DL Lab2: forward pass & back-propagation

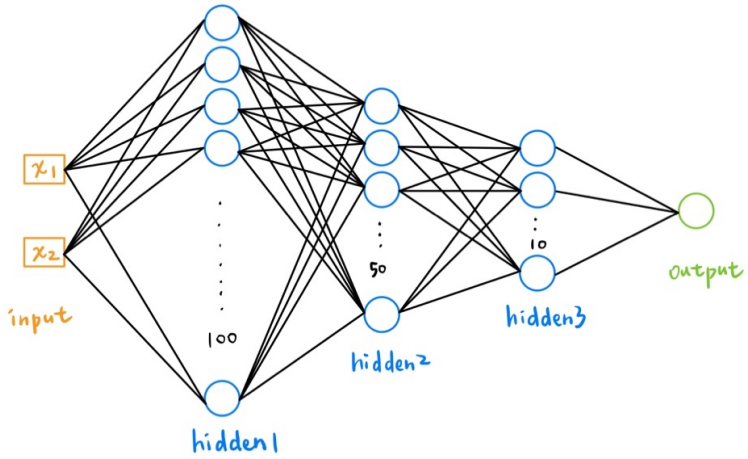
1. Implement a simple neural network with three hidden layers (with 100 nodes in hidden layer 1 , 50 nodes in hidden layer 2 and 10 nodes in hidden layer 3).

```
def __init__(self, num_step=6000, print_interval=100, learning_rate=(1e-2)*5):
    self.num_step = num_step
    self.print_interval = print_interval
    self.learning_rate = learning_rate

    input_dim = 2
    hidden1_dim = 100
    hidden2_dim = 50
    hidden3_dim = 10
    output_dim = 1

    self.hidden1_weights = np.random.randn(input_dim, hidden1_dim)
    self.hidden1_bias = np.zeros((1, hidden1_dim))
    self.hidden2_weights = np.random.randn(hidden1_dim, hidden2_dim)
    self.hidden2_bias = np.zeros((1, hidden2_dim))
    self.hidden3_weights = np.random.randn(hidden2_dim, hidden3_dim)
    self.hidden3_bias = np.zeros((1, hidden3_dim))
    self.output_weights = np.random.randn(hidden3_dim, output_dim)
    self.output_bias = np.zeros((1, output_dim))
```

上述程式碼定義了具有一個輸入層、三個隱藏層和一個輸出層的神經網路基礎架構，並設定了一些基本的參數，包括訓練的迭代次數（num_step）、列印訓練次數的間隔（print_interval）以及學習率（learning_rate）。



- 每個隱藏層的權重（weights）使用隨機數值 randn 正態分佈初始化：
  如果所有權重都初始化為同一個數值（例如 0），那麼在反向傳播過程中，每個權重的更新都會是一樣的。這意味著每個神經元在每層中都會學到相同的特徵，這就失去了多神經元的意義。隨機初始化權重可以打破這種對稱性，使每個神經元能學到不同的特徵。
- 偏置（bias）是全為 1 的陣列：
  初始化偏置為 1（或其他非零值）可能有助於神經元在初期更容易被激活，尤其是使用某些激活函數（如 ReLU）時。這可以在初期的訓練中使神經元更“活躍”。

2. You must use the back-propagation algorithm in this NN and build it from scratch. Only Numpy and other Python standard libraries are allowed.

訓練過程：

```
def train(self, inputs, labels):
    # make sure that the amount of data and label is match
    assert inputs.shape[0] == labels.shape[0]

    n = inputs.shape[0]

    for epoch in range(self.num_step):
        total_loss = 0

        for idx in range(n):
            # (1) Forward pass
            self.output = self.forward(inputs[idx : idx + 1, :])
            # (2) Compute loss
            self.error = self.output - labels[idx : idx + 1, :]
            loss = 0.5 * np.square(self.error).sum()
            total_loss += loss
            # (3) Propagate gradient backward to the front
            self.backward()

        if epoch % self.print_interval == 0:
            average_loss = total_loss / n
            print(f"Epoch {epoch}: Average Loss = {average_loss}")
            self.test(inputs, labels)

    print("Training finished")
    self.test(inputs, labels)
```

