National Sun Yat-sen University
Department of Electrical Engineering

112-2 機器學習系統設計實務與應用

HW2 Keras神經網路應用

姓名: 胡庭翊

學號: B103015006

組別: 你說的隊

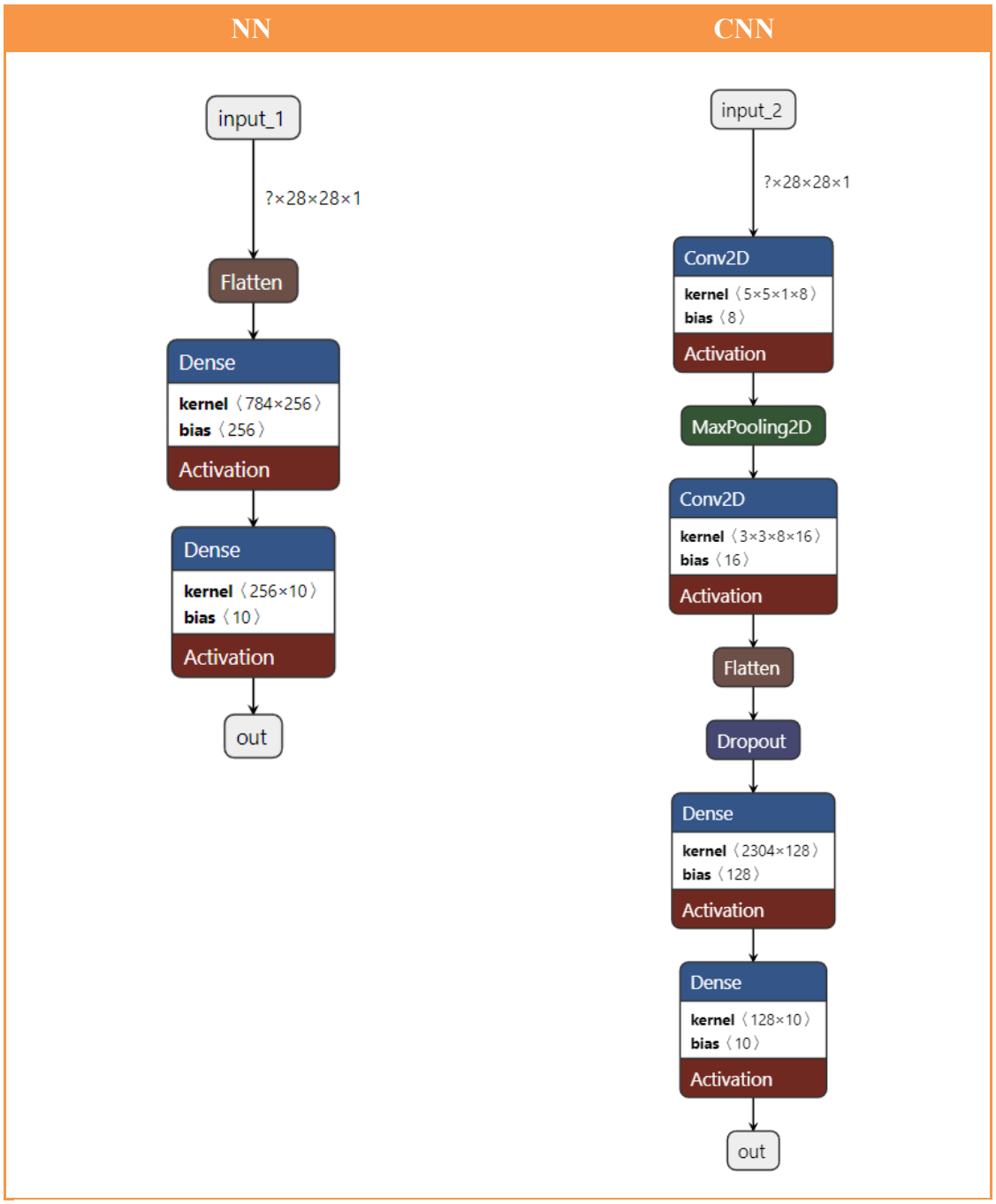
1. Dataset MNIST

以下會針對 TensorFlow 官方提供的手寫數字資料集 MNIST 建構不同的 NN 及 CNN 架構訓練，並且從不同可能變因探討架構之間不同有何影響。

1.1. 對照組

首先先就範本內的初始架構做預測：

1.1.1. 網路架構



表一、對照組網路架構

1.1.1.1.NN 網路架構分析

這裡的 NN 模型首先是先將訓練與測試資料從三圍轉為四維，接著再使用網路架構，將三維的輸入轉換成一維後，使用 ReLU 做為激勵函數增加類神經網路內的非線性特徵，再使用 Softmax 放置在 hidden layer 的最後一層。

