Entropy(BCELoss)，而最後一層則透過Sigmoid Function來讓結果介於(-1,1)之間，通過簡化我們可以得到最後一層L的$\Delta_L = \hat{y} - y$，而對於其他層$L-1$到1，其$\Delta_l = ((\mathbf{W}_{l+1})^T\Delta_{l+1}) \odot [\mathbf{a}_l \odot (1-\mathbf{a}_l)]$，其中$a_l$為Foward時輸入此Layer的值，並令$a_0 = x$，而$\odot$則是Elementwise product，而

更新則是透過$w_l = w_l - \Delta_l * a_{l-1}$來進行。以下為透過圖1為例子作為說明：

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z3} \frac{\partial z_3}{\partial w_3} = -(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}})(\hat{y}(1-\hat{y}))a_2 = (\hat{y} - y)a_2$$

對於最後一層的w_3對Loss Function的偏微分計算如上所示，由於BCELoss微分後的分母與Sigmoid微分後相乘剛好可以化簡，因此我們可以得到$\Delta_3 = (\hat{y} - y)$。但若不是用這個組合，也只需計算個別計算$\frac{\partial L}{\partial \hat{y}}$與$\frac{\partial \hat{y}}{\partial z_3}$再相乘即可。

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z3} \frac{\partial z_3}{\partial a_2} \frac{\partial a_2}{\partial z_2} \frac{\partial z_2}{\partial w_2} = (\hat{y} - y)w_3 \odot (a^2 \odot (1 - a^2))a^1$$
$$= ((w_3)^T \Delta_2) \odot [a_2 \odot (1 - a_2)]a_1$$

倒數第二層的w_2對於Loss Function的偏微分為$((w_3)^T \Delta_2) \odot [a_2 \odot (1 - a_2)]a_1$，我們可以發現得出來的結論中在$a_1$前面那項剛好就是$((w_3)^T \Delta_2) \odot [a_2 \odot (1 - a_2)]$，我們將其定為$\Delta_2$。

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z3} \frac{\partial z_3}{\partial a_2} \frac{\partial a_2}{\partial z_2} \frac{\partial z_2}{\partial a1} \frac{\partial a1}{\partial z1} \frac{\partial z1}{\partial x1}$$
$$= (\hat{y} - y)w_3 \odot (a^2 \odot (1 - a^2))w_2 \odot (a^1 \odot (1 - a^1))a_0$$
$$= ((w_2)^T \Delta_2) \odot [a_1 \odot (1 - a_1)]a_0$$

倒數第三層也就是w_1對於Loss Function的偏微分為$((w_2)^T \Delta_2) \odot [a_1 \odot (1 - a_1)]a_0$，因此$\Delta_1$也與上面推導的結果一樣，就是$((w_2)^T \Delta_2) \odot [a_1 \odot (1 - a_1)]$

由此方法我們在計算時就能減少許多重複計算的部分，在程式的撰寫上也更有彈性，讓使用者可以自由的設定模型的參數與想要層數上，不需再額外推導繁瑣的更新公式。

2.3.2 Implementation

在實作上，建立模型需要先定義此模型的Forward 與Backward 的方式，如程式碼6所示，Backward 就如在Method 所提的方式，一開始的Δ_L 是由Loss Function 所回傳

的$\frac{\partial L}{\partial \hat{y}}$ 值，進入迴圈後會將Δ_L乘上$\frac{\partial \hat{y}}{z_3}$，並依據規則依序計算出其他的$\Delta_l$，最後再由程式碼7中的方式，將每一層linear的wieght與bias更新，以此就能夠訓練整個模型。

```
class Model(nn.module):
    def __init__(self, input_size, output_size, activation, hidden_size):
        super().__init__()
        if activation in nn.activation_functions_list:
            self.act = eval(f"nn.{activation}")
        else:
            raise ValueError(f"Activation function {activation} is not supported")
        self.net = [
            nn.Linear(input_size, hidden_size),
            self.act(),
            nn.Linear(hidden_size, hidden_size),
            self.act(),
            nn.Linear(hidden_size, output_size),
            nn.sigmoid(),
        ]
        self.delta = np.empty(len(self.net) + 1, dtype = object)
    def forward(self, x):
        for layer in self.net:
            x = layer.forward(x)
        return x
    def backward(self, delta, y_pred):
        self.delta[-1] = delta
        for i in range(len(self.net) - 1, -1, -1):
            if isinstance(self.net[i], nn.layer):
                self.delta[i] = self.net[i].backward(self.delta[i + 1])
            elif isinstance(self.net[i], nn.activation):
                if i + 1 >= len(self.net):
                    self.delta[i] = self.delta[i + 1] * self.net[i].backward(y_pred)
                else:
                    self.delta[i] = self.delta[i + 1] * self.net[i].backward(self.net[
    i + 1].a)
            else:
                assert False, "Unknown layer type"
```