(1) 進行前向傳播計算。

(2) 計算輸出和實際結果之間的誤差（error），並且計算損失（loss）。這裡使用均方誤差（MSE），乘以 0.5 是為了在對 MSE 進行微分時簡化計算。

$$loss = \frac{1}{2}\sum(y_i - \hat{y}_i)^2$$

(3) 進行反向傳播，計算梯度並更新權重和偏置。

前向傳播：神經網路計算輸入數據到輸出的過程

```
def forward(self, inputs):
    self.inputs = inputs

    hidden1_output = sigmoid(np.dot(inputs, self.hidden1_weights) + self.hidden1_bias)
    hidden2_output = sigmoid(np.dot(hidden1_output, self.hidden2_weights) + self.hidden2_bias)
    hidden3_output = sigmoid(np.dot(hidden2_output, self.hidden3_weights) + self.hidden3_bias)
    self.hidden1_output = hidden1_output
    self.hidden2_output = hidden2_output
    self.hidden3_output = hidden3_output

    output = sigmoid(np.dot(hidden3_output, self.output_weights) + self.output_bias)
    self.output = output

    return output
```

(1) 計算每個隱藏層和輸出層的線性組合輸出：

$$z^{(l)} = a^{(l-1)}W^{(l)} + b^{(l)}$$

$z^{(l)}$是第l層的線性組合輸出
$a^{(l-1)}$是前一層$l-1$的激活輸出
$W^{(l)}$是第l層的權重
$b^{(l)}$是第l層的偏置

(2) 將線性組合輸出通過一個激活函數（Sigmoid）：

$$a^{(l)} = f\left(z^{(l)}\right)$$

$a^{(l)}$是第l層的激活輸出
f是激活函數

反向傳播：用於調整神經網絡中的權重和偏置的算法，基於訓練數據的實際輸出和預期輸出之間的誤差。

```
def backward(self):
    delta_output = self.error * der_sigmoid(self.output)
    d_output_weights = np.dot(self.hidden3_output.T, delta_output)

    delta_hidden3 = np.dot(delta_output, self.output_weights.T) * der_sigmoid(self.hidden3_output)
    d_hidden3_weights = np.dot(self.hidden2_output.T, delta_hidden3)

    delta_hidden2 = np.dot(delta_hidden3, self.hidden3_weights.T) * der_sigmoid(self.hidden2_output)
    d_hidden2_weights = np.dot(self.hidden1_output.T, delta_hidden2)

    delta_hidden1 = np.dot(delta_hidden2, self.hidden2_weights.T) * der_sigmoid(self.hidden1_output)
    d_hidden1_weights = np.dot(self.inputs.T, delta_hidden1)

    self.output_weights -= self.learning_rate * d_output_weights
    self.output_bias -= self.learning_rate * delta_output
    self.hidden3_weights -= self.learning_rate * d_hidden3_weights
    self.hidden3_bias -= self.learning_rate * delta_hidden3
    self.hidden2_weights -= self.learning_rate * d_hidden2_weights
    self.hidden2_bias -= self.learning_rate * delta_hidden2
    self.hidden1_weights -= self.learning_rate * d_hidden1_weights
    self.hidden1_bias -= self.learning_rate * delta_hidden1
```

(1) 計算誤差：計算後一層的誤差乘以後一層權重的轉置，再乘上自己這層的輸出對其總輸入的導數。

$$\delta_{output} = error \times back_weights^{T} \times [output \times (1 - output)]$$

進行前向傳播時，會使用前一層的輸出（一個列向量）乘以權重矩陣。因此在反向傳播時，需要進行相反的操作，將後一層的誤差（一個列向量）乘以權重矩陣的轉置。

(2) 計算權重的梯度：前一層輸出值的轉置乘以自己這層的誤差。

$$\frac{\partial loss}{\partial W} = previous\_output^{T} \times \delta_{output}$$

