深度學習

HW8

學號 : B103012002

姓名 : 林凡皓

接續上一次作業，將後半部分 (Update rules) 完成。本次作業內容從七、update rules 開始。

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：

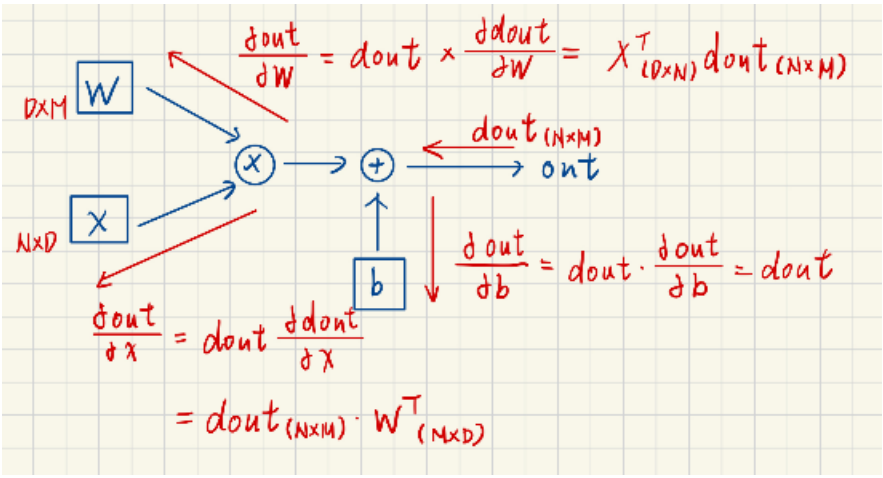
```
Testing Linear.forward function:
difference:  3.683042917976506e-08
```

  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：

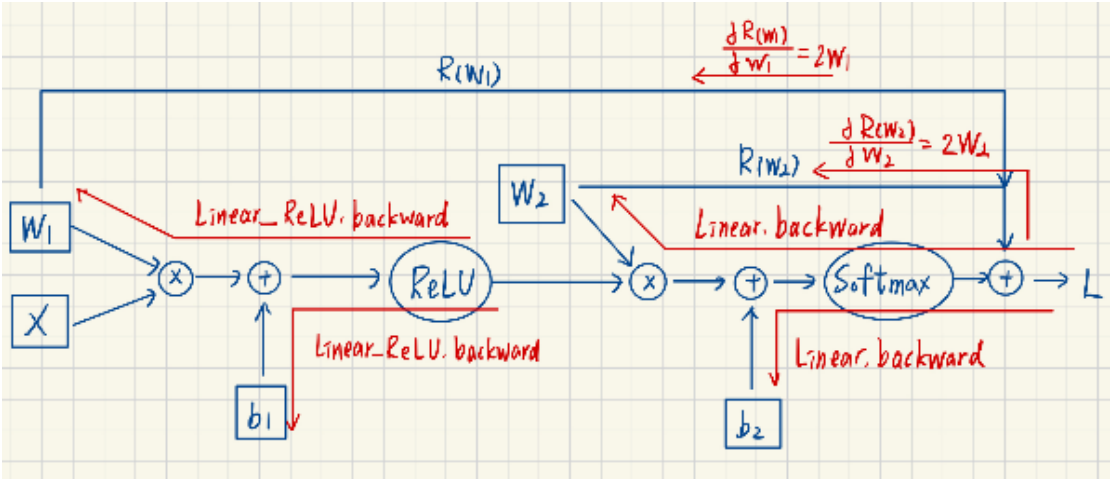
     為甚麼可以直接將 b 初始化為 0 但是 W 卻不行，甚至還要特別使用 weight_scale 來控制初始化大小?