Listing 6: Backpropagation

```
class SGD():
    def __init__(self, model, lr = 0.001):
        self.lr = lr
        self.model = model
    def step(self):
```

```
        for i, layer in enumerate(self.model.net):
            if isinstance(layer, nn.layer):
                layer.w -= self.lr * (self.model.delta[i + 1] @ layer.a.T)
                layer.b -= self.lr * self.model.delta[i + 1]
```

Listing 7: SGD

3 Results of your testing

3.1 Hyperparameter Settings

以下為本次實驗的超參數設定:

- **Activation Function**: Sigmoid
- **Loss Function**: Binary Cross Entropy
- **Optimizer**: SGD
- **NN Architecture**: $x \to linear(2, 32) \xrightarrow{\sigma} linear(32, 32) \xrightarrow{\sigma} linear(32, 1) \xrightarrow{\sigma} \hat{y}$
- **Epoch**: 5000
- **Learning Rate**: $5 * 10^{-2}$

3.2 Screenshot and comparison figure

實驗結果如表1所示，可見模型確實成功訓練並完成了此分類的任務。

3.3 Show the accuracy of your prediction

實驗結果如表2所示，第一個row為兩個任務各自的Epoch與Loss的關係，第二個row則是部分擷取最後九個點的預測結果與對於整個資料集的預測準確率，我們可以觀察到Linear的結果相較於XOR的結果更靠近0或1，可見XOR的任務較為困難。

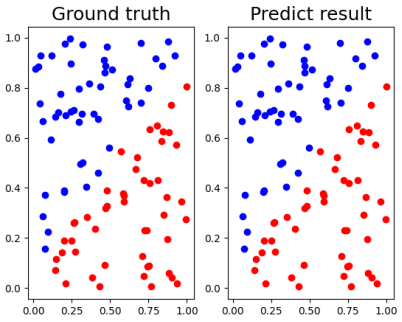
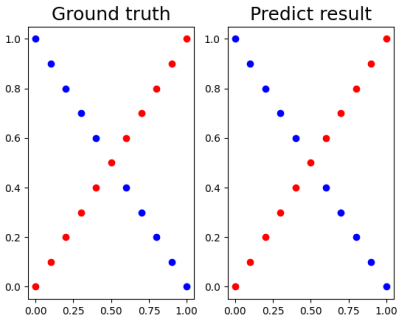
Linear	XOR
	

Table 1: Result

Linear	XOR
Epoch 500/5000 Loss: 0.05663244 Epoch 1000/5000 Loss: 0.03983622 Epoch 1500/5000 Loss: 0.03402205 Epoch 2000/5000 Loss: 0.03133604 Epoch 2500/5000 Loss: 0.02993964 Epoch 3000/5000 Loss: 0.02917074 Epoch 3500/5000 Loss: 0.02873435 Epoch 4000/5000 Loss: 0.02848248 Epoch 4500/5000 Loss: 0.02833573 Epoch 5000/5000 Loss: 0.02824974	Epoch 500/5000 Loss: 0.31611481 Epoch 1000/5000 Loss: 0.23784658 Epoch 1500/5000 Loss: 0.20701075 Epoch 2000/5000 Loss: 0.19210126 Epoch 2500/5000 Loss: 0.18418954 Epoch 3000/5000 Loss: 0.17978816 Epoch 3500/5000 Loss: 0.17727625 Epoch 4000/5000 Loss: 0.17582199 Epoch 4500/5000 Loss: 0.17497311 Epoch 5000/5000 Loss: 0.17447523
[6.13203018e-07] [3.63784915e-01] [8.20798810e-01] [1.06650747e-06] [1.02071072e-02] [2.62242979e-06] [1.39522461e-07] [9.99971459e-01] [1.62103941e-07]] Final Accuracy: 1.00	[0.53532701] [0.15402676] [0.79981006] [0.07087889] [0.93510711] [0.03745177] [0.97180459] [0.02357602] [0.98207442]] Final Accuracy: 1.00

Table 2: Prediction

3.4 Learning curve (loss, epoch curve)

實驗結果如表3所示，x軸方向為Epoch，y軸方向則是Loss的大小，由於兩者所使用的超參數皆相同，因此我們同樣可以從Loss下降的幅度與收斂值觀察XOR的任務相比與Linear確實是較為具有挑戰性的。