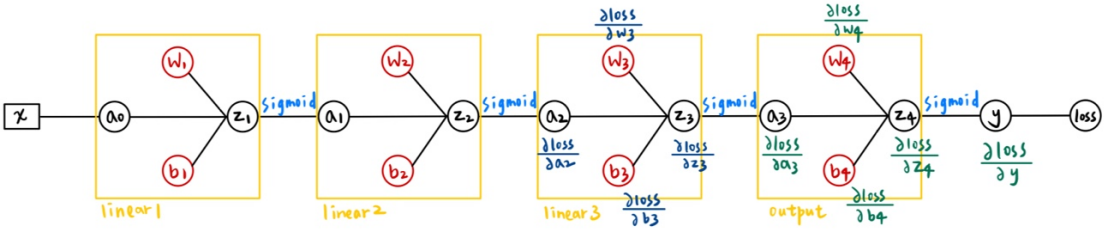
(3) 計算偏置的梯度，即 δ_{output}。因為偏置的輸入可以被視為 1，表示計算出來的誤差直接關聯到偏置上。

(4) 更新權重和偏置：將原本的權重/偏置減掉權重/偏置乘以學習率的結果

$$W = W - \eta \times \frac{\partial loss}{\partial W}$$

$$b = b - \eta \times \frac{\partial loss}{\partial b}$$

(5) 計算圖（Computing Graph）



數學推導過程：

對輸出層做計算：

delta_output

$loss = \frac{1}{2}(y-\hat{y})^2$

$\frac{\partial loss}{\partial y} = y - \hat{y}$

$\frac{\partial loss}{\partial z_4} = \frac{\partial loss}{\partial y} \times \frac{\partial y}{\partial z_4}$

$= (y-\hat{y})\left(\frac{\partial sig(z_4)}{\partial z_4}\right)$

$= (y-\hat{y})(sig(z_4))(1-sig(z_4))$

$= \underbrace{(y-\hat{y})}_{error}\ \underbrace{(y)(1-y)}_{der\_sigmoid}$

d_output_weights

$\frac{\partial loss}{\partial w_4} = \frac{\partial loss}{\partial z_4} \times \boxed{\frac{\partial z_4}{\partial w_4}}$ a_3 (hidden3_output)

d_output_bias

$\frac{\partial loss}{\partial b_4} = \frac{\partial loss}{\partial z_4} \times \boxed{\frac{\partial z_4}{\partial b_4}}$ 1

$\frac{\partial loss}{\partial a_3} = \frac{\partial loss}{\partial z_4} \times \boxed{\frac{\partial z_4}{\partial a_3}}$ w_4 (output_weights)

對 hidden3 做計算：

delta_hidden3

$\frac{\partial loss}{\partial z_3} = \boxed{\frac{\partial loss}{\partial y} \times \frac{\partial y}{\partial z_4}} \times \frac{\partial z_4}{\partial a_3} \times \frac{\partial a_3}{\partial z_3}$

$\boxed{\frac{\partial loss}{\partial y} \times \frac{\partial y}{\partial z_4}}$ = delta_output, $\frac{\partial z_4}{\partial a_3}$ = w_4, $\frac{\partial a_3}{\partial z_3}$ = der_sigmoid

d_hidden3_weights

$\frac{\partial loss}{\partial w_3} = \frac{\partial loss}{\partial z_3} \times \boxed{\frac{\partial z_3}{\partial w_3}}$ a_2 (hidden2_output)

d_hidden3_bias

$\frac{\partial loss}{\partial b_3} = \frac{\partial loss}{\partial z_3} \times \boxed{\frac{\partial z_3}{\partial b_3}}$ 1