ReLU 的優點是比起其他函數更能解決梯度消失的問題，計算量小較節省資源，以及其全有全無的特性更接近生物神經網路的模型。其中，梯度消失指的是當建立深層的類神經網路時，在接近輸出端的類神經元更新速度與接近輸入端的類神經元速度不一致，而導致接近輸入層的網路無法作用，實際上整個網路架構只有少數層運作，失去深度學習的意義。

Softmax 回歸是使用 Softmax 運算使得最後一層輸出的機率分佈總和為 1，將最後一層的所有節點輸出都通過指數函數，再將結果相加作為分母、個別輸出作為分子，通常都會放置在類神經網路的最後一層。

1.1.1.2.CNN 網路架構分析

至於 CNN 模型結構則是由兩層卷積層加上池化層，後面再加上與 NN 架構相似的全連接層組成，使用 Keras 高階封裝的神經框架，搭建網路透過 tensorflow 運作。

範本中首先經過卷積層 Conv2D(8, (5, 5), strides = (1, 1), activation = 'relu', name = 'conv1',padding='same')(X_input)，使用 8 個 5x5 的 filter 配合固定的權重做到影像處理，提取特徵。並且將 strides 設為(1, 1)，意指卷積在圖像上更新的 xy 方向移動部署是所有的點逐格掃描。同時，除了增加 ReLU 激勵函式外，也使用 padding 補邊方法，針對輸入的維度做擴增，使得最後的輸出結果與原輸入大小相同。

池化層的部分 MaxPooling2D((2, 2), name='max_pool1_W1')(W1)使用最大值方法 maxpool-pooling，保留每個 2x2 資訊中的最大值，再將其餘資訊丟棄，將資訊壓縮簡化以增加模型的收斂及穩定性。

最後再經過一次的卷積層 Conv2D(16, (3, 3), strides = (1, 1), activation = 'relu', name = 'conv2')(W1) ，使用 16 層 3x3 的 filter 做影像處理。除了 filter 的層數與大小，還有這次沒有使用 padding 之外，其餘都與前面的卷積層相仿，而後續的全連接層架構亦類似於上面的 NN 架構。

1.1.2. 超參數設置

超參數指的是訓練神經網路時需要手動設定的參數。這裡，我將每次迭代時從訓練集中提取，送入神經網路的資料數量設為 44000，提取其中的 0.05 作為訓練資料的比例，其中的 0.2 作為驗證的資料比例，進行總計 250 次的迭代訓練。也就是說，共有 2200 份資料做為資料集，8800 份資料作為驗證集。

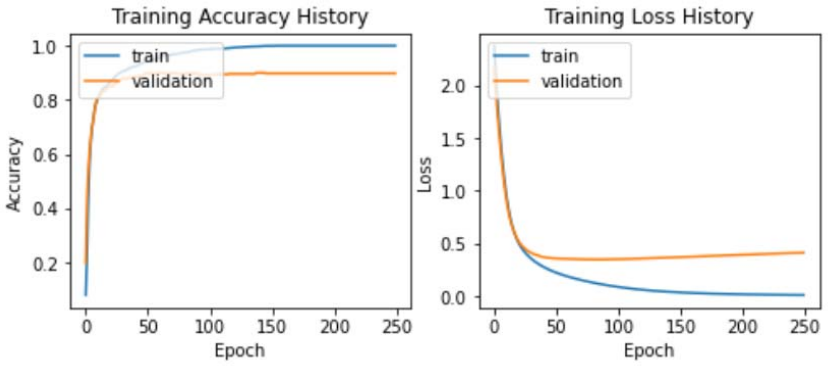
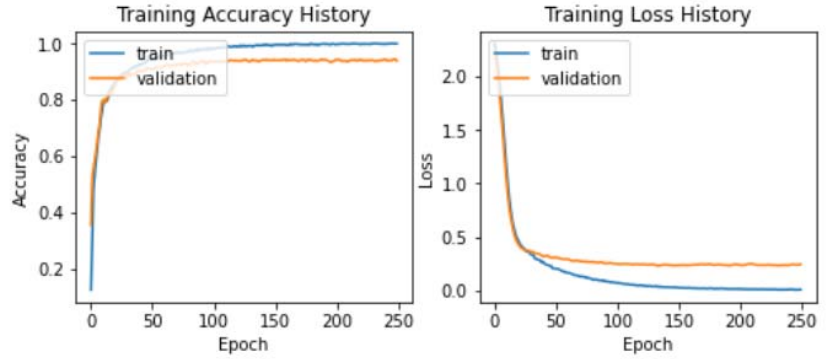
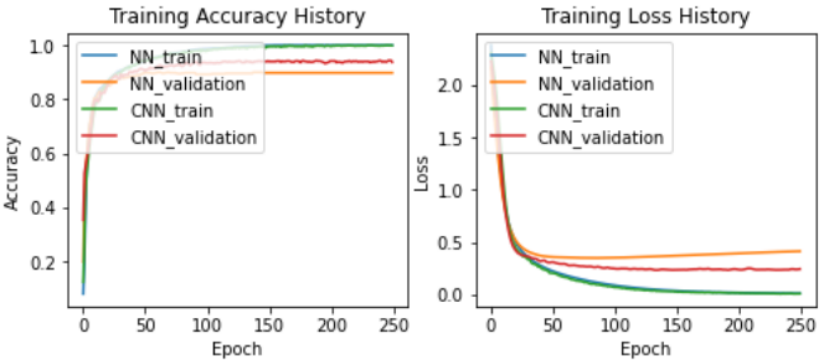
```
# In[2]:   Hyperparameter

BATCH_SIZE = 44000 #44000 #change there
NUM_EPOCHS = 250
TRAIN_DATA = 0.05
VAL = 0.2
```