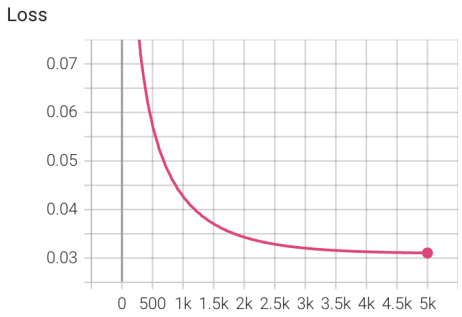
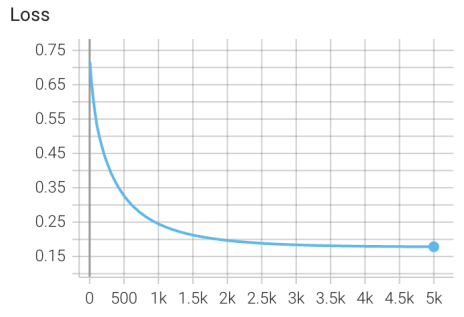
Linear	XOR
	

Table 3: Learning Curve

4 Discussion

4.1 Try different learning rates

以下實驗分別測試五種不同的Learning Rate，分別為：0.0005, 0.005, 0.05, 0.5, 1

4.1.1 Linear

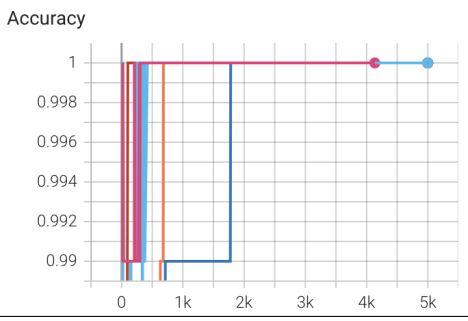
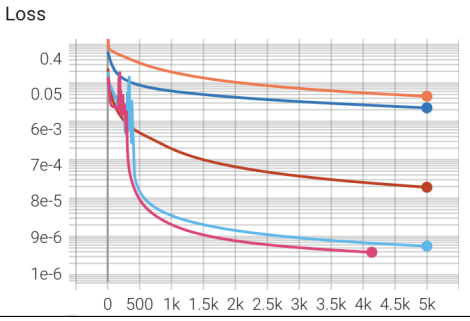
Accuracy ↑	Loss ↓
	

Table 4: Different Learning Rates on Linear

實驗結果如表4所示，其顏色與LR的對照為0.0005(Orange)，0.005(Blue)，0.05(Red)，0.5(Cyan), 1(Magenta)。我們可以觀察到在Accuracy的部分，不同的LR對Acc的差異不大，都為0.99至1.0，但在LR大於$5 * 10^{-}1$的結果在Loss的前期可以看到很明顯的震盪。

4.1.2 XOR

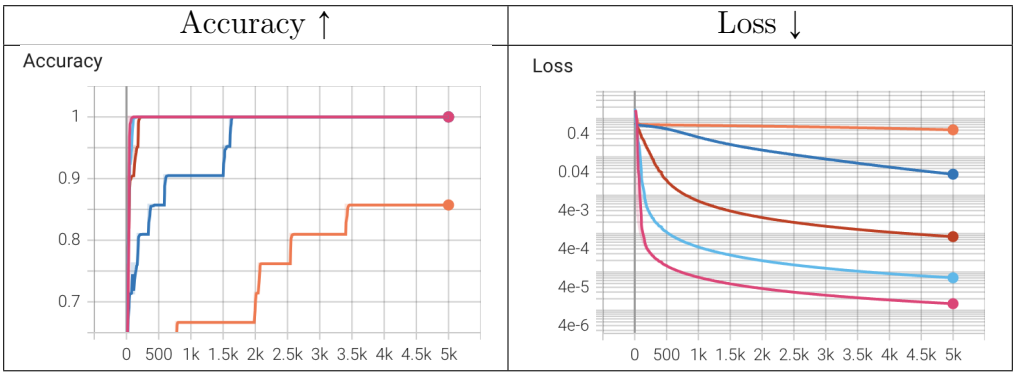


Table 5: Different Learning Rates on XOR

實驗結果如表5所示，其顏色與LR的對照為0.0005(Orange)，0.005(Blue)，0.05(Red)，0.5(Cyan), 1(Magenta)。我們可以觀察到除了RL為0.0005的情況，其餘不同的settings到最後的Acc皆為1，我認為是因為XOR這個任務的資料非常少，因此LR太小反而讓模型困在Local minimum進而提早收斂，才導致此結果。

