# 深度學習

# HW7

學號 : B103012002

姓名 : 林凡皓

# 一、 Linear layer

- Linear.forward :
  1. 解題思路 ： 先利用 view( )將輸入 x 變成 shape 為(N, D)的 tensor。接著透過 forward propagation 的公式 $out = xW + b$ 來計算 forward propagation 的輸出結果。
  2. 執行結果 ：

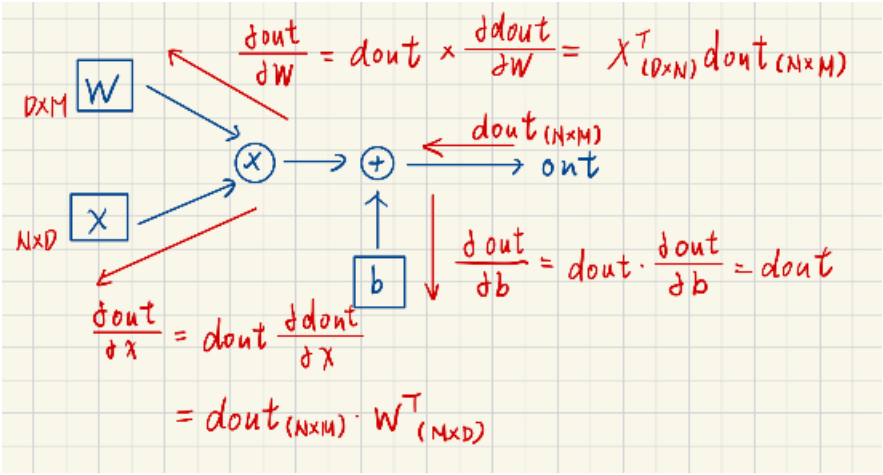
```
Testing Linear.forward function:
difference:  3.683042917976506e-08
```

  由此結果可以看出，計算結果與預期結果相差很小，代表說這個 function 功能正確。

  3. 額外討論 ：

  Forward propagation 的公式為 $out = xW + b$ 。之所以是 xW 是因為考慮到 tensor 的形狀。out、x、W 的形狀分別為(N, M)、(N, D)、(D, M)，因此透過 xW 的形狀為(N, M)滿足 out 的形狀可以得知，矩陣乘法順序應該為 xW。

- Linear.backward :
  1. 解題思路 ： 透過 computational graph 來幫助我們計算微分，computational graph 如下




  根據上圖推倒結果，完成 dx、dw、db 即可。

  2. 執行結果 ：

```
Testing Linear.backward function:
dx error:  5.540736853994794e-10
dw error:  3.964520455338778e-10
db error:  5.373171200544344e-10
```

  由上面結果可以看出，跟預期結果相比，誤差很小，代表說此

function 功能正確。

3. 額外討論 ：

透過 computational graph 計算出來的 dx 為 $dout \times W^T$ ，這其實是 reshape 後的 dx，並不是我們要的。因此計算出此結果後，還需要透過 reshape 將形狀變成 (N, d1, …, d_k)。

## 二、 ReLU activation

- ReLU.forward :

1. 解題思路 ：

透過 torch.relu( )即可完成此題。

2. 執行結果 ：

```
Testing ReLU.forward function:
difference:  4.5454545613554664e-09
```

由上面結果可以看出，跟預期結果相比，誤差很小，代表說此 function 功能正確。

- ReLU.backward :

1. 解題思路 ：

由 $ReLU(x) = \max(0, x)$可以知道，對此函數做微分的結果為，當 x > 0，微分結果會是 upstream derivatives dout，而當 x<=0，微分結果為 0。因此我先將 dx 令成 dout，再透過 index 將 x<=0 的部分改成 0。

2. 執行結果 ：

```
Testing ReLU.backward function:
dx error:  2.6317796097761553e-10
```

由上面結果可以看出，跟預期結果相比，誤差很小，代表說此 function 功能正確。

## 三、 “Sandwich” layers

在神經網路中，通常會出現一些常見的層模式，像是 linear layer 後面會接上 ReLU。為了定義這些層模式，我們可以定義一個 convenience layer，也就是說定義一個 layer 為 linear layer 和 ReLU 的組合。透過這種抽象化的概念，我們可以更方便的去管理 code 架構。接下來就是將剛剛實現的 Linear 和 ReLU 做結合，變成一個叫做 Linear_ReLU 的 convenience layer。

- Linear_ReLU.forward :
  1. 解題思路 ： 將 input 的 x、w、b 送進剛剛定義好的 Linear.forward。接著再將經過 Linear.forward 的輸出送進 ReLU.forward 即可。要注意的是，我們需要將過程中的參數記錄下來，以便在 backward path 中使用。
- Linear_ReLU.backward :
  1. 解題思路 ： 先接收從 Linear_ReLU.forward 的過程中存下來的參數，接著分別將這些參數送進先前定義好的 ReLU.backward 和 Linear.backward。由於是 back propagate，因此要先經過 ReLU.backward 再經過 Linear.backward。
  2. 執行結果 ：

```
Testing Linear_ReLU.forward and Linear_ReLU.backward:
dx error:  1.210759699545244e-09
dw error:  7.462948482161807e-10
db error:  8.915028842081707e-10
```

由上面結果可以看出，跟預期結果相比，誤差很小，代表說此 function 功能正確。

## 四、 Two-layer network

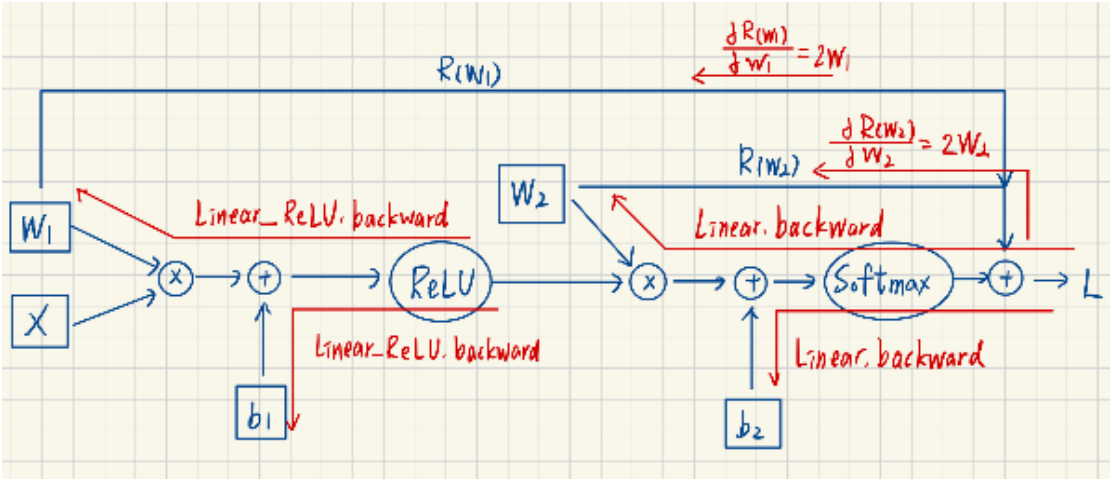
- __init__ :
  1. 解題思路 ： 這個為 TwoLayerNet 物件的初始化。由於是兩層的神經網路，因此會有四個參數要做初始化，分別為 W1、W2、b1、b2(他們都是 tensor)。W1 和 W2 我選擇利用 randn(normal distribution)的方式做隨機的初始化，b1、b2 則是初始化為 0。
  2. 額外討論 ：