     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。

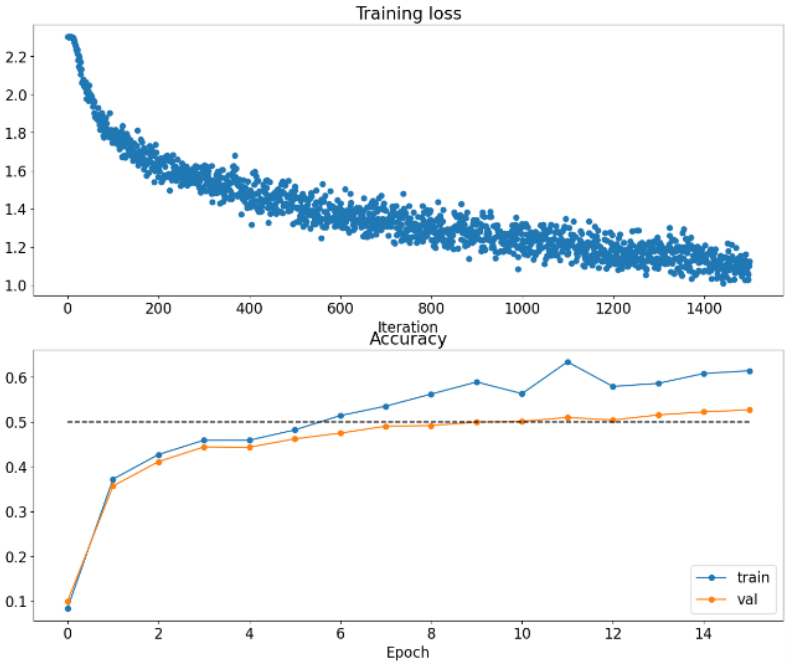
- 訓練結果：

  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

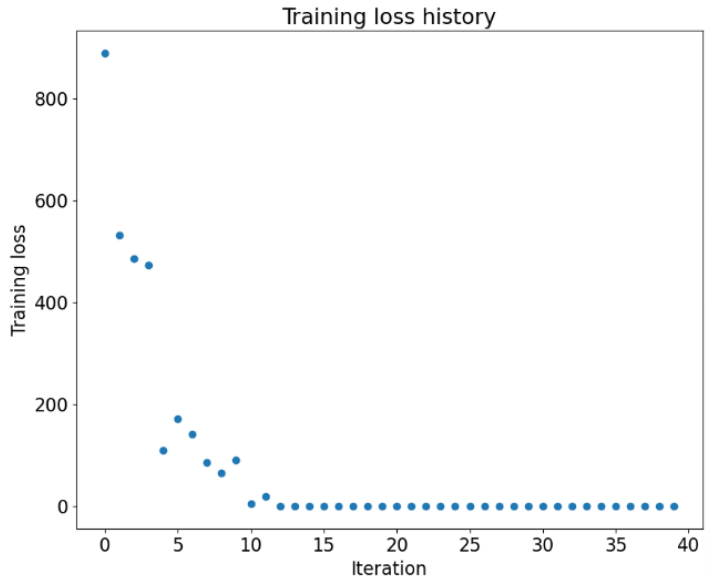


六、 Multilayer network

- 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
```



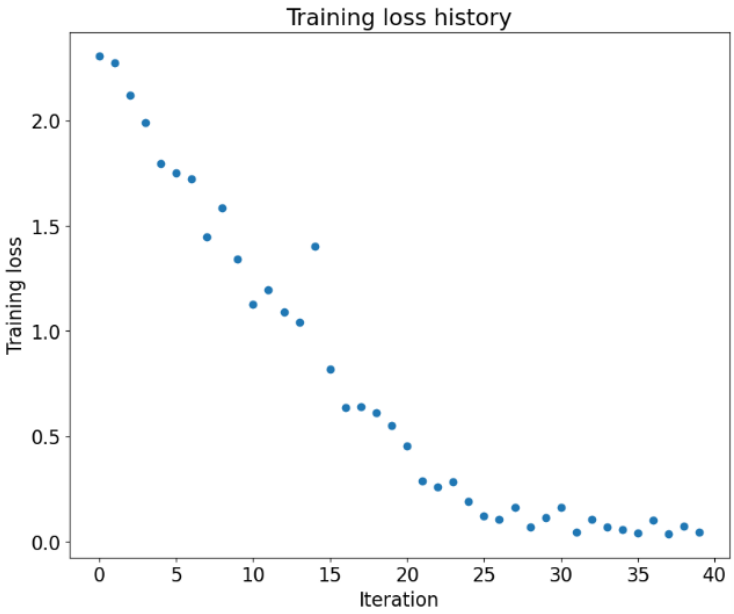
可以看出在 20epochs 內，training accuracy 就可以來到 100%，

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%，

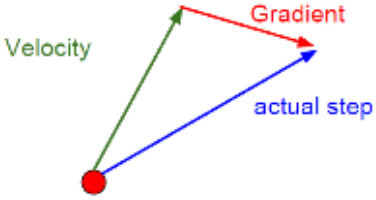
overfitting 很嚴重。

七、 Update rules

- sgd + momentum :

1. 解題思路 : 根據 Nesterov Momentum 的公式，如下圖，來完成此函數。

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



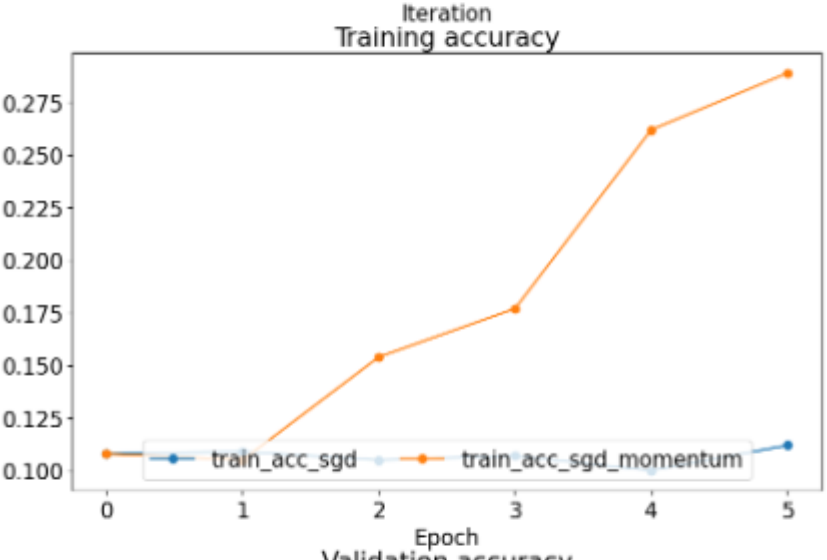
2. 測試結果 :

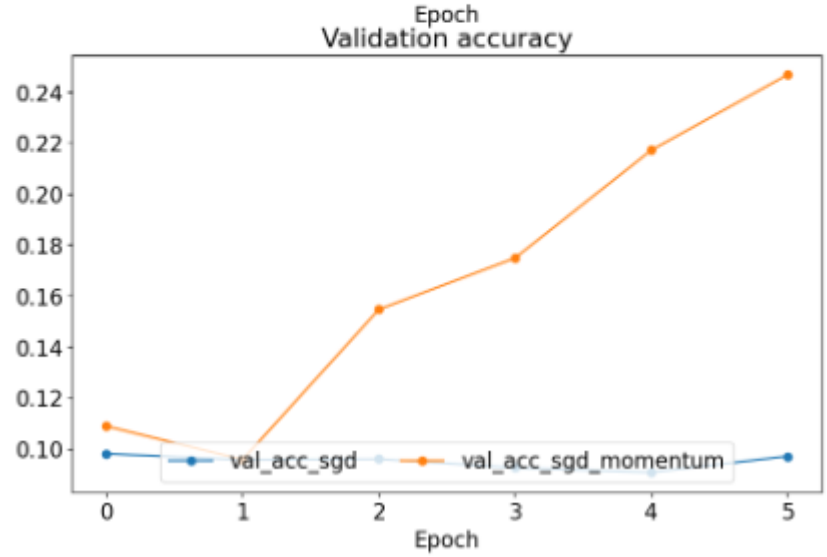
```
next_w error:  1.6802078709310813e-09
velocity error:  2.9254212825785614e-09
```

3. 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。

- RMSProp :

  1. 解題思路：根據 RMSProp 的公式來完成此題。有關 RMSProp 更加詳細說明，請參考八、額外嘗試中的 optimizer 部分。

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

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

2. 執行結果：