4.2 Try different numbers of hidden units

以下實驗分別測試五種不同的hidden units，分別為：64，32，16，8，4，2，1

4.2.1 Linear

實驗結果如表6所示，其顏色與hidden units的對照為64(Orange)，32(Blue)，16(Red)，8(Cyan)，4(Magenta)，2(Green)，1(Gray)，從實驗中可以發現由於Linear是一個非常簡單且線性可分的問題，只要一個hidden layer就能夠完成分類的任務，因此不管在Acc與Loss上選擇不同hidden unit的差異不大。

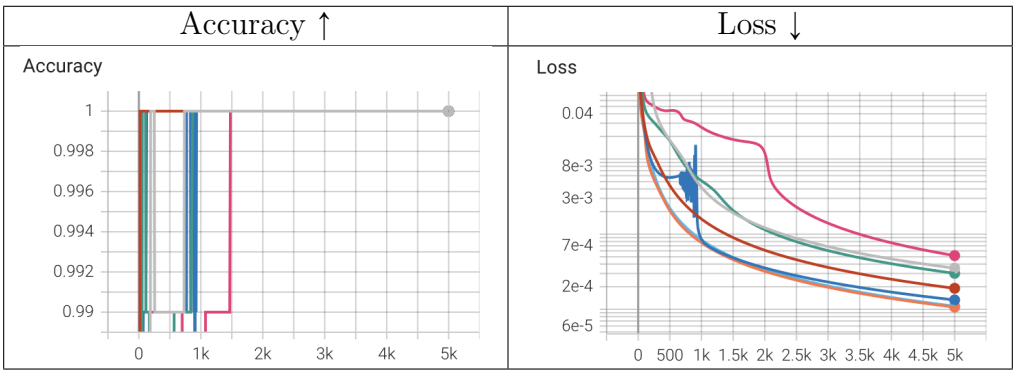


Table 6: Different hidden units on Linear

4.2.2 XOR

實驗結果如表7所示，其顏色與hidden units的對照為64(Orange)，32(Blue)，16(Red)，8(Cyan)，4(Magenta)，2(Green)，1(Gray)，從實驗中可以發現XOR的問題除了hidden unit為1的情況，其餘不管是在Acc與Loss皆差異不大。

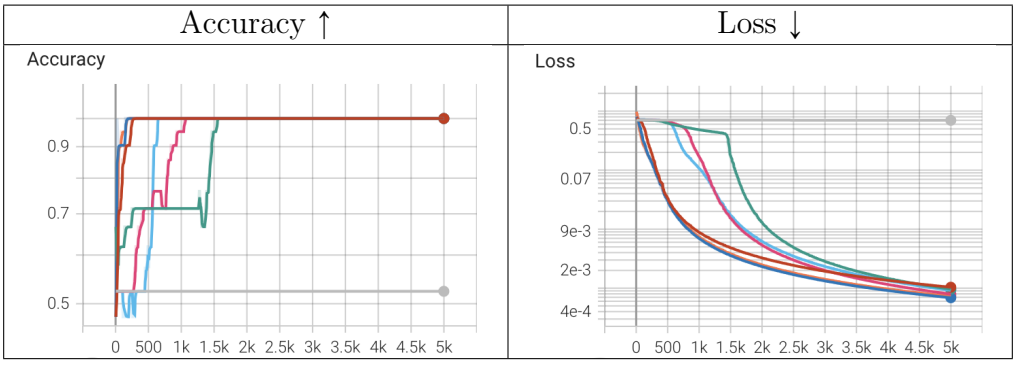


Table 7: Different hidden units on XOR

4.3 Try without activation functions

4.3.1 Linear

實驗結果如表8所示，有activation function(Orange)，沒有activation function(Blue)，從實驗中結果我發現對於Linear這種線性可分的任務有無使用activation function結果差異不大，但不使用activation function所產生的結果可能會較不穩定。

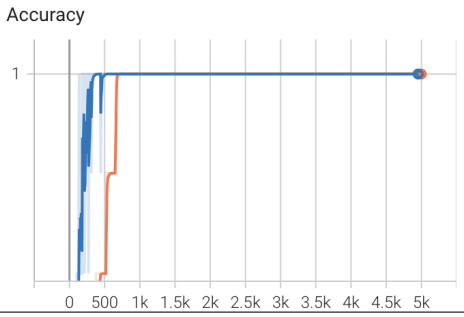
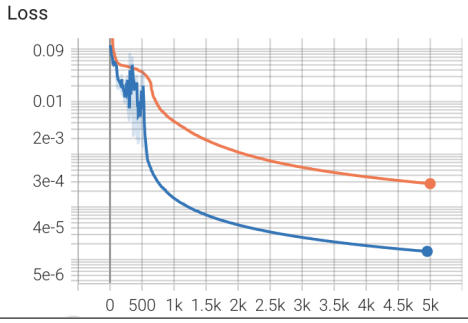
Accuracy ↑	Loss ↓
	