     為甚麼可以直接將 b 初始化為 0 但是 W 卻不行，甚至還要特別使用 weight_scale 來控制初始化大小?

     b 可以初始化為 0 是因為它其實只是在做平移，並不需要做過多的變化或是隨機性。但是 W 就不一樣了。W 會直接的影像到神經網路的收斂性，在做 gradient descent 的時候，W 設置不好可能會造成所謂的梯度消失或是梯度爆炸，因此通常在做 W 的初始化都會採用一些隨機初始化的方法。

- TwoLayerNet.loss :

1. 解題思路 ：




根據上面的推導的結果以及先前定義好的 function 來完成。在計算 dw 的時候，由於先前並沒有加上 regularization 的影響，因此經過 Linear.backward 和 Linear_ReLU.backward 後還需要加上 $\frac{\partial R(w)}{\partial w} = 2w$ 的部分。

2. 執行結果 ：

```
Testing initialization ...
Testing test-time forward pass ...
Testing training loss (no regularization)
Running numeric gradient check with reg =  0.0
W1 relative error: 2.94e-07
W2 relative error: 1.65e-09
b1 relative error: 1.01e-06
b2 relative error: 4.63e-09
Running numeric gradient check with reg =  0.7
W1 relative error: 2.70e-08
W2 relative error: 9.86e-09
b1 relative error: 2.28e-06
b2 relative error: 2.90e-08
```

可以看出不管有沒有考慮 regularization，計算出來的結果誤差都很小，代表說功能正確。

## 五、 Solver

Solver 為一個將訓練分類器所需要的東西都封裝起來的類別。要使用 Solver，我們需要先創建一個 Solver 類別，並將 dataset、learning rate、batch size……等參數輸入進去。接著可以透過呼叫 train 方法來訓練模型。訓練完成後會將參數存放到 model.params 中，訓練過程的 loss、training

accuracy、validation accuracy 會分別存放到 solver.loss_history、solver.train_acc_history 和 solver.val_acc_history 中。

- __init__ :

  初始化權重的想法和 TwoLayerNet 相同。不一樣的是這次要針對多層的神經網路做初始化。對於每一層來說，W 的形狀為 (dim_of_last_layer, dim_of_current_layer)，而 b 的形狀為 (dim_of_current_layer, )。

- loss :

  loss 的部分分為 forward path 和 backward path。

  Forward path 的部分就是透過 for loop 不斷的將計算出來的東西送進 Linear_ReLU.forward，除了最後一層是送入 Linear.forward。與先前不同的是，這一次還要考慮使用 Dropout layer 的情況。Dropout 的部分會使用到 Dropout.forward，因此先說明 Dropout.forward 的實現。

  - Dropout.forward : 利用 torch.rand_like(x)生成與 x 相同形狀的 tensor，並對這個 tensor 使用 mask，讓大於 dropout 機率的位置設成 1，其餘設成 0。

  接著是 backward path 的部分。Backward path 的主要實現邏輯與 forward path 相同，都是透過 for loop 來進行多次的 propagation，差別在於 backward path 是不斷的經過 Linear_RuLU.backward。由於也要多考慮到 Dropout 的部分，因此這邊需要多實現 Dropout.backward。

  - Dropout.backward : Dropout.backward 的主要概念就是將在 forward path 過程中，被 dropout 的部分設成 0，其餘為 1。

- create_solver_instance :

  創建一個 Solver 物件並將要設定的參數輸入進去。這邊嘗試使用的參數如下 ：hidden_dim = 200、update_rule = sgd、learning_rate = 0.4、lr_decay = 0.96、batch_size = 512、num_epochs = 15。

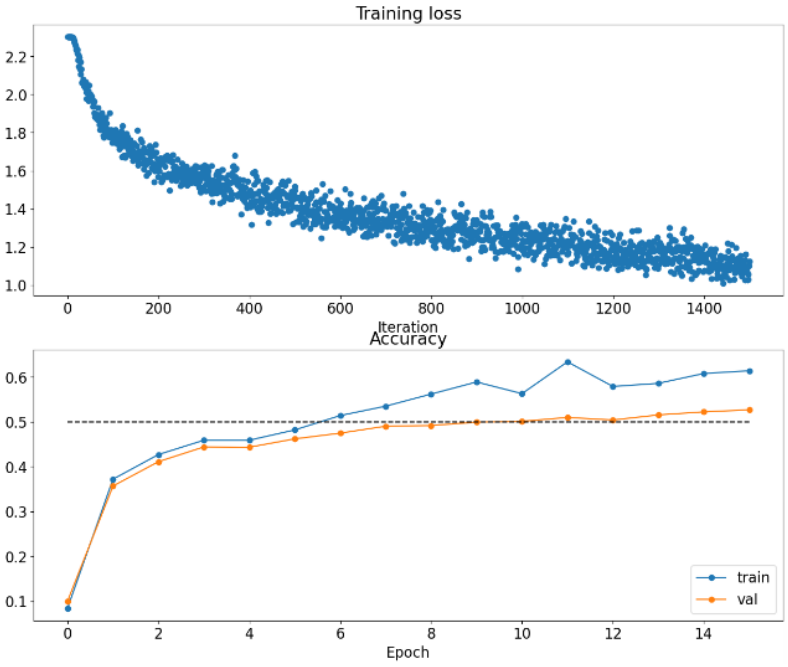
- 訓練結果 ：

  使用 create_solver_instance 所設置的參數後，訓練結果如下

```
(Time 0.03 sec; Iteration 1 / 1500) loss: 2.302587
(Epoch 0 / 15) train acc: 0.084000; val_acc: 0.099700
(Epoch 1 / 15) train acc: 0.372000; val_acc: 0.356700
(Time 2.38 sec; Iteration 101 / 1500) loss: 1.778478
(Epoch 2 / 15) train acc: 0.427000; val_acc: 0.411100
(Time 4.46 sec; Iteration 201 / 1500) loss: 1.623535
(Epoch 3 / 15) train acc: 0.459000; val_acc: 0.444000
(Time 6.55 sec; Iteration 301 / 1500) loss: 1.565434
(Epoch 4 / 15) train acc: 0.459000; val_acc: 0.443400
(Time 8.64 sec; Iteration 401 / 1500) loss: 1.474563
(Epoch 5 / 15) train acc: 0.482000; val_acc: 0.462000
(Time 10.73 sec; Iteration 501 / 1500) loss: 1.498445
(Epoch 6 / 15) train acc: 0.514000; val_acc: 0.474800
(Time 12.81 sec; Iteration 601 / 1500) loss: 1.430747
(Epoch 7 / 15) train acc: 0.535000; val_acc: 0.490300
(Time 14.90 sec; Iteration 701 / 1500) loss: 1.345892
(Epoch 8 / 15) train acc: 0.562000; val_acc: 0.491800
(Time 16.99 sec; Iteration 801 / 1500) loss: 1.236137
(Epoch 9 / 15) train acc: 0.589000; val_acc: 0.499400
(Time 19.07 sec; Iteration 901 / 1500) loss: 1.339699
(Epoch 10 / 15) train acc: 0.563000; val_acc: 0.501100
(Time 21.15 sec; Iteration 1001 / 1500) loss: 1.237303
(Epoch 11 / 15) train acc: 0.634000; val_acc: 0.509800
(Time 23.25 sec; Iteration 1101 / 1500) loss: 1.160639
(Epoch 12 / 15) train acc: 0.579000; val_acc: 0.504100
...
(Time 27.42 sec; Iteration 1301 / 1500) loss: 1.103148
(Epoch 14 / 15) train acc: 0.608000; val_acc: 0.522200
(Time 29.51 sec; Iteration 1401 / 1500) loss: 1.078400
(Epoch 15 / 15) train acc: 0.614000; val_acc: 0.526900
```