```
next_w error:  4.0647978808298 26e-09
cache error:  1.8620321382570356e-09
```

- Adam：

1. 解題思路：根據 Adam 的公式來完成本題。有關 Adam 更加詳細說明，請參考八、額外嘗試中的 optimizer 部分。

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial L_t}{\partial W_t}$$

$$v_t = \beta_1 v_{t-1} + (1-\beta_2)(\frac{\partial L_t}{\partial W_t})^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

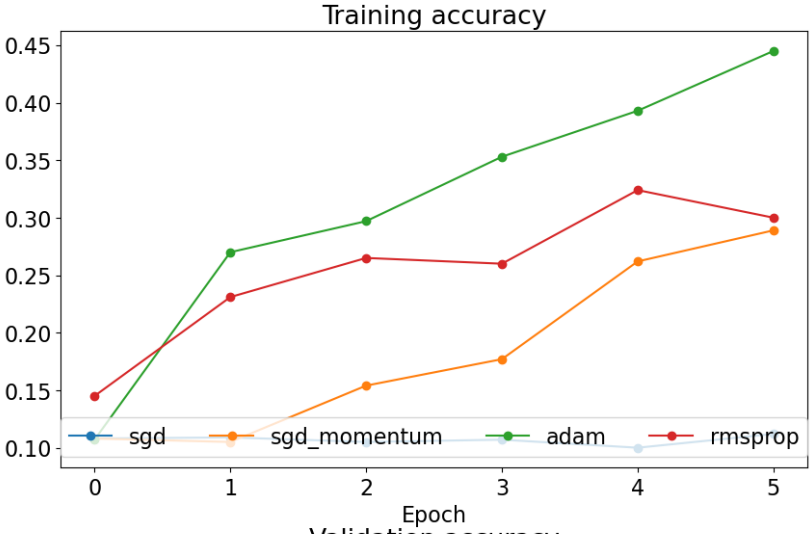
$$W \leftarrow W - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

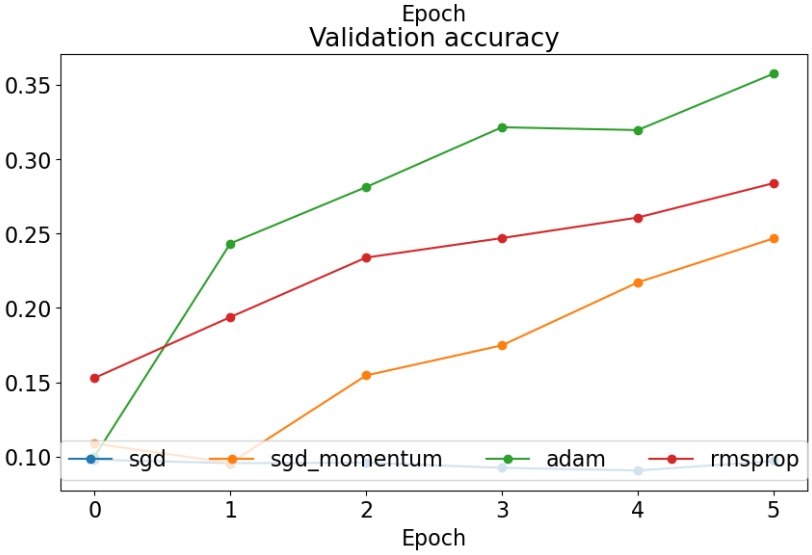
2. 執行結果：