Table 8: Comparison of Using Activation Function on Linear

4.3.2 XOR

實驗結果如表9所示，有activation function(Orange)，沒有activation function(Blue)，從實驗中結果我發現對於XOR這種線性不可分的問題，若不使用activation function，不管如何運算其輸出都還是輸入的線性轉換，因此無法解決這種問題。

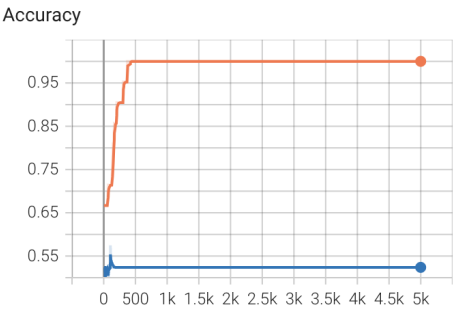
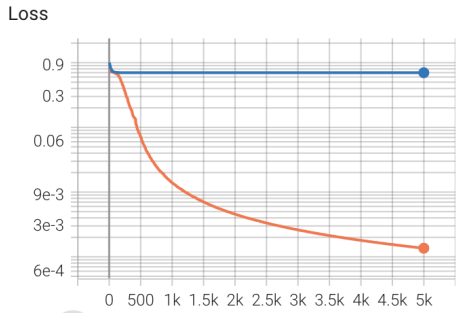
Accuracy ↑	Loss ↓
	

Table 9: Comparison of Using Activation Function on XOR

5 Extra

5.1 Implement different optimizers

在本次實驗中，除了原始的SGD外，我還另外實作了SGDM(SGD with Momentum)、RMSProp與Adam，請見A。

5.1.1 Linear

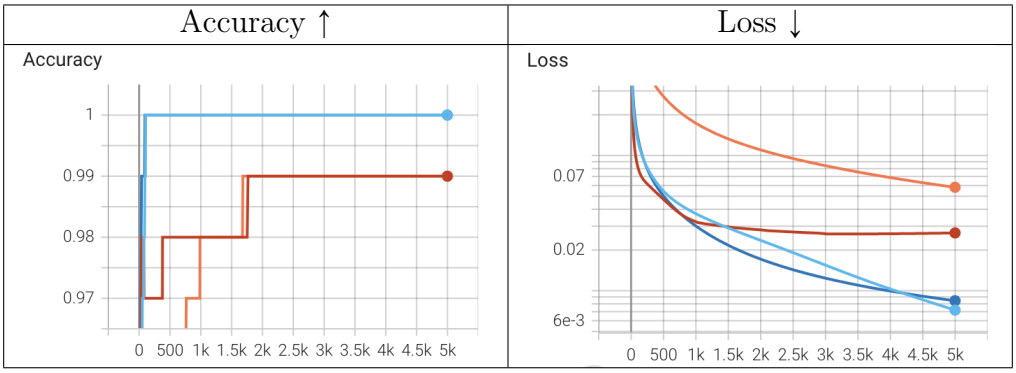


Table 10: Different Optimizers on Linear

實驗結果如表10分別為SGD(Orange)、SGDM(Blue)、RMSProp(Red)、Adam(Cyan)，可以觀察到從SGD更換成其他Optimizer皆能有效的提升模型更新的效率，在Linear這個任務中，使用Adam與SGDM的效果是最好的。

5.1.2 XOR

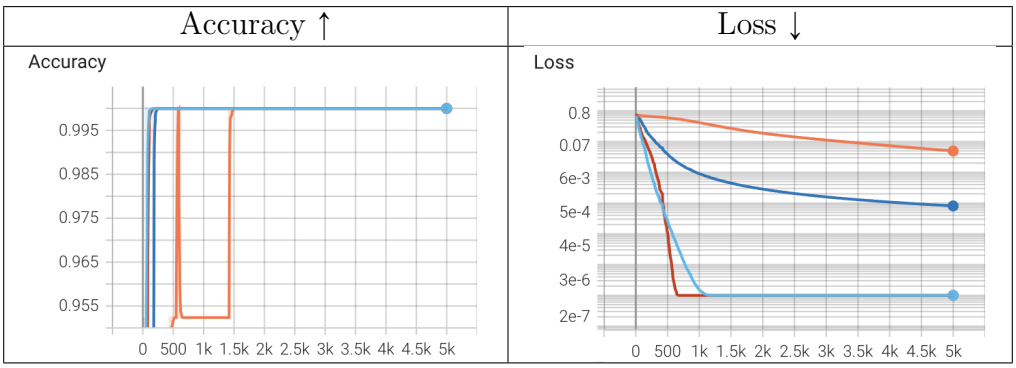


Table 11: Different Optimizers on Linear

實驗結果如表11分別為SGD(Orange)、SGDM(Blue)、RMSProp(Red)、Adam(Cyan)，可以觀察到從SGD更換成其他Optimizer皆能有效的提升模型更新的效率，在XOR這個任務中，使用Adam與RMSProp的效果最好的。