最佳的準確度為 52.69 %。

將結果視覺化如下

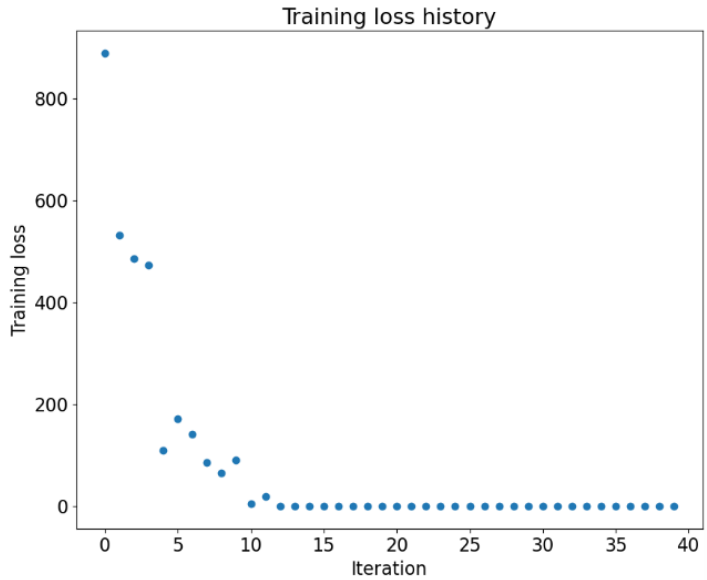


## 六、 Multilayer network

- get_three_layer_network_params :

  採用三層的神經網路對 50 筆資料訓練。這次可以調整的參數為 weight_scale 和 learning_rate。先將 weight_scale 設成 1，learning rate 設成 1e-2。訓練結果如下

```
(Time 0.01 sec; Iteration 1 / 40) loss: 888.938721
(Epoch 0 / 20) train acc: 0.260000; val_acc: 0.116000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.126200
(Epoch 2 / 20) train acc: 0.560000; val_acc: 0.141700
(Epoch 3 / 20) train acc: 0.600000; val_acc: 0.134500
(Epoch 4 / 20) train acc: 0.780000; val_acc: 0.141400
(Epoch 5 / 20) train acc: 0.900000; val_acc: 0.147500
(Time 0.11 sec; Iteration 11 / 40) loss: 4.525985
(Epoch 6 / 20) train acc: 0.980000; val_acc: 0.145300
(Epoch 7 / 20) train acc: 1.000000; val_acc: 0.145200
(Epoch 8 / 20) train acc: 1.000000; val_acc: 0.145200
(Epoch 9 / 20) train acc: 1.000000; val_acc: 0.145200
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.145200
(Time 0.19 sec; Iteration 21 / 40) loss: 0.000000
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.145200
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.145200
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.145200
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.145200
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.145200
(Time 0.27 sec; Iteration 31 / 40) loss: 0.000000
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.145200
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.145200
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.145200
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.145200
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.145200
```



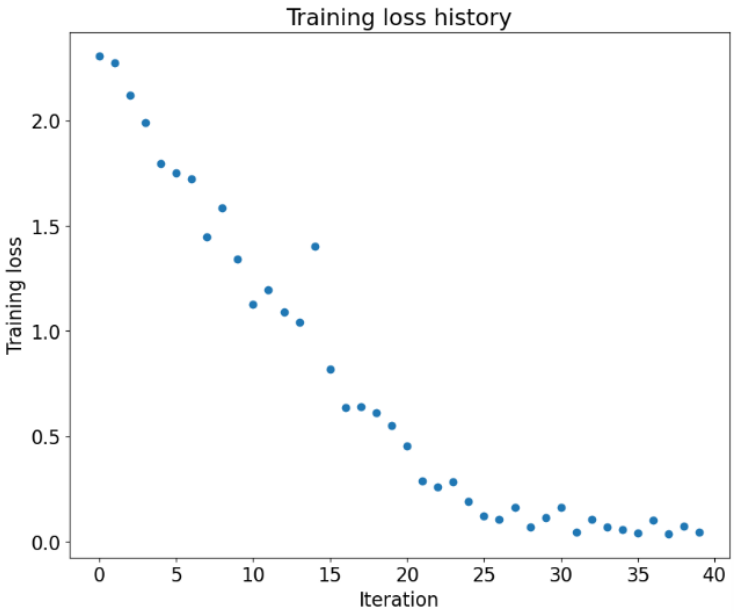

可以看出在 20epochs 內，training accuracy 就可以來到 100%，

overfitting 很嚴重。

- get_five_layer_network_params :

  採用五層的神經網路對 50 筆資料訓練。這次可以調整的參數為 weight_scale 和 learning_rate。先將 weight_scale 設成 1，learning rate 設成 0.1。訓練結果如下

```
(Time 0.00 sec; Iteration 1 / 40) loss: 2.307012
(Epoch 0 / 20) train acc: 0.220000; val_acc: 0.107200
(Epoch 1 / 20) train acc: 0.280000; val_acc: 0.114500
(Epoch 2 / 20) train acc: 0.440000; val_acc: 0.114000
(Epoch 3 / 20) train acc: 0.500000; val_acc: 0.118100
(Epoch 4 / 20) train acc: 0.600000; val_acc: 0.140400
(Epoch 5 / 20) train acc: 0.780000; val_acc: 0.153500
(Time 0.36 sec; Iteration 11 / 40) loss: 1.127662
(Epoch 6 / 20) train acc: 0.760000; val_acc: 0.150200
(Epoch 7 / 20) train acc: 0.600000; val_acc: 0.158300
(Epoch 8 / 20) train acc: 0.820000; val_acc: 0.170700
(Epoch 9 / 20) train acc: 0.880000; val_acc: 0.173400
(Epoch 10 / 20) train acc: 0.900000; val_acc: 0.153100
(Time 0.48 sec; Iteration 21 / 40) loss: 0.453191
(Epoch 11 / 20) train acc: 0.940000; val_acc: 0.166300
(Epoch 12 / 20) train acc: 0.980000; val_acc: 0.169300
(Epoch 13 / 20) train acc: 0.960000; val_acc: 0.177700
(Epoch 14 / 20) train acc: 0.960000; val_acc: 0.171200
(Epoch 15 / 20) train acc: 0.940000; val_acc: 0.180500
(Time 0.58 sec; Iteration 31 / 40) loss: 0.163125
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.178600
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.184200
(Epoch 18 / 20) train acc: 0.980000; val_acc: 0.187800
(Epoch 19 / 20) train acc: 0.980000; val_acc: 0.178200
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.187800
```