```
next_w error:  3.756728297598868e-09
v error:  3.4048987160545265e-09
m error:  2.7863777298536 51e-09
```

- Optimizer 的比較：

比較 sgd、sgd_momentum、adam 與 rmsprop 對於訓練過程的影響，結果如下圖





由結果可以看出，在引入 momentum 的概念後，收斂速度有大幅提升。在加入隨梯度大小調整 learning rate 的概念後(rmsprop、adam)，收斂速度會再次提升，此外，也可以注意到除了速度上的提升，loss 與 accuracy 都可以優化的更好，超越沒有調整 learning rate 時的表現。

八、 Dropout

Dropout 會在 forward path 中隨機將部分神經元的輸出結果設成 0，以防止 overfitting 的發生。

- Dropout : forward

1. 解題思路：

   dropout 在訓練階段與測試階段的表現有所不同，因此需要根據這兩種模式寫入不一樣的方法。

   在訓練階段，可以透過遮罩的方式。生成一個隨機 tensor (tensor 中的數值需要在[0, 1)中)，將 tensor 中大於 p 的值設成 1，其餘為 0。在讓輸入經過這個 mask 即為輸出。

   在測試階段，由於不需要執行 dropout 的動作，因此只需要將輸入回傳即可。

2. 執行結果：

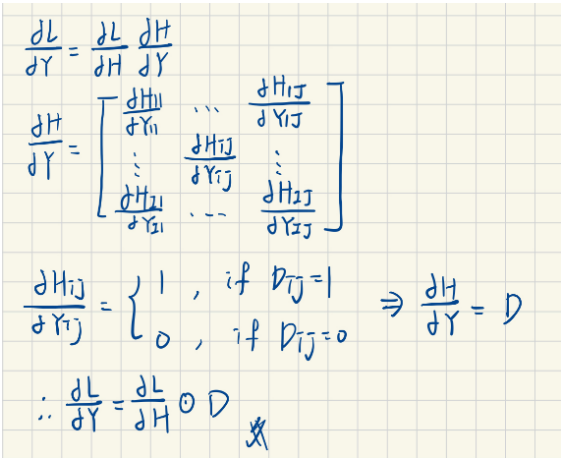
```
Running tests with p =  0.25
Mean of input:  10.002638910253042
Mean of train-time output:  10.00733147848024
Mean of test-time output:  10.002638910253042
Fraction of train-time output set to zero:  0.24967199563980103
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  10.002638910253042
Mean of train-time output:  10.006291379139071
Mean of test-time output:  10.002638910253042
Fraction of train-time output set to zero:  0.3998199999332428
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  10.002638910253042
Mean of train-time output:  10.006688044617057
Mean of test-time output:  10.002638910253042
Fraction of train-time output set to zero:  0.6999120116233826
Fraction of test-time output set to zero:  0.0
```

由上面結果可以看出，fraction of train-time output set to zero 的數值會根據 dropout 機率 p 的設置而改變，且兩者的數值會幾乎相等，代表說這個函數的公正確。

- Dropout : backward

1. 解題思路：

   Backward path 和 forward path 一樣會在訓練與測試時有不一樣的表現。在測試時和 forward path 相同，之需要將 dx 設成 dout 即可。訓練的部分要根據下圖推導的結果來完成。



2. 執行結果：

```
dx relative error:  3.882378970269327e-09
```

九、 Full-connected nets with dropout

- 在 FullyConnectedNet 中加入 dropout 功能

  先前執行 FullyConnectedNet 都沒有使用到 dropout，現在試著使用 dropout 來確認 dropout 可以正常運作。執行結果如下

```
Running check with dropout =  0
Initial loss:  2.3053575717037686
W1 relative error: 6.06e-08
W2 relative error: 1.02e-07
W3 relative error: 5.89e-08
b1 relative error: 1.28e-07
b2 relative error: 2.05e-08
b3 relative error: 3.41e-09