圖一、對照組超參數設置

1.1.3. 準確度

1.1.3.1.訓練

NN	 Test loss : 0.41491 Test accuracy : 0.89700
CNN	 Test loss : 0.17929 Test accuracy : 0.95200
NN、CNN 比較	

表二、對照組訓練準確度

損失函數介於 0 到 1 之間，用以評估兩個機率分配有多接近。而訓練模型的目的便是讓損失函數最小化。

從結果可知，NN 測試表現損失函數較大(0.41491)，而 CNN 的損失函數較小(0.17929)，這也反映到了測試結果的正確性。另外，從結果圖可以發現 CNN 模型在做跌代時數值也有微幅的震盪，以訓練精準度而言，CNN 模型可以更精確地尋找損失函數的最低值，不會在遇到 local minima 時便誤以為是 global minima，然後停止繼續搜尋。

1.1.3.2.測試

NN	10000 picture total wrong = 1021 image accuracy = 0.8979 average per execute time: 0.379534 ms
CNN	10000 picture total wrong = 480 image accuracy = 0.952 average per execute time: 0.672524 ms

表三、對照組測試準確度

以 10000 張照片實際上測試後，我們可以發現 CNN 表現顯優於 NN，錯誤照片數 CNN 幾乎是 NN 的一半，不過也因為 CNN 模型複雜許多，所以運作時間是 NN 的兩倍。

1.2. Activation Function

除了 ReLU，常見的激勵函數還有 Sigmoid 以及 tan h。維持與對照組相同的網路架構以及超參數，並且控制最靠近輸出端的 hidden layer 依舊使用 Softmax，我先就激勵函數進行替換與分析:

1.2.1. Sigmoid

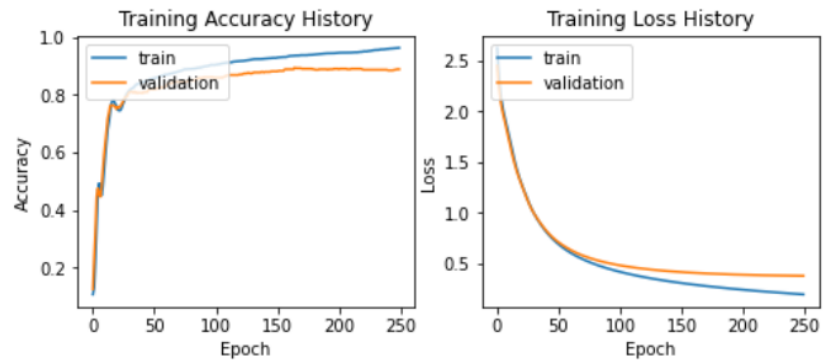
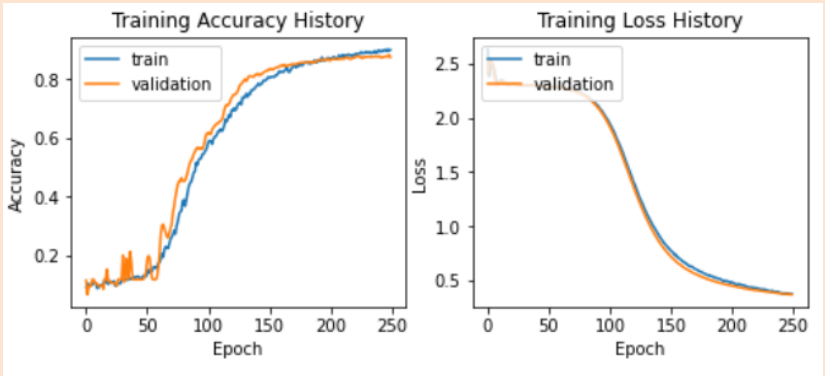
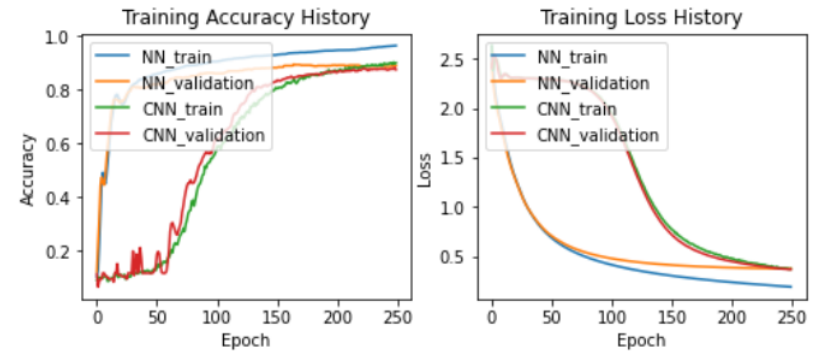
Sigmoid 函數也叫 Logistic 函數，是最早被使用，使用範圍也最廣的激勵函數。它可以將一個實數映射到[0, 1] 的區間，因此也可以被用來做二分類。