可以看出在 20epochs 內，training accuracy 就可以來到 100%，

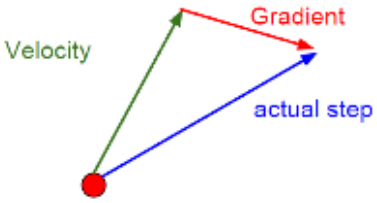
overfitting 很嚴重。

## 七、 Update rules

- sgd + momentum :

  根據 Nesterov Momentum 的公式，如下圖，來完成此函數。

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$




- 測試結果 :

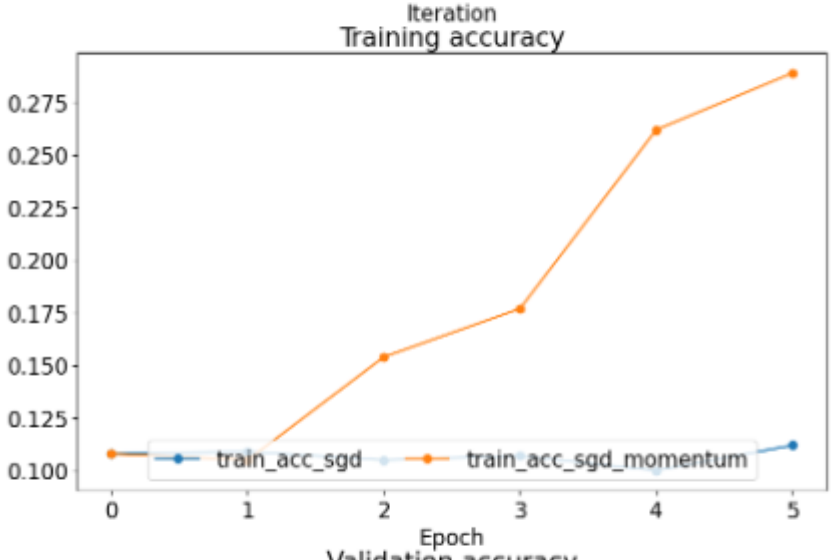
```
next_w error:  1.6802078709310813e-09
velocity error:  2.9254212825785614e-09
```

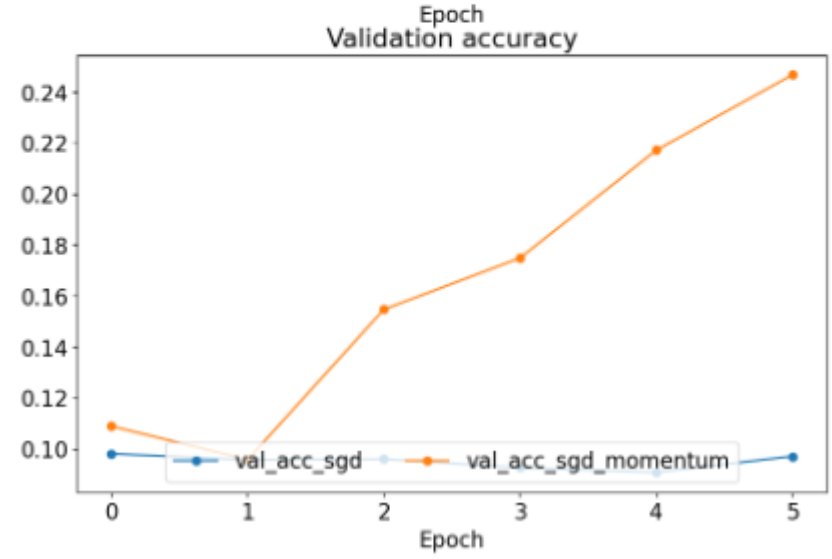
- 比較 sgd 與 sgd + momentum :

  分別透過 sgd 與 sgd + momentum 來訓練一個六層的神經網路。執行結果如下 :

```
running with  sgd
(Time 0.01 sec; Iteration 1 / 200) loss: 2.302109
(Epoch 0 / 5) train acc: 0.108000; val_acc: 0.098000
(Epoch 1 / 5) train acc: 0.109000; val_acc: 0.095600
(Epoch 2 / 5) train acc: 0.105000; val_acc: 0.095900
(Epoch 3 / 5) train acc: 0.107000; val_acc: 0.092500
(Epoch 4 / 5) train acc: 0.100000; val_acc: 0.090700
(Epoch 5 / 5) train acc: 0.112000; val_acc: 0.097000

running with  sgd_momentum
(Time 0.00 sec; Iteration 1 / 200) loss: 2.302904
(Epoch 0 / 5) train acc: 0.108000; val_acc: 0.108800
(Epoch 1 / 5) train acc: 0.105000; val_acc: 0.095700
(Epoch 2 / 5) train acc: 0.154000; val_acc: 0.154600
(Epoch 3 / 5) train acc: 0.177000; val_acc: 0.174800
(Epoch 4 / 5) train acc: 0.262000; val_acc: 0.217100
(Epoch 5 / 5) train acc: 0.289000; val_acc: 0.246700
```