Running check with dropout =  0.25
Initial loss:  2.3136012862272435
W1 relative error: 3.55e-08
W2 relative error: 4.62e-08
W3 relative error: 2.16e-08
b1 relative error: 1.02e-07
b2 relative error: 2.10e-08
b3 relative error: 3.47e-09

Running check with dropout =  0.5
Initial loss:  2.312331406530074
W1 relative error: 1.94e-08
W2 relative error: 1.82e-08
W3 relative error: 1.03e-08
b1 relative error: 5.34e-08
b2 relative error: 1.13e-08
b3 relative error: 3.86e-09
```

由結果可以看到，對於不同 dropout 機率，各個參數的誤差都非常小，且會根據不同的 dropout 機率做改變，代表說 dropout 的運作會根據 dropout 機率變動而調整，且整體功能正確。

- Regularization experiment

  對不同的神經網路做訓練，來查看 dropout 是否真的有 regularization 的功能。這邊對三個神經網路做訓練，網路架構分別為

  1. Hidden size = 256，dropout = 0
  2. Hidden size = 512，dropout = 0
  3. Hidden size = 512，dropout = 0.5

  訓練結果如下

```
Training a model with dropout=0.00 and width=256
(Time 0.02 sec; Iteration 1 / 3900) loss: 2.302505
(Epoch 0 / 100) train acc: 0.213000; val_acc: 0.212100
(Epoch 10 / 100) train acc: 0.710000; val_acc: 0.477600
(Epoch 20 / 100) train acc: 0.836000; val_acc: 0.477000
(Epoch 30 / 100) train acc: 0.934000; val_acc: 0.467800
(Epoch 40 / 100) train acc: 0.965000; val_acc: 0.462100
(Epoch 50 / 100) train acc: 0.994000; val_acc: 0.468200
(Epoch 60 / 100) train acc: 0.945000; val_acc: 0.459700
(Epoch 70 / 100) train acc: 0.995000; val_acc: 0.467700
(Epoch 80 / 100) train acc: 0.921000; val_acc: 0.456100
(Epoch 90 / 100) train acc: 0.992000; val_acc: 0.466700
(Epoch 100 / 100) train acc: 0.969000; val_acc: 0.455400

Training a model with dropout=0.00 and width=512
(Time 0.01 sec; Iteration 1 / 3900) loss: 2.301456
(Epoch 0 / 100) train acc: 0.185000; val_acc: 0.206800
(Epoch 10 / 100) train acc: 0.725000; val_acc: 0.471900
(Epoch 20 / 100) train acc: 0.867000; val_acc: 0.469600
(Epoch 30 / 100) train acc: 0.938000; val_acc: 0.464500
(Epoch 40 / 100) train acc: 0.974000; val_acc: 0.479400
(Epoch 50 / 100) train acc: 0.924000; val_acc: 0.464200
(Epoch 60 / 100) train acc: 0.954000; val_acc: 0.462600
(Epoch 70 / 100) train acc: 0.924000; val_acc: 0.448600
(Epoch 80 / 100) train acc: 0.970000; val_acc: 0.483700
(Epoch 90 / 100) train acc: 0.937000; val_acc: 0.462800
(Epoch 100 / 100) train acc: 0.970000; val_acc: 0.471600
```