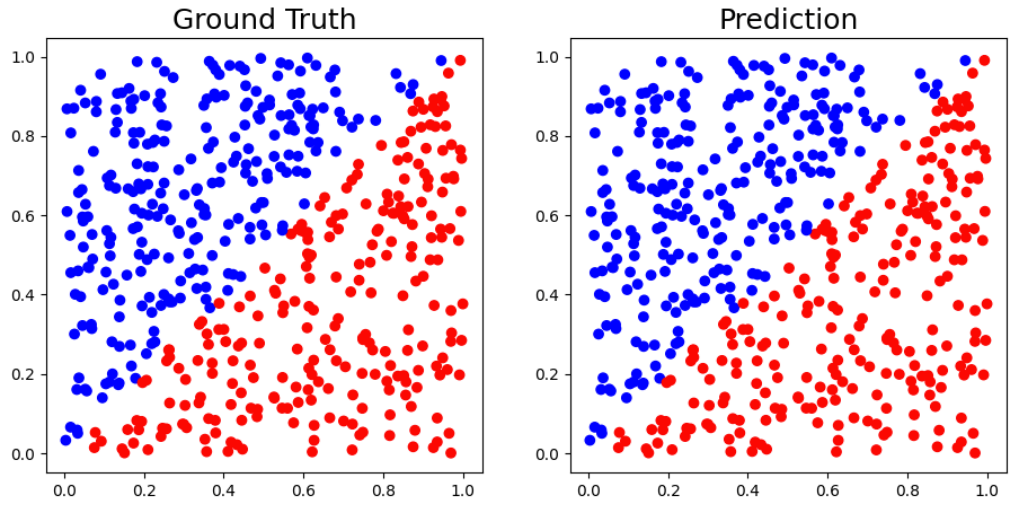
$\frac{\partial loss}{\partial a_2} = \frac{\partial loss}{\partial z_3} \times \boxed{\frac{\partial z_3}{\partial a_2}}$ w_3 (hidden3_weights)

對 hidden2 和 hidden1 的計算以此類推 ...