由結果可以很明顯地看到，sgd＋momentum 的收斂速度快很多。

# 八、 額外嘗試

- 重新訓練三層、五層的神經網路 ：

  先前只是利用 50 張資料來查看模型是否可以 overfit，因此在確認能夠 overfit 後，我嘗試調整參數並用 40000 張訓練資料來訓練模型。

  1. 重新訓練三層神經網路 ：

     我主要針對 learning rate、weight scale 和 regularization strength 去做調整。我採用 for loop 來對多種參數組合作迭代。訓練完成後，最佳的結果如下

```
lr = 1, weight_scale = 0.01, reg = 0
(Time 0.01 sec; Iteration 1 / 7800) loss: 2.302249
(Epoch 0 / 100) train acc: 0.100000; val_acc: 0.101400
(Epoch 1 / 100) train acc: 0.335000; val_acc: 0.365100
(Epoch 2 / 100) train acc: 0.410000; val_acc: 0.399600
(Epoch 3 / 100) train acc: 0.480000; val_acc: 0.453200
(Epoch 4 / 100) train acc: 0.477000; val_acc: 0.435800
(Epoch 5 / 100) train acc: 0.523000; val_acc: 0.476900
(Epoch 6 / 100) train acc: 0.551000; val_acc: 0.501300
(Epoch 7 / 100) train acc: 0.577000; val_acc: 0.494600
(Epoch 8 / 100) train acc: 0.578000; val_acc: 0.501200
(Epoch 9 / 100) train acc: 0.577000; val_acc: 0.486200
(Epoch 10 / 100) train acc: 0.639000; val_acc: 0.523000
(Epoch 11 / 100) train acc: 0.607000; val_acc: 0.496900
(Epoch 12 / 100) train acc: 0.688000; val_acc: 0.535300
(Epoch 13 / 100) train acc: 0.703000; val_acc: 0.525900
(Epoch 14 / 100) train acc: 0.644000; val_acc: 0.483800
(Epoch 15 / 100) train acc: 0.747000; val_acc: 0.531200
(Epoch 16 / 100) train acc: 0.782000; val_acc: 0.539900
(Epoch 17 / 100) train acc: 0.773000; val_acc: 0.548400
(Epoch 18 / 100) train acc: 0.806000; val_acc: 0.542100
(Epoch 19 / 100) train acc: 0.808000; val_acc: 0.539000
(Epoch 20 / 100) train acc: 0.836000; val_acc: 0.543300
(Epoch 21 / 100) train acc: 0.846000; val_acc: 0.545600
(Epoch 22 / 100) train acc: 0.901000; val_acc: 0.547000
(Epoch 23 / 100) train acc: 0.908000; val_acc: 0.549100
(Epoch 24 / 100) train acc: 0.825000; val_acc: 0.509800
(Epoch 25 / 100) train acc: 0.923000; val_acc: 0.555000
(Epoch 26 / 100) train acc: 0.877000; val_acc: 0.538000
```

```
(Epoch 27 / 100) train acc: 0.827000; val_acc: 0.512800
(Epoch 28 / 100) train acc: 0.960000; val_acc: 0.553300
(Epoch 29 / 100) train acc: 0.976000; val_acc: 0.556900
(Epoch 30 / 100) train acc: 0.922000; val_acc: 0.545800
(Epoch 31 / 100) train acc: 0.973000; val_acc: 0.555400
(Epoch 32 / 100) train acc: 0.972000; val_acc: 0.558200
(Epoch 33 / 100) train acc: 0.984000; val_acc: 0.556200
(Epoch 34 / 100) train acc: 0.989000; val_acc: 0.563200
(Epoch 35 / 100) train acc: 0.989000; val_acc: 0.551000
(Epoch 36 / 100) train acc: 0.998000; val_acc: 0.561600
(Epoch 37 / 100) train acc: 0.999000; val_acc: 0.563900
(Epoch 38 / 100) train acc: 1.000000; val_acc: 0.568600
(Epoch 39 / 100) train acc: 0.999000; val_acc: 0.567500
(Epoch 40 / 100) train acc: 0.999000; val_acc: 0.565800
(Epoch 41 / 100) train acc: 1.000000; val_acc: 0.564000
(Epoch 42 / 100) train acc: 1.000000; val_acc: 0.565100
(Epoch 43 / 100) train acc: 1.000000; val_acc: 0.566500
(Epoch 44 / 100) train acc: 1.000000; val_acc: 0.564300
```

```
(Epoch 45 / 100) train acc: 1.000000; val_acc: 0.561600
(Epoch 46 / 100) train acc: 1.000000; val_acc: 0.564100
(Epoch 47 / 100) train acc: 1.000000; val_acc: 0.564400
(Epoch 48 / 100) train acc: 1.000000; val_acc: 0.564100
(Epoch 49 / 100) train acc: 1.000000; val_acc: 0.566100
(Epoch 50 / 100) train acc: 1.000000; val_acc: 0.566200
(Epoch 51 / 100) train acc: 1.000000; val_acc: 0.565400
(Epoch 52 / 100) train acc: 1.000000; val_acc: 0.565700
(Epoch 53 / 100) train acc: 1.000000; val_acc: 0.563700
(Epoch 54 / 100) train acc: 1.000000; val_acc: 0.563900
(Epoch 55 / 100) train acc: 1.000000; val_acc: 0.565500
(Epoch 56 / 100) train acc: 1.000000; val_acc: 0.566500
(Epoch 57 / 100) train acc: 1.000000; val_acc: 0.564800
(Epoch 58 / 100) train acc: 1.000000; val_acc: 0.564100
(Epoch 59 / 100) train acc: 1.000000; val_acc: 0.565100
(Epoch 60 / 100) train acc: 1.000000; val_acc: 0.565800
(Epoch 61 / 100) train acc: 1.000000; val_acc: 0.566100
(Epoch 62 / 100) train acc: 1.000000; val_acc: 0.565800
(Epoch 63 / 100) train acc: 1.000000; val_acc: 0.567300
(Epoch 64 / 100) train acc: 1.000000; val_acc: 0.565500
(Epoch 65 / 100) train acc: 1.000000; val_acc: 0.565800
(Epoch 66 / 100) train acc: 1.000000; val_acc: 0.566300
(Epoch 67 / 100) train acc: 1.000000; val_acc: 0.565900
(Epoch 68 / 100) train acc: 1.000000; val_acc: 0.564800
(Epoch 69 / 100) train acc: 1.000000; val_acc: 0.564000
(Epoch 70 / 100) train acc: 1.000000; val_acc: 0.565400
(Epoch 71 / 100) train acc: 1.000000; val_acc: 0.565800
```

```
(Epoch 72 / 100) train acc: 1.000000; val_acc: 0.564900
(Epoch 73 / 100) train acc: 1.000000; val_acc: 0.564800
(Epoch 74 / 100) train acc: 1.000000; val_acc: 0.564500
(Epoch 75 / 100) train acc: 1.000000; val_acc: 0.566000
(Epoch 76 / 100) train acc: 1.000000; val_acc: 0.565100
(Epoch 77 / 100) train acc: 1.000000; val_acc: 0.565300
(Epoch 78 / 100) train acc: 1.000000; val_acc: 0.564600
(Epoch 79 / 100) train acc: 1.000000; val_acc: 0.564900
(Epoch 80 / 100) train acc: 1.000000; val_acc: 0.563300
(Epoch 81 / 100) train acc: 1.000000; val_acc: 0.564700
(Epoch 82 / 100) train acc: 1.000000; val_acc: 0.564600
(Epoch 83 / 100) train acc: 1.000000; val_acc: 0.564400
(Epoch 84 / 100) train acc: 1.000000; val_acc: 0.565300
(Epoch 85 / 100) train acc: 1.000000; val_acc: 0.565400
(Epoch 86 / 100) train acc: 1.000000; val_acc: 0.564500
(Epoch 87 / 100) train acc: 1.000000; val_acc: 0.564500
(Epoch 88 / 100) train acc: 1.000000; val_acc: 0.564000
(Epoch 89 / 100) train acc: 1.000000; val_acc: 0.563400
(Epoch 90 / 100) train acc: 1.000000; val_acc: 0.564000
(Epoch 91 / 100) train acc: 1.000000; val_acc: 0.564700
(Epoch 92 / 100) train acc: 1.000000; val_acc: 0.564200
(Epoch 93 / 100) train acc: 1.000000; val_acc: 0.564300
(Epoch 94 / 100) train acc: 1.000000; val_acc: 0.563700
(Epoch 95 / 100) train acc: 1.000000; val_acc: 0.564600
(Epoch 96 / 100) train acc: 1.000000; val_acc: 0.564400
(Epoch 97 / 100) train acc: 1.000000; val_acc: 0.564300
(Epoch 98 / 100) train acc: 1.000000; val_acc: 0.563400
(Epoch 99 / 100) train acc: 1.000000; val_acc: 0.564100
(Epoch 100 / 100) train acc: 1.000000; val_acc: 0.564000
```

神經網路架構為第一層 hidden layer size = 512，第二層 hidden layer size = 1024。

由上面的結果可以看出，對於三層的神經網路來說，在 learning rate = 1、weight scale = 0.01 和 regularization strength = 0 時，會有最好的 validation accuracy 為 56.4 %。