5.2 Implement different activation functions

在本次實驗中，除了原始的sigmoid外，我還另外實作了ReLU、LeakyReLU與Tanh，實作部分請參見B。

5.2.1 Linear

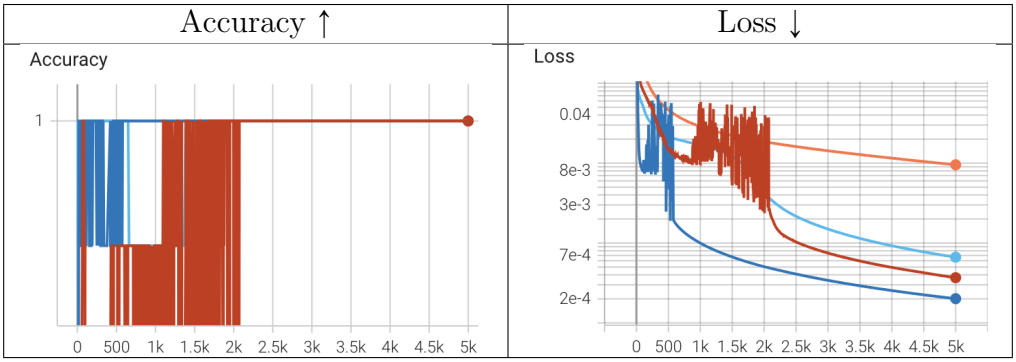


Table 12: Different Activation Function on Linear

實驗結果如表12分別為sigmoid(Orange)、ReLU(Blue)、LeakyReLU(Red)、Tanh(Cyan)，可以觀察到從sigmoid更換成其他activation function皆能有效的使loss下降，但也可以發現使用ReLU與LeakyReLU都會在前期產生不小的震盪，後期則會趨於穩定，最後則是ReLU的Loss最低。

5.2.2 XOR

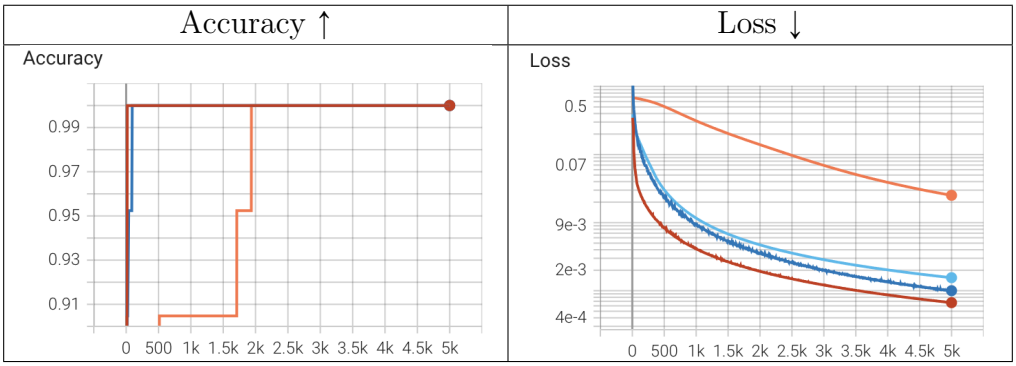


Table 13: Different Activation Function on XOR

實驗結果如表13分別為sigmoid(Orange)、ReLU(Blue)、LeakyReLU(Red)、Tanh(Cyan)，可以觀察到從sigmoid更換成其他activation function皆能有效的使loss下降，而最後效果最好的則是LeakyReLU。

5.3 Implement convolutional layers (Not Finished Yet)

在本實驗中，我嘗試實作了Conv1d的Forward，透過對於kernel對輸入的點進行運算並加總，但對於Backward的實作上遇到一些困難因此無法完成。

```
class Simple_Conv1d(layer):
    def __init__(self, kernel_size = 3):
        self.w = np.random.randn((kernel_size, 1))
        self.b = np.random.randn(1)
        self.a = None
    def forward(self, x):
        self.a = x
        h = np.zeros(x.shape[-2] - self.w.shape[0] + 1)
        for i in range(h.shape[0]):
            h[i] = np.sum(x[i:i + self.w.shape[0]] * self.w)
        return h + self.b
    def backward(self, delta):
        pass
```