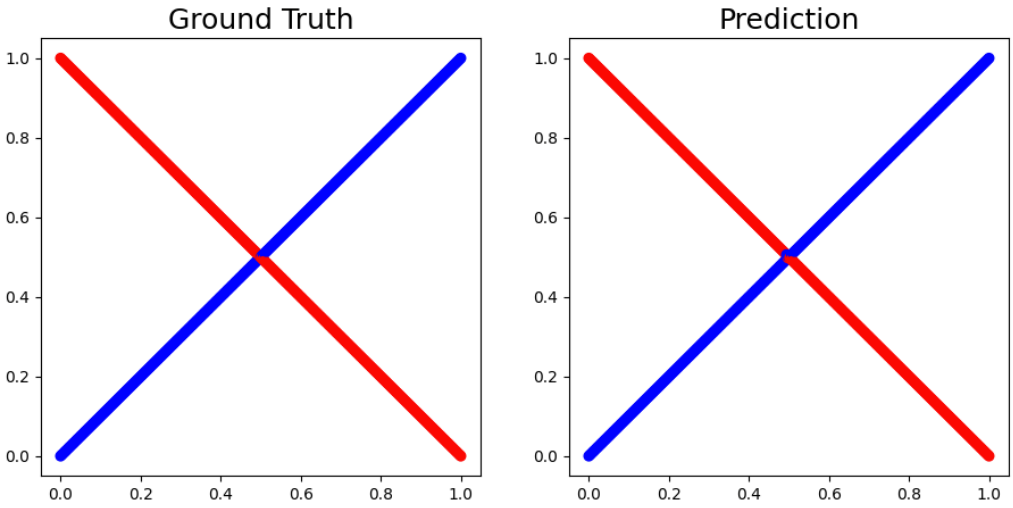
由上述數學推導可以得出程式碼的計算過程。

3. Plot your comparison between ground truth and the predicted result.

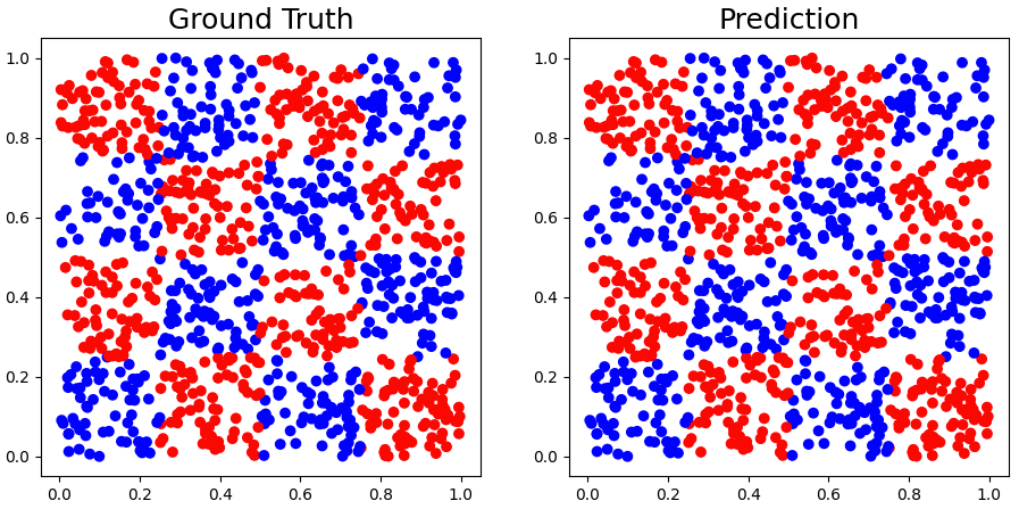
Linear



XOR



Chessboard



4. The training epochs is not restricted, but model performance will be evaluated.

方法一、Adaptive learning rate methods: AdaGrad

- ✓ **個別學習率調整**：它能夠為模型中的每個參數單獨調整學習率。對於出現頻率較高的特徵，它會降低學習率；對於出現頻率低的特徵，它會提高學習率（適合用於處理稀疏數據）。
- ✓ **自動調節**：不需要手動調整學習率。

(1) 在初始化的最後，針對每個隱藏層和輸出層的權重和偏置增加累積器：

```
# Initialize accumulators for AdaGrad
self.acc_hidden1_weights = np.ones_like(self.hidden1_weights)
self.acc_hidden1_bias = np.ones_like(self.hidden1_bias)
self.acc_hidden2_weights = np.ones_like(self.hidden2_weights)
self.acc_hidden2_bias = np.ones_like(self.hidden2_bias)
self.acc_hidden3_weights = np.ones_like(self.hidden3_weights)
self.acc_hidden3_bias = np.ones_like(self.hidden3_bias)
self.acc_output_weights = np.ones_like(self.output_weights)
self.acc_output_bias = np.ones_like(self.output_bias)
```

(2) 將反向傳播中原本的權重和偏置更新方法改成下方的程式碼：

- **累積梯度的平方**：將梯度的平方加到對應的累積器上。此為 AdaGrad 的核心思想，即保持對過去所有梯度平方的累積記錄，使其在初始階段迅速改進學習過程。

- **更新權重和偏置**：使用梯度和累積器更新權重和偏置。這裡的學習率被除以累積器的平方根（加上一個非常小的數，通常是 1e-8，以防止分母為 0）。對於那些在過去有較大梯度的參數，其學習率會較小；而對於那些在過去有較小梯度的參數，其學習率會較大。

```
self.acc_output_weights += d_output_weights ** 2
self.acc_output_bias += delta_output ** 2
self.output_weights -= self.learning_rate * d_output_weights / (np.sqrt(self.acc_output_weights) + 1e-8)
self.output_bias -= self.learning_rate * delta_output / (np.sqrt(self.acc_output_bias) + 1e-8)

self.acc_hidden3_weights += d_hidden3_weights ** 2
self.acc_hidden3_bias += delta_hidden3 ** 2
self.hidden3_weights -= self.learning_rate * d_hidden3_weights / (np.sqrt(self.acc_hidden3_weights) + 1e-8)
self.hidden3_bias -= self.learning_rate * delta_hidden3 / (np.sqrt(self.acc_hidden3_bias) + 1e-8)

self.acc_hidden2_weights += d_hidden2_weights ** 2
self.acc_hidden2_bias += delta_hidden2 ** 2
self.hidden2_weights -= self.learning_rate * d_hidden2_weights / (np.sqrt(self.acc_hidden2_weights) + 1e-8)
self.hidden2_bias -= self.learning_rate * delta_hidden2 / (np.sqrt(self.acc_hidden2_bias) + 1e-8)

self.acc_hidden1_weights += d_hidden1_weights ** 2
self.acc_hidden1_bias += delta_hidden1 ** 2
self.hidden1_weights -= self.learning_rate * d_hidden1_weights / (np.sqrt(self.acc_hidden1_weights) + 1e-8)
self.hidden1_bias -= self.learning_rate * delta_hidden1 / (np.sqrt(self.acc_hidden1_bias) + 1e-8)
```