2. 重新訓練五層神經網路 ：

這次我主要針對 learning rate、weight scale、regularization strength 和 dropout 做調整。最好的訓練結果如下

```
lr = 1.1, weight_scale = 0.01, reg = 0, dropout=0.1
(Time 0.00 sec; Iteration 1 / 7800) loss: 2.302588
(Epoch 0 / 100) train acc: 0.099000; val_acc: 0.102200
(Epoch 1 / 100) train acc: 0.118000; val_acc: 0.097700
(Epoch 2 / 100) train acc: 0.094000; val_acc: 0.101400
(Epoch 3 / 100) train acc: 0.184000; val_acc: 0.162900
(Epoch 4 / 100) train acc: 0.189000; val_acc: 0.182100
(Epoch 5 / 100) train acc: 0.269000; val_acc: 0.251100
(Epoch 6 / 100) train acc: 0.236000; val_acc: 0.245500
(Epoch 7 / 100) train acc: 0.342000; val_acc: 0.342200
(Epoch 8 / 100) train acc: 0.400000; val_acc: 0.396100
(Epoch 9 / 100) train acc: 0.424000; val_acc: 0.415700
(Epoch 10 / 100) train acc: 0.443000; val_acc: 0.420300
(Epoch 11 / 100) train acc: 0.470000; val_acc: 0.451900
(Epoch 12 / 100) train acc: 0.503000; val_acc: 0.465000
(Epoch 13 / 100) train acc: 0.523000; val_acc: 0.467900
(Epoch 14 / 100) train acc: 0.542000; val_acc: 0.480200
(Epoch 15 / 100) train acc: 0.574000; val_acc: 0.493800
(Epoch 16 / 100) train acc: 0.503000; val_acc: 0.467600
(Epoch 17 / 100) train acc: 0.538000; val_acc: 0.473200
(Epoch 18 / 100) train acc: 0.582000; val_acc: 0.487200
(Epoch 19 / 100) train acc: 0.596000; val_acc: 0.496300
(Epoch 20 / 100) train acc: 0.597000; val_acc: 0.498400
(Epoch 21 / 100) train acc: 0.646000; val_acc: 0.514900
(Epoch 22 / 100) train acc: 0.635000; val_acc: 0.525900
(Epoch 23 / 100) train acc: 0.612000; val_acc: 0.494000
(Epoch 24 / 100) train acc: 0.626000; val_acc: 0.524300
(Epoch 25 / 100) train acc: 0.672000; val_acc: 0.515400
(Epoch 26 / 100) train acc: 0.652000; val_acc: 0.532800
```

```
(Epoch 27 / 100) train acc: 0.630000; val_acc: 0.507200
(Epoch 28 / 100) train acc: 0.669000; val_acc: 0.519700
(Epoch 29 / 100) train acc: 0.689000; val_acc: 0.516200
(Epoch 30 / 100) train acc: 0.721000; val_acc: 0.529600
(Epoch 31 / 100) train acc: 0.707000; val_acc: 0.528000
(Epoch 32 / 100) train acc: 0.689000; val_acc: 0.524200
(Epoch 33 / 100) train acc: 0.736000; val_acc: 0.527300
(Epoch 34 / 100) train acc: 0.727000; val_acc: 0.525400
(Epoch 35 / 100) train acc: 0.677000; val_acc: 0.497100
(Epoch 36 / 100) train acc: 0.725000; val_acc: 0.518900
(Epoch 37 / 100) train acc: 0.734000; val_acc: 0.530100
(Epoch 38 / 100) train acc: 0.735000; val_acc: 0.521900
(Epoch 39 / 100) train acc: 0.731000; val_acc: 0.527500
(Epoch 40 / 100) train acc: 0.767000; val_acc: 0.524900
(Epoch 41 / 100) train acc: 0.784000; val_acc: 0.530700
(Epoch 42 / 100) train acc: 0.792000; val_acc: 0.523000
(Epoch 43 / 100) train acc: 0.779000; val_acc: 0.527200
(Epoch 44 / 100) train acc: 0.755000; val_acc: 0.514900
(Epoch 45 / 100) train acc: 0.808000; val_acc: 0.532300
(Epoch 46 / 100) train acc: 0.778000; val_acc: 0.520100
(Epoch 47 / 100) train acc: 0.804000; val_acc: 0.533700
(Epoch 48 / 100) train acc: 0.851000; val_acc: 0.534800
(Epoch 49 / 100) train acc: 0.793000; val_acc: 0.525700
(Epoch 50 / 100) train acc: 0.822000; val_acc: 0.533000
(Epoch 51 / 100) train acc: 0.833000; val_acc: 0.529100
(Epoch 52 / 100) train acc: 0.858000; val_acc: 0.535100
(Epoch 53 / 100) train acc: 0.824000; val_acc: 0.528700
(Epoch 54 / 100) train acc: 0.844000; val_acc: 0.535500
(Epoch 55 / 100) train acc: 0.819000; val_acc: 0.515500
```

```
(Epoch 56 / 100) train acc: 0.862000; val_acc: 0.532600
(Epoch 57 / 100) train acc: 0.867000; val_acc: 0.533000
(Epoch 58 / 100) train acc: 0.872000; val_acc: 0.520600
(Epoch 59 / 100) train acc: 0.873000; val_acc: 0.537000
(Epoch 60 / 100) train acc: 0.887000; val_acc: 0.530800
(Epoch 61 / 100) train acc: 0.873000; val_acc: 0.533500
(Epoch 62 / 100) train acc: 0.902000; val_acc: 0.532500
(Epoch 63 / 100) train acc: 0.883000; val_acc: 0.526000
(Epoch 64 / 100) train acc: 0.887000; val_acc: 0.533900
(Epoch 65 / 100) train acc: 0.888000; val_acc: 0.537100
(Epoch 66 / 100) train acc: 0.891000; val_acc: 0.532500
(Epoch 67 / 100) train acc: 0.874000; val_acc: 0.529400
(Epoch 68 / 100) train acc: 0.893000; val_acc: 0.530300
(Epoch 69 / 100) train acc: 0.894000; val_acc: 0.530300
(Epoch 70 / 100) train acc: 0.917000; val_acc: 0.532700
(Epoch 71 / 100) train acc: 0.912000; val_acc: 0.528400
(Epoch 72 / 100) train acc: 0.901000; val_acc: 0.531200
(Epoch 73 / 100) train acc: 0.909000; val_acc: 0.536600
(Epoch 74 / 100) train acc: 0.914000; val_acc: 0.531900
(Epoch 75 / 100) train acc: 0.909000; val_acc: 0.530900
(Epoch 76 / 100) train acc: 0.908000; val_acc: 0.531500
(Epoch 77 / 100) train acc: 0.913000; val_acc: 0.530800
(Epoch 78 / 100) train acc: 0.927000; val_acc: 0.536000
(Epoch 79 / 100) train acc: 0.922000; val_acc: 0.534900
(Epoch 80 / 100) train acc: 0.932000; val_acc: 0.527600
(Epoch 81 / 100) train acc: 0.923000; val_acc: 0.536000
(Epoch 82 / 100) train acc: 0.935000; val_acc: 0.528600
(Epoch 83 / 100) train acc: 0.926000; val_acc: 0.528300
```

```
(Epoch 84 / 100) train acc: 0.931000; val_acc: 0.534600
(Epoch 85 / 100) train acc: 0.929000; val_acc: 0.532200
(Epoch 86 / 100) train acc: 0.932000; val_acc: 0.530900
(Epoch 87 / 100) train acc: 0.934000; val_acc: 0.533200
(Epoch 88 / 100) train acc: 0.952000; val_acc: 0.537400
(Epoch 89 / 100) train acc: 0.932000; val_acc: 0.527600
(Epoch 90 / 100) train acc: 0.940000; val_acc: 0.527000
(Epoch 91 / 100) train acc: 0.942000; val_acc: 0.524300
(Epoch 92 / 100) train acc: 0.943000; val_acc: 0.521000
(Epoch 93 / 100) train acc: 0.949000; val_acc: 0.524200
(Epoch 94 / 100) train acc: 0.951000; val_acc: 0.521700
(Epoch 95 / 100) train acc: 0.950000; val_acc: 0.525100
(Epoch 96 / 100) train acc: 0.957000; val_acc: 0.533500
(Epoch 97 / 100) train acc: 0.959000; val_acc: 0.538200
(Epoch 98 / 100) train acc: 0.952000; val_acc: 0.528600
(Epoch 99 / 100) train acc: 0.963000; val_acc: 0.535100
(Epoch 100 / 100) train acc: 0.947000; val_acc: 0.532300
```

