DL Lab4: 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).

由於我底下的 forward 寫法是用 np.dot，所以這邊第一層 hidden layer 要設置成 2x100，矩陣的長寬才會對應。

```
# 初始化權重和偏置
self.hidden1_weights = np.random.randn(2, 100) # 第一層輸入2，輸出100個隱藏節點
self.hidden1_bias = np.zeros((1, 100))

self.hidden2_weights = np.random.randn(100, 50) # 第二層輸入100，輸出50個隱藏節點
self.hidden2_bias = np.zeros((1, 50))

self.hidden3_weights = np.random.randn(50, 10) # 第三層輸入50，輸出10個隱藏節點
self.hidden3_bias = np.zeros((1, 10))

self.output_weights = np.random.randn(10, 1) # 最後一層輸入10，輸出1
self.output_bias = np.zeros((1, 1))
```

(2) The number of data points generated should > 100, and > 1000 for chessboard.

在 Linear、XOR 以及 Chessboard 這三種型態的資料，我分別生成了 1000、1000 和 2000 筆，我發現由於 Chessboard 的資料比較分散，對於神經網路來說比較難學習，因此他 train 所需要的資料量也會比較多。

```
Linear_data_train, Linear_label_train = GenData.fetch_data("Linear", 1000)
```

```
XOR_data_train, XOR_label_train = GenData.fetch_data("XOR", 1000)
```

```
Chessboard_data_train, Chessboard_label_train = GenData.fetch_data("Chessboard", 2000)
```

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

為了使模型更加靈活，我在 forward 那邊除了原本的矩陣相乘之外，我還加入了偏置矩陣，避免矩陣乘完之後的輸出為 0，通過 activation function 後結果還是 0，這樣會導致網路整體的學習率降低，因為運用到的神經元會比較少，因此加入偏置矩陣來使一些很小或者為 0 的特徵可以正常運作。

```
# 第一層
self.z1 = np.dot(inputs, self.hidden1_weights) + self.hidden1_bias
self.a1 = relu(self.z1)  # 激活函數
```

因此我在 backward 的時候，除了每一層的權重梯度之外，我還有運算了每一層的偏置梯度，最後再把梯度乘上學習率並更新權重和偏置。

```python
def backward(self, inputs):
    """Implementation of the backward pass.
    It should utilize the saved loss to compute gradients and update the network
    """
    """ FILL IN HERE """

    # 輸出層的梯度
    output_delta = self.error * der_sigmoid(self.output)  # 誤差 * 激活函數的導數
    output_grad_w = np.dot(self.a3.T, output_delta)
    output_grad_b = np.sum(output_delta, axis=0, keepdims=True)

    # 第三層的梯度
    a3_delta = np.dot(output_delta, self.output_weights.T) * der_relu(self.a3)
    hidden3_grad_w = np.dot(self.a2.T, a3_delta)
    hidden3_grad_b = np.sum(a3_delta, axis=0, keepdims=True)

    # 第二層的梯度
    a2_delta = np.dot(a3_delta, self.hidden3_weights.T) * der_relu(self.a2)
    hidden2_grad_w = np.dot(self.a1.T, a2_delta)
    hidden2_grad_b = np.sum(a2_delta, axis=0, keepdims=True)

    # 第一層的梯度
    a1_delta = np.dot(a2_delta, self.hidden2_weights.T) * der_relu(self.a1)
    hidden1_grad_w = np.dot(inputs.T, a1_delta)
    hidden1_grad_b = np.sum(a1_delta, axis=0, keepdims=True)

    # 更新權重和偏置
    self.output_weights -= self.learning_rate * output_grad_w
    self.output_bias -= self.learning_rate * output_grad_b

    self.hidden3_weights -= self.learning_rate * hidden3_grad_w
    self.hidden3_bias -= self.learning_rate * hidden3_grad_b

    self.hidden2_weights -= self.learning_rate * hidden2_grad_w
    self.hidden2_bias -= self.learning_rate * hidden2_grad_b

    self.hidden1_weights -= self.learning_rate * hidden1_grad_w
    self.hidden1_bias -= self.learning_rate * hidden1_grad_b
```

(4) Learning Rate Schedules and Adaptive Learning Rate Methods.

在學習率方面，我一開始是使用 0.001 當作固定的學習率，不過這樣會使模型的參數最後很難收斂，因此我引入了學習率下降的機制，我嘗試過每 100 個 epoch 下降為 0.99 倍，可是我發現這種固定倍率下降的方式對於我們網路的效果並不好，最後我想到使用 Exponential 的方式去做下降，在 Jupyter notebook 檔案內可以看到我把學習率用圖表的方式展示出來，並且模型的準確度提升蠻明顯的。

```python
def update_learning_rate(self, epoch):
    """根據epoch更新學習率"""
    self.decay_rate=0.001
    self.learning_rate = self.initial_learning_rate * np.exp(-self.decay_rate * epoch)
    self.learning_rate_arr.append(self.learning_rate)
```