(3) 相同 learning rate，但不同模型架構的比較：

Learning rate		0.05	
分類任務	Epoch	一般模型	加 AdaGrad
Linear	5000	99.89%	99.40%
XOR	5000	98.93%	97.93%
Chessboard	8000	98.25%	91.02%
收斂速度		快	慢
損失函數		震盪	平滑

一般的模型和增加 AdaGrad 的模型分別在 Linear, XOR, Chessboard 任務中，準確率達到差不多水準的 epoch 數：

- Linear：
  一般的模型：第 1000 個 epoch 的準確率 99.11%
  增加 AdaGrad 的模型：第 2200 個 epoch 的準確率 99%
- XOR：
  一般的模型：第 700 個 epoch 的準確率 97.16%
  增加 AdaGrad 的模型：第 2500 個 epoch 的準確率 97.15%
- Chessboard：
  一般的模型：第 800 個 epoch 的準確率 90.42%
  增加 AdaGrad 的模型：第 6800 個 epoch 的準確率 90.02%

從上述資訊可以比較兩著的收斂速度為「一般模型」較快，且最終的準確率亦為「一般模型」較高，因此自適應學習率方法 AdaGrad 較不適合此分類任務。

方法二、Learning rate schedules:

(1) 在初始化的參數中新增 decay_rate，用以設定學習率的衰減率。

```
def __init__(self, num_step=6000, print_interval=100, learning_rate=(1e-2)*65, decay_rate=0.95):
    self.num_step = num_step
    self.print_interval = print_interval
    self.learning_rate = learning_rate
    self.decay_rate = decay_rate
```

(2) 在訓練 epoch 的最後，設定每訓練 1000 個 epoch 就做學習率衰減，透過將原始的學習率乘上衰減率來進行更新。

```
if (epoch+1) % 1000 == 0:
    self.learning_rate *= self.decay_rate
    print(f"Epoch {epoch+1}: updated learning rate: {self.learning_rate}")
```

(3) 相同模型架構，但調整不同 learning rate 的比較：

Learning rate		0.05	0.065 (每 1000 epoch 乘以 0.95)
分類任務	Epoch	一般模型	
Linear	5000	99.89%	99.97%
XOR	5000	98.93%	99.06%
Chessboard	8000	98.25%	99.79%
收斂速度		慢	快

一般模型在調整不同 learning rate 的情況下，分別在 Linear, XOR, Chessboard 任務中，準確率達到差不多水準的 epoch 數：

- Linear：
  學習率固定：第 4300 個 epoch 的準確率 99.75%
  學習率衰減：第 300 個 epoch 的準確率 99.74%
- XOR：
  學習率固定：第 1900 個 epoch 的準確率 98.37%
  學習率衰減：第 1400 個 epoch 的準確率 98.34%
- Chessboard：
  學習率固定：第 3000 個 epoch 的準確率 95.45%
  學習率衰減：第 1900 個 epoch 的準確率 95.37%

從上述資訊可以比較兩著的收斂速度為「使用學習率衰減方法」的較快，且最終的準確率亦為「使用學習率衰減方法」的較高，因此對於某些分類任務，應在訓練過程中適時的調整學習率。

其模型成效包括：

- **加快收斂**：在訓練的初始階段使用較高的學習率可以加快收斂速度，快速接近最佳解。隨著訓練的進行，適時降低學習率可以使模型在接近最佳解時更加穩定，減少震盪。

- **提高模型性能**：合適的學習率衰減策略可以幫助找到更好的局部最佳解，甚至更接近全局最佳解，從而提高模型的準確性和性能。

- **防止過擬合**：過高的學習率可能會導致模型在訓練集上表現良好，但在驗證集或測試集上表現不佳，這是過擬合的表現。通過適當降低學習率，模型將更細微地調整其權重，有助於提高模型的泛化能力。