網路架構為第一層 hidden layer size = 128，第二層 hidden layer size = 256、第三層 hidden layer size = 512、第四層 hidden layer size = 1024。

由上面的圖片可以看出，最好的結果發生在 learning rate = 1.1、weight scale = 0.01、regularization strength = 0 和 dropout = 0.1。最好的準確度為 53%。

由重新練模型結果可以發現，對於三層網路與五層網路來說，最好的準確度大約落在 53~56%，與之前作業的兩層網路相比，其實相差不遠，代表說對於 CIFAR-10 來說，只增加 1、2 層並不會造成甚麼進步。比較讓人意外的是，拿三層網路與五層網路相

比，五層網路表現比較不好，這不太合理，因此接下來討論一下原因：

i. Overfitting：加大模型後確實會更加容易造成 overfitting，但是我認為在這次作業中，造成準確度下降的原因不完全是因為 overfitting，因為訓練的時候，我有加上 regularization 和 dropout，在這兩個參數比較大的時候，模型並不會有表現上提升。

ii. 資料分布：資料分布造成訓練上的困難確實會造成模型表現不如預期。因此我嘗試檢查資料預處理過後的樣子，結果如下

```
tensor([[-0.2600, -0.3228, -0.2953,  ...,  0.1025, -0.1171, -0.1642],
        [ 0.1125,  0.0027, -0.0796,  ...,  0.0986,  0.1103,  0.1182],
        [ 0.5086,  0.5008,  0.5008,  ..., -0.1210, -0.1210, -0.1171],
        ...,
        [-0.3816, -0.3738, -0.3620,  ..., -0.0544, -0.0583, -0.0701],
        [ 0.0341,  0.0223,  0.0106,  ...,  0.0868,  0.0907,  0.0946],
        [-0.0012, -0.0600, -0.0914,  ..., -0.1250, -0.1171, -0.1093]],
       device='cuda:0', dtype=torch.float64)
```

由上面顯示 X_train 的結果可以看到，資料已經有經過 normalization 的預處理，因此資料分布不均應該也不會是問題。

iii. 訓練前沒有 shuffle：想像一下，訓練模型時，假如一個 epoch 有 30 個資料，前十個資料都是 0，接著十個資料都是 1，最後十個資料都是 2。這會造成模型在一開始被訓練成只會回答 0 的懶惰模型，接著雖然會去學習 1 特徵，但是沒回答幾題後發現答案都是 1，又變回懶惰模型……。這就會造成模型學習不起來。為了確認是否有這樣的問題，我將前面以個 label 顯示出來，結果如下

```
tensor([6, 9, 9, 4, 1, 1, 2, 7, 8, 3, 4, 7, 7, 2, 9, 9, 9, 3, 2, 6, 4, 3, 6, 6,
        2, 6, 3, 5, 4, 0, 0, 9, 1, 3, 4, 0, 3, 7, 3, 3, 5, 2, 2, 7, 1, 1, 1, 2,
        2, 0, 9, 5, 7, 9, 2, 2, 5, 2, 4, 3, 1, 1, 8, 2, 1, 1, 4, 9, 7, 8, 5, 9,
        6, 7, 3, 1, 9, 0, 3, 1, 3, 5, 4, 5, 7, 7, 4, 7, 9, 4, 2, 3, 8, 0, 1, 6,
        1, 1, 4, 1], device='cuda:0')
```

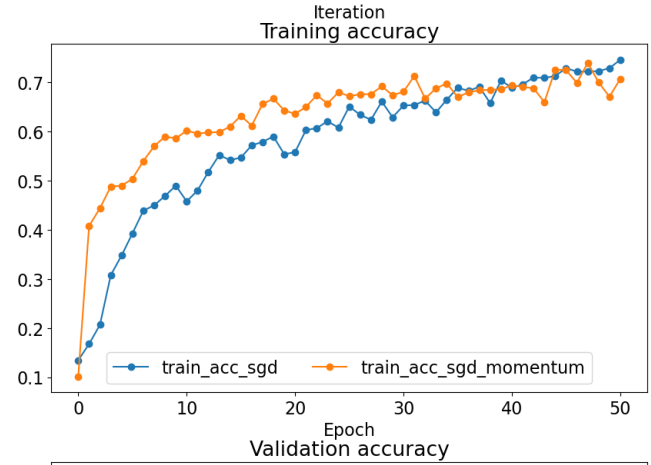
由結果可以看出，資料是有經過 shuffle 的。

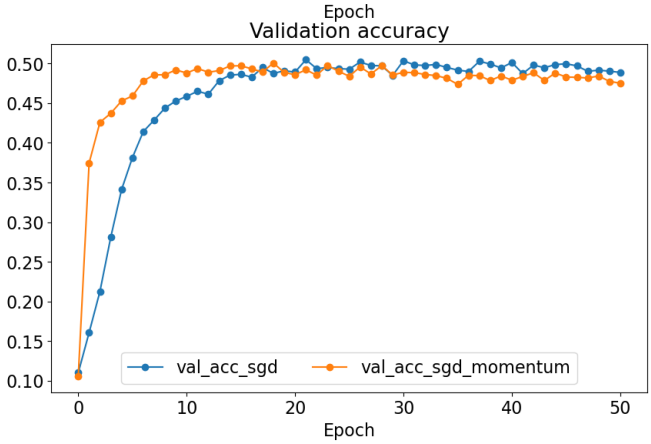
經過一系列的檢查，我沒有發現到問題所在，因此我認為會造成這樣的現象是因為單純增加一兩層的 hidden layer 對於準確度提升並沒有太大幫助，當訓練超參數稍微有一些偏差，

就很可能造成準確度不如預期。

- 嘗試使用 SGD 與 SGD + Momentum 來訓練六層神經網路

  嘗試修改 learning rate、weight scale、regularization strength 之後訓練模型，發現到模型的準確度基本上都會卡在 50%上下。訓練節果如下


由視覺化後結果可以很明顯地看到，準確度會卡在 50 %左右。結合前面重新訓練三層、五層神經網路，以及之前作業的兩層網路的結果會發現到，不論是哪一種架構，準確度大約都在 50%左右，代表說單純是增加幾層網路其實對於準確度提升很有限。如果需進一步提升準確度，可能需要改變神經網路架構，像是採用 CNN 或是 ResNet 等架構。