```
Training a model with dropout=0.50 and width=512
(Time 0.00 sec; Iteration 1 / 3900) loss: 2.302781
(Epoch 0 / 100) train acc: 0.231000; val_acc: 0.239600
(Epoch 10 / 100) train acc: 0.608000; val_acc: 0.468300
(Epoch 20 / 100) train acc: 0.675000; val_acc: 0.480400
(Epoch 30 / 100) train acc: 0.784000; val_acc: 0.490400
(Epoch 40 / 100) train acc: 0.822000; val_acc: 0.498300
(Epoch 50 / 100) train acc: 0.845000; val_acc: 0.496700
(Epoch 60 / 100) train acc: 0.875000; val_acc: 0.495500
(Epoch 70 / 100) train acc: 0.883000; val_acc: 0.497700
(Epoch 80 / 100) train acc: 0.920000; val_acc: 0.495000
(Epoch 90 / 100) train acc: 0.920000; val_acc: 0.499400
(Epoch 100 / 100) train acc: 0.937000; val_acc: 0.493000
```

由上面結果可以看到幾個現象：

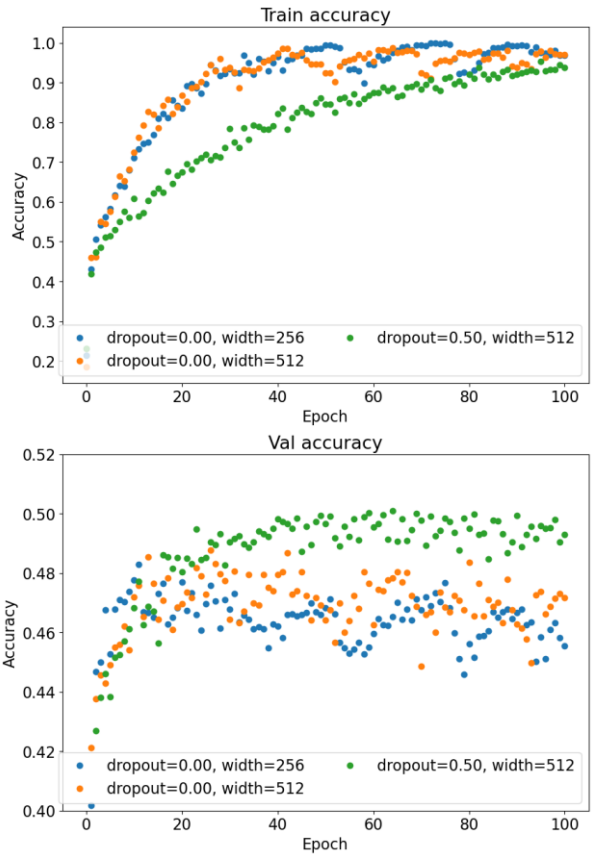
1. Hidden size 的影響：

   比較 hidden size = 256 與 hidden size = 512 結果可以發現，hidden size = 512 的準確度會高於 hidden size = 256，代表說較大的 hidden size 可以做到更多的非線性來更好的擬和訓練資料，使準確度提升。

2. Dropout 的影響：

   比較 hidden size = 512 以及相同 hidden size 但是 dropout = 0.5 的模型，會發現到對於訓練資料來說，dropout = 0 會有比較高的準確度，但是在 validation set 上，反而是 dropout = 0.5 的模型有比較高的準確度。會造成這樣的現象是因為 hidden size = 512、dropout = 0 的模型發生嚴重的 overfitting，導致說在 validation 上的表現不如預期。而在加入 dropout 後，overfitting 的程度被稍微改善了，因此在 validation 上的表現稍微提升了一些。由這次嘗試也可以看到，dropout 的功能就如同 regularization，可以防止 overfitting 的發生。

   將訓練過程視覺化後結果如下



由視覺化後結果可以更明顯看出，在 training accuracy 上，加入 dropout 表現最差，因為要防止 overfitting 所以對訓練資料的擬和程度不會像沒有 dropout 一樣高。但是在 validation accuracy 上，有 dropout 的模型表現最好，這便是 regularization 產生的效果。

十、 額外嘗試

- 嘗試重新訓練模型：

在 regularization experiment 的地方，使用三種不同模型做訓練，最終最佳的準確度落在 49 %，我嘗試重新調整參數，試圖提升準確度。這次我嘗試調整 hidden size 和 dropout 機率。Optimizer 使用 adam，learning rate = 5e-3，並對 40000 張訓練資料做訓練。從訓練過程，我觀察到幾個現象

1. Dropout 機率對訓練的影響

跟先前說明的一樣，dropout 確實有 regularization 的作用，剛才只有測試 dropout = 0.5 的部分，因此我嘗試加大 dropout 機率，結果如下

```
Training a model with dropout=0.70 and width=512
(Time 0.00 sec; Iteration 1 / 7800) loss: 2.303504
(Epoch 0 / 100) train acc: 0.257000; val_acc: 0.231000
(Epoch 10 / 100) train acc: 0.498000; val_acc: 0.451300
(Epoch 20 / 100) train acc: 0.561000; val_acc: 0.490000
(Epoch 30 / 100) train acc: 0.577000; val_acc: 0.495800
(Epoch 40 / 100) train acc: 0.619000; val_acc: 0.499400
(Epoch 50 / 100) train acc: 0.655000; val_acc: 0.496300
(Epoch 60 / 100) train acc: 0.659000; val_acc: 0.493600
(Epoch 70 / 100) train acc: 0.685000; val_acc: 0.503400
(Epoch 80 / 100) train acc: 0.695000; val_acc: 0.502000
(Epoch 90 / 100) train acc: 0.700000; val_acc: 0.510000
(Epoch 100 / 100) train acc: 0.739000; val_acc: 0.504700
```

由上圖可以看到，在一樣是 hidden size = 512 的情況下，持續加大 dropout 機率到 0.7 之後，training accuracy 來到只有 73.9 %，但是 validation 卻可以到 50 %左右，代表說加大 dropout 後 overfitting 的現象又再更進一步改善了。

Dropout 這麼大確實可以讓 training accuracy 和 validation accuracy 更加相近，但是這樣的結果就會有最高的 validation accuracy 嗎? 經過我的測試後，答案是否。我嘗試使用 dropout = 0.3 對相同的網路架構做訓練，結果如下