Listing 8: Convolution Layer

5.4 Multiclass Architecture

在實作上，由於本次實驗的兩個任務皆為一個二分類問題，因此在結果上只需要輸出為一個神經元$\hat{y}$，並以是否大於0.5作為依據來判斷是某個class即可。然而若遇到其他多分類的任務此架構便不再適用。因此為了讓模型能夠加彈性，我另外實作了multiclass的版本，將輸出的結果變為一個vector，並透過Softmax將其轉換此輸入為某個class的機率，並透過Cross Entropy來計算Loss來更新神經網路，Softmax與Cross Entropy的實作如程式碼9與10。

在實作時，由於Softmax的計算為單調性的，因此預測出label實際上就是值最大的那個index，因此不需要真的經過Softmax，而實際上Softmax會直接放在CrossEntropy Loss中才會使用，會先將預測的結果做完Softmax後在與實

際label的one-hot encoding的結果計算Cross Entropy來計算Loss，而其對於前一層輸出的偏微分為$\frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{z} = (\hat{y} - y)$，剛好與過去使用的Sigmoid加上BCELoss的結果相同，其Neural Network的架構如程式碼11。

```python
class Softmax(activation):
    def forward(self, x):
        self.a = x
        exp_x = np.exp(x - np.max(x))
        return exp_x / np.sum(exp_x, axis = 0)

    def backward(self, x):
        return x * (1 - x)
```

Listing 9: Softmax

```python
class CrossEntropyLoss():
    def __init__(self):
        pass
    def __call__(self, y_pred, y_target, eps = 1e-8):
        y_target = one_hot_encoding(y_target, y_pred.shape[0]).T
        y_pred = np.exp(y_pred - np.max(y_pred)) / np.sum(np.exp(y_pred - np.max(
    y_pred)), axis = 0)
        y_pred = np.clip(y_pred, eps, 1 - eps)
        loss = -np.mean(y_target * np.log(y_pred))
        delta = (y_pred - y_target)
        return loss, delta
```

Listing 10: Cross Entropy Loss

```python
class Model(nn.module):
    def __init__(self, input_size, output_size, activation, hidden_size):
        super().__init__()
        if activation in nn.activation_functions_list:
            self.act = eval(f"nn.{activation}")
        else:
            raise ValueError(f"Activation function {activation} is not supported")
        self.net = [
            nn.Linear(input_size, hidden_size),
            self.act(),
            nn.Linear(hidden_size, hidden_size),
            self.act(),
            nn.Linear(hidden_size, output_size),
        ]
```

```
        self.delta = np.empty(len(self.net) + 1, dtype = object)

    def forward(self, x):
        for layer in self.net:
            x = layer.forward(x)
        return x

    def backward(self, delta, y_pred):
        self.delta[-1] = delta
        for i in range(len(self.net) - 1, -1, -1):
            if isinstance(self.net[i], nn.layer):
                self.delta[i] = self.net[i].backward(self.delta[i + 1])
            elif isinstance(self.net[i], nn.activation):
                if i + 1 >= len(self.net):
                    self.delta[i] = self.delta[i + 1] * self.net[i].backward(y_pred)
                else:
                    self.delta[i] = self.delta[i + 1] * self.net[i].backward(self.net[
i + 1].a)
            else:
                assert False, "Unknown layer type"
```

Listing 11: Neural Network with multiclass

A Optimizer

A.1 SGDM

```
class SGDM():
    def __init__(self, model, lr = 0.001, momentum = 0.9):
        assert isinstance(model, nn.module), "model should be an instance of module"
        assert 0 < momentum < 1, "momentum should be in (0, 1)"
        self.lr = lr
        self.momentum = momentum
        self.model = model
        self.v_w = []
        self.v_b = []
        for layer in self.model.net:
            if isinstance(layer, nn.layer):
                self.v_w.append(np.zeros_like(layer.w))
                self.v_b.append(np.zeros_like(layer.b))
            else:
                self.v_w.append(None)
                self.v_b.append(None)

    def step(self):
        for i, layer in enumerate(self.model.net):
            if isinstance(layer, nn.layer):
                self.v_w[i] = self.momentum * self.v_w[i] - self.lr * self.model.delta
    [i + 1] @ layer.a.T
                self.v_b[i] = self.momentum * self.v_b[i] - self.lr * self.model.delta
    [i + 1]
                layer.w += self.v_w[i]
                layer.b += self.v_b[i]
```