- Optimizer :

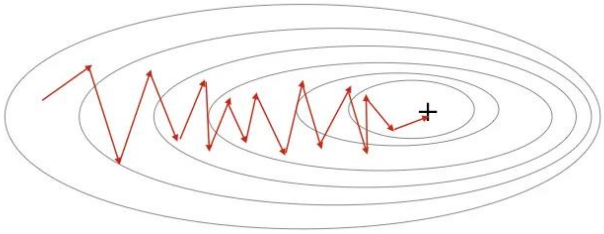
  由於這次作業有做到 sgd + momentum 的實現，因此我希望可以多加了解一下 optimizer 的原理與種類。

  1. Gradient Descent : gradient descent 是一種基於凸函數的優化演算法。他透過沿著負 gradient 方向來降低 loss。

$$W_{new} = W_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$

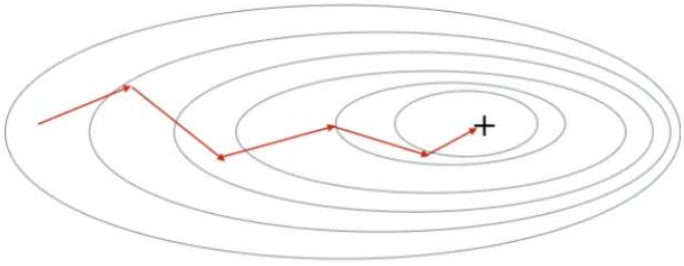
     - 優點 : 很好理解，實做起來簡單。
     - 缺點 : 由於他一次要計算所有訓練資料的梯度並一次更新所有參數，因此計算速度很慢，而且需要花費很多記憶體空間。

  2. Stochastic Gradient Descent (SGD) : 主要概念與 gradient descent 相同，差別在於他一次只更新一個參數。



     - 優點 : 他可以對大型的資料做計算，且需要比較少的記憶體空間。
     - 缺點 : 由於 SGD 更新參數時只有使用一個樣本，因此可能有 noise，使模型更新不穩定。此外，計算成本也比較高。

  3. Mini-Batch Gradient Descent : 將訓練資料分成 batches 並對這些 batches 做參數更新。

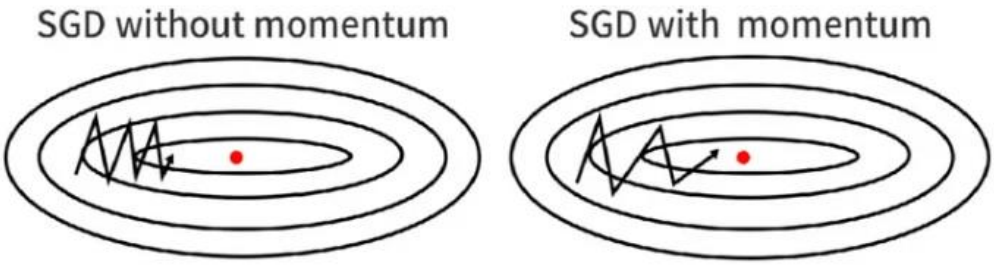


     - 優點 : 收斂過程會更佳穩定，計算複雜度也比較低。
     - 缺點 : 容易受到 learning rate 的影響。

  4. SGD with Momentum : 在 SGD 的基礎上加上 momentum。

Momentum 使參數在更新一定程度的保留前一次更新時的方向，並利用當前的梯度對該方向做調整。

$$\nu_{new} = \eta * \nu_{old} - \alpha * \frac{\partial(Loss)}{\partial(W_{old})}$$




- 優點 ：momentum 可以幫助減少 noise，而且 exponential weighted average 可以讓曲線更平滑。
- 缺點 ： 需要額外的超參數。

5. AdaGrad (Adaptive Gradient Descent ) ： 先前討論的所有方法中，learning rate 都是固定的。AdaGrad 使我們可以對不同的神經元、迭代次數使用不一樣的 learning rate。

$$W_{new} = W_{old} + \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * \frac{\partial(Loss)}{\partial(W_{old})}$$

- 優點 ：learning rate 可以根據迭代次數做調整。對於稀疏的資料也可以有不錯的表現。
- 缺點 ： 對於深層的神經網路來說，learning rate 會變得很小，導致一些神經元權重更新很慢，也就是所謂的 dead neuron problem。

6. RMS-Prop (Root Mean Square Propagaion) ： 為一種特殊版本的 AdaGrad。Learning rate 為梯度的 exponential average。通常會結合 momentum 一起使用。

$$cache_{new} = \gamma * cache_{old} + (1 - \gamma) * (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

- 優點 ：learning rate 可以自動更新，而且對於每一個參數都可

以有不一樣的 learning rate。

- 缺點：訓練比較慢。

7. Adam (Adaptive Moment Estimation)：為目前最流行的 optimizer。他是一種可以為每一個參數都計算 adaptive learning rate 的方法。他結合了 momentum 和 RMS-Prop 的使用。

$$w_t = w_{t-1} - \frac{\eta}{\sqrt{S_{dw_t} - \varepsilon}} * V_{dw_t}$$

$$b_t = b_{t-1} - \frac{\eta}{\sqrt{S_{db_t} - \varepsilon}} * V_{db_t}$$

- 優點：計算效率高、需要比較少的記憶體空間。

8. 如何選擇 optimizer：

宗傑以上幾種 optimizer 的特性，可以歸納出幾個選擇標準。

- 對於稀疏的資料，採用 adaptive learning rate 的 optimizer。
- 在很多情況下，Adam、RMSprop 有差不多的表現。
- Adam 通常會比 RMSprop 表現更好一點，因為 Adam 可以透過 momentum 來保持更新的方向的一致性。
- 對於較大型的模型，建議使用 Adam，因為 Adam 具有更好的收斂性能。

## 九、 心得總結

這次作業嘗試使用多層的神經網路來對 CIFAR-10 資料集做訓練。透過 convenience layer 和封裝的概念，將需要的東西裝成一包來幫助我們更好的管理 code 以及網路架構。此外也實現了 SGD + momentum optimizer 來幫助我們訓練模型。

接下來嘗試從新訓練作業中的幾個神經網路。經過一番嘗試之後發現到僅僅增加幾層神經網路對於準確度提升並沒有太大幫助，如果希望有更大的提升，可能要嘗試更換網路架構，像是採用 ResNet、CNN 等等之類的架構。

最後我自己到網路上搜尋與整理有關 optimizer 的資料，並對多種 optimizer 的原理與優缺做比較。基本上 RMSprop 與 Adam 都有很不錯的表現，而

Adam 通常會表現得比 RMSprop 還要好，因此很多情況下 Adam 會是一個 default 的選擇。

# 十、 Reference


[1] Musstafa “Optimizers in Deep Learning” https://musstafa0804.medium.com/optimizers-in-deep-learning-7bf81fed78a0

[2] LS_learner “訓練及準確度很高，驗證集準確率低的問題” https://blog.csdn.net/qq_39777550/article/details/108965486

[3] ML Glossary “Optimizers” https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html

[4] OpenAI. (2023). ChatGPT (Mar 14 version) [Large language model]. https://chat.openai.com/