```
Training a model with dropout=0.30 and width=512
(Time 0.00 sec; Iteration 1 / 7800) loss: 2.301730
(Epoch 0 / 100) train acc: 0.220000; val_acc: 0.222100
(Epoch 10 / 100) train acc: 0.619000; val_acc: 0.511800
(Epoch 20 / 100) train acc: 0.708000; val_acc: 0.517700
(Epoch 30 / 100) train acc: 0.802000; val_acc: 0.520700
(Epoch 40 / 100) train acc: 0.838000; val_acc: 0.515300
(Epoch 50 / 100) train acc: 0.879000; val_acc: 0.521600
(Epoch 60 / 100) train acc: 0.916000; val_acc: 0.528200
(Epoch 70 / 100) train acc: 0.912000; val_acc: 0.515200
(Epoch 80 / 100) train acc: 0.932000; val_acc: 0.523100
(Epoch 90 / 100) train acc: 0.948000; val_acc: 0.507000
(Epoch 100 / 100) train acc: 0.921000; val_acc: 0.514900
```

由結果可以發現到，降低 dropout 機率後，validation accuracy 可以來到 52.3 %，比 dropout 機率為 0.7 時來的高。經由這次實驗可以得知，dropout 其實也是一種超參數，需要經過不斷嘗試來找尋最適合的數值。

2. 訓練最佳結果：

我嘗試調整的 hidden size 為 128、256、512、1024，dropout 機率為 0、0.3、0.5、0.7、0.9，經過多次訓練，最佳結果如下

```
Training a model with dropout=0.50 and width=1024
(Time 0.01 sec; Iteration 1 / 7800) loss: 2.305831
(Epoch 0 / 100) train acc: 0.216000; val_acc: 0.221400
(Epoch 10 / 100) train acc: 0.532000; val_acc: 0.464000
(Epoch 20 / 100) train acc: 0.653000; val_acc: 0.495500
(Epoch 30 / 100) train acc: 0.684000; val_acc: 0.504100
(Epoch 40 / 100) train acc: 0.748000; val_acc: 0.516200
(Epoch 50 / 100) train acc: 0.776000; val_acc: 0.512800
(Epoch 60 / 100) train acc: 0.802000; val_acc: 0.508200
(Epoch 70 / 100) train acc: 0.834000; val_acc: 0.521800
(Epoch 80 / 100) train acc: 0.866000; val_acc: 0.522300
(Epoch 90 / 100) train acc: 0.884000; val_acc: 0.523500
(Epoch 100 / 100) train acc: 0.873000; val_acc: 0.529100
```

最佳結果發生在 hidden size = 1024，dropout 機率為 0.5。醉雞的準確度為 52.91%。

- 嘗試使用一層 CNN + 一層全連接層：

由於教授上課已經交到 CNN 的部分，因此我嘗試利用 tensorflow 來建立一個 CNN，並與這次作業中的的模型做比較。

1. CNN 架構：

這次嘗試建構的 CNN 架構為一層卷積層加上一層全連接層。詳細架構如下圖

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 conv2d_1 (Conv2D)            (None, 29, 29, 32)        1568

 flatten_1 (Flatten)          (None, 26912)             0

 dense_1 (Dense)              (None, 10)                269130

=================================================================
Total params: 270698 (1.03 MB)
Trainable params: 270698 (1.03 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

2. 訓練結果：