Listing 12: SGDM

A.2 RMSProp

```
class RMSprop():
    def __init__(self, model, lr = 0.001, alpha = 0.9, epsilon = 1e-8):
        assert isinstance(model, nn.module), "model should be an instance of module"
        assert 0 < alpha < 1, "alpha should be in (0, 1)"
        assert 0 < epsilon, "epsilon should be positive"
        self.lr = lr
        self.alpha = alpha
```

```
        self.epsilon = epsilon
        self.model = model
        self.v_w = []
        self.v_b = []
        for layer in self.model.net:
            if isinstance(layer, nn.layer):
                self.v_w.append(np.zeros_like(layer.w))
                self.v_b.append(np.zeros_like(layer.b))
            else:
                self.v_w.append(None)
                self.v_b.append(None)

    def step(self):
        for i, layer in enumerate(self.model.net):
            if isinstance(layer, nn.layer):
                self.v_w[i] = self.alpha * self.v_w[i] + (1 - self.alpha) * (self.
    model.delta[i + 1] @ layer.a.T) ** 2
                self.v_b[i] = self.alpha * self.v_b[i] + (1 - self.alpha) * (self.
    model.delta[i + 1]) ** 2
                layer.w -= self.lr * self.model.delta[i + 1] @ layer.a.T / (np.sqrt(
    self.v_w[i]) + self.epsilon)
                layer.b -= self.lr * self.model.delta[i + 1] / (np.sqrt(self.v_b[i]) +
     self.epsilon)
```

Listing 13: RMSProp

A.3 Adam

```
class Adam():
    def __init__(self, model, lr = 0.001, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-8):
        assert isinstance(model, nn.module), "model should be an instance of module"
        assert 0 < beta1 < 1 and 0 < beta2 < 1, "beta1 and beta2 should be in (0, 1)"
        assert 0 < epsilon, "epsilon should be positive"
        self.lr = lr
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.model = model
        self.m_w, self.v_w, self.m_b, self.v_b = [], [], [], []
        for layer in self.model.net:
            if isinstance(layer, nn.layer):
                self.m_w.append(np.zeros_like(layer.w))
```

```
                self.v_w.append(np.zeros_like(layer.w))
                self.m_b.append(np.zeros_like(layer.b))
                self.v_b.append(np.zeros_like(layer.b))
            else:
                self.m_w.append(None)
                self.v_w.append(None)
                self.m_b.append(None)
                self.v_b.append(None)
        self.t = 0

    def step(self):
        self.t += 1
        for i, layer in enumerate(self.model.net):
            if isinstance(layer, nn.layer):
                self.m_w[i] = self.beta1 * self.m_w[i] + (1 - self.beta1) * (self.
    model.delta[i + 1] @ layer.a.T)
                self.v_w[i] = self.beta2 * self.v_w[i] + (1 - self.beta2) * (self.
    model.delta[i + 1] @ layer.a.T) ** 2
                m_hat_w = self.m_w[i] / (1 - self.beta1 ** self.t)
                v_hat_w = self.v_w[i] / (1 - self.beta2 ** self.t)
                layer.w -= self.lr * m_hat_w / (np.sqrt(v_hat_w) + self.epsilon)

                self.m_b[i] = self.beta1 * self.m_b[i] + (1 - self.beta1) * self.model
    .delta[i + 1]
                self.v_b[i] = self.beta2 * self.v_b[i] + (1 - self.beta2) * (self.
    model.delta[i + 1]) ** 2
                m_hat_b = self.m_b[i] / (1 - self.beta1 ** self.t)
                v_hat_b = self.v_b[i] / (1 - self.beta2 ** self.t)
                layer.b -= self.lr * m_hat_b / (np.sqrt(v_hat_b) + self.epsilon)
```

Listing 14: Adam

B Activation Function

B.1 ReLU

```
class ReLU(activation):
    def forward(self, x):
        self.a = x
        return np.maximum(0, x)
    def backward(self, x):
        return (x > 0).astype(int)
```

Listing 15: ReLU

B.2 LeakyReLU

```
class LeakyReLU(activation):
    def __init__(self, alpha = 0.02):
        self.alpha = alpha
    def forward(self, x):
        self.a = x
        return np.maximum(self.alpha * x, x)
    def backward(self, x):
        return (x > 0).astype(int) + self.alpha * (x <= 0).astype(int)
```

Listing 16: LeakyReLU

B.3 Tanh

```
class Tanh(activation):
    def forward(self, x):
        self.a = x
        return np.tanh(x)
    def backward(self, x):
        return 1 - x ** 2
```

Listing 17: Tanh