```
Epoch 1/20
1563/1563 [==============================] - 8s 5ms/step - loss: 1.4994 - accuracy: 0.4709 - val_loss: 1.3716 - val_accuracy: 0.5188
Epoch 2/20
1563/1563 [==============================] - 7s 5ms/step - loss: 1.2398 - accuracy: 0.5704 - val_loss: 1.2940 - val_accuracy: 0.5454
Epoch 3/20
1563/1563 [==============================] - 7s 5ms/step - loss: 1.1285 - accuracy: 0.6100 - val_loss: 1.2239 - val_accuracy: 0.5684
Epoch 4/20
1563/1563 [==============================] - 7s 5ms/step - loss: 1.0487 - accuracy: 0.6345 - val_loss: 1.1981 - val_accuracy: 0.5837
Epoch 5/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.9613 - accuracy: 0.6701 - val_loss: 1.2735 - val_accuracy: 0.5679
Epoch 6/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.8800 - accuracy: 0.6924 - val_loss: 1.2021 - val_accuracy: 0.5988
Epoch 7/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.8067 - accuracy: 0.7216 - val_loss: 1.2515 - val_accuracy: 0.5910
Epoch 8/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.7464 - accuracy: 0.7425 - val_loss: 1.3036 - val_accuracy: 0.5827
Epoch 9/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.6873 - accuracy: 0.7639 - val_loss: 1.2957 - val_accuracy: 0.5893
Epoch 10/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.6341 - accuracy: 0.7814 - val_loss: 1.3537 - val_accuracy: 0.5978
Epoch 11/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.5853 - accuracy: 0.7986 - val_loss: 1.4181 - val_accuracy: 0.5885
Epoch 12/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.5416 - accuracy: 0.8136 - val_loss: 1.4267 - val_accuracy: 0.5946
Epoch 13/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.5049 - accuracy: 0.8254 - val_loss: 1.5239 - val_accuracy: 0.5909
Epoch 14/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.4657 - accuracy: 0.8398 - val_loss: 1.6574 - val_accuracy: 0.5789
Epoch 15/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.4287 - accuracy: 0.8529 - val_loss: 1.6074 - val_accuracy: 0.5889
Epoch 16/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.4009 - accuracy: 0.8650 - val_loss: 1.6815 - val_accuracy: 0.5823
Epoch 17/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.3654 - accuracy: 0.8764 - val_loss: 1.7331 - val_accuracy: 0.5895
Epoch 18/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.3384 - accuracy: 0.8882 - val_loss: 1.8365 - val_accuracy: 0.5842
Epoch 19/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.3135 - accuracy: 0.8956 - val_loss: 1.9134 - val_accuracy: 0.5713
Epoch 20/20
1563/1563 [==============================] - 7s 5ms/step - loss: 0.2946 - accuracy: 0.9027 - val_loss: 1.9609 - val_accuracy: 0.5828
```

由結果可以看出，最佳的準確度為 59.88 %，將近有六成的準確度。

3. 結果討論：

由訓練結果可以看出，CNN 的準確度比一層全連接層的準確度高上許多，此外，教授上課時有提到，使用卷積層的話通常不會加上 dropout layer，因為卷積層本商就具有對 overfitting 的抵抗力，經過這次嘗試也證明了教授所言。在我自己建立的 CNN 架構中，沒有加入 dropout layer，但是最後 training accuracy 只有 90.27%，而 validation accuracy 卻來到 58.28%。與全連接層加上 dropout 的結果進行比較，overfitting 的現象甚至還沒那麼嚴重。因此我認為

在實際應用中，dropout 並非那麼實用，與其使用 dropout 加上全連接層，不如直接使用卷積層。

- Optimizer :

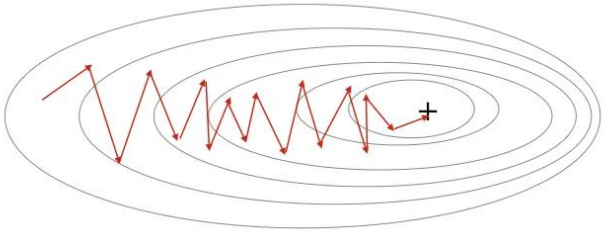
  由於這次作業有做到 sgd + momentum 的實現，因此我希望可以多加了解一下 optimizer 的原理與種類。

  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。

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

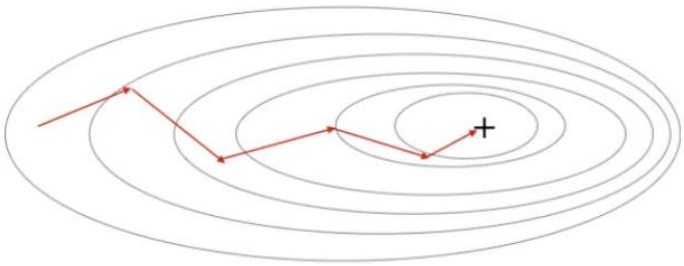
     - 優點：很好理解，實做起來簡單。
     - 缺點：由於他一次要計算所有訓練資料的梯度並一次更新所有參數，因此計算速度很慢，而且需要花費很多記憶體空間。

  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。

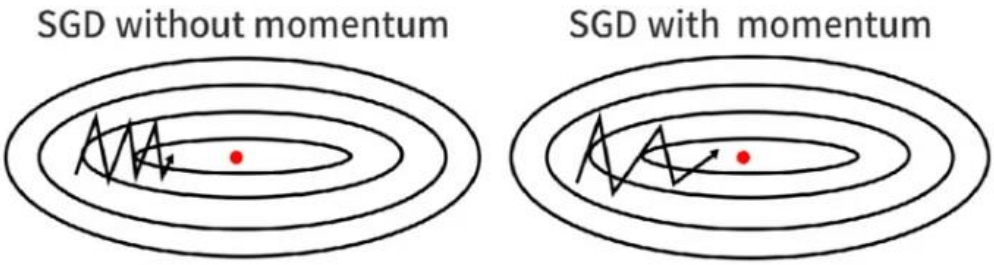
  3. Mini-Batch Gradient Descent : 將訓練資料分成 batches 並對這些 batches 做參數更新。



     - 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。

十一、 Reference

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

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

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

[4] Devashree Madhugiri “Using CNN for Image Classification on CIFAR-10 Dataset” https://devashree-madhugiri.medium.com/using-cnn-for-image-classification-on-cifar-10-dataset-7803d9f3b983