Sigmoid 的優點是能將輸出範圍限於 0 到 1 之間，並且 Sigmoid 是連續變數，便於求導。不過，Sigmoid 存在著梯度消失的大問題，而且 Sigmoid 函數有著 zero center 的現象。當後面神經元的輸入皆為正數時對權重值求梯度，那麼梯度數值恆為正，因此在誤差反向傳遞的過程中，權重都正方向更新或往負方向更新，導致收斂曲線不平滑，形成綑綁的現象，也影響模型的收斂速度。除此之外，Sigmoid 的運算效能也低於 ReLU。

以下是使用 Sigmoid 方法進行的訓練與測試效果:

1.2.2. 準確度

1.2.2.1.訓練

NN	 Test loss : 0.41070 Test accuracy : 0.88130
CNN	 Test loss : 0.41383 Test accuracy : 0.88180
NN、CNN 比較	

表四、Sigmoid 訓練準確度

從結果圖可以驗證，使用 Sigmoid 時訓練產生了綑綁現象。NN 模型因為結構還沒有那麼複雜，所以影響程度還沒有到很嚴重，正確度與使用 ReLU 建構的 NN 模型差異並不是很大。然而，換到層數多的 CNN 時，Sigmoid 的綑綁現象已經大到影響模型的收斂曲線，不僅收斂曲線不平滑，最終的訓練正確度也因此大打折扣，訓練成效與 NN 模型相異無幾。

1.2.2.2.測試

NN	10000 picture total wrong = 1187 image accuracy = 0.8813 average per execute time: 0.474311 ms
CNN	10000 picture total wrong = 1182 image accuracy = 0.8818 average per execute time: 0.807964 ms

表五、Sigmoid 測試準確度

以 10000 張照片實際上測試後，從結果可以驗證 Sigmoid 的運算效能低於 ReLU 的運算效能，以 MNIST 訓練的表現來看，Sigmoid 的效能約為 ReLU 的 4/3 倍。另外，雖然因為 CNN 結構較複雜導致運算速度為 NN 模型的兩倍，但就使用 Sigmoid 作為激勵函數而言，除非增加 epoch 的次數，不然建構 CNN 模型的必要性比較沒有那麼大。

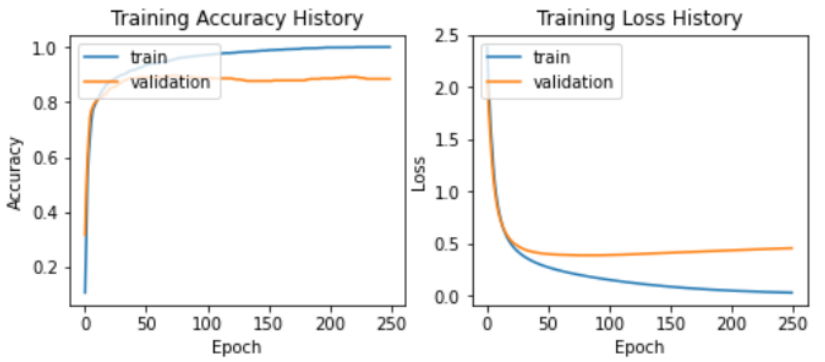
1.2.3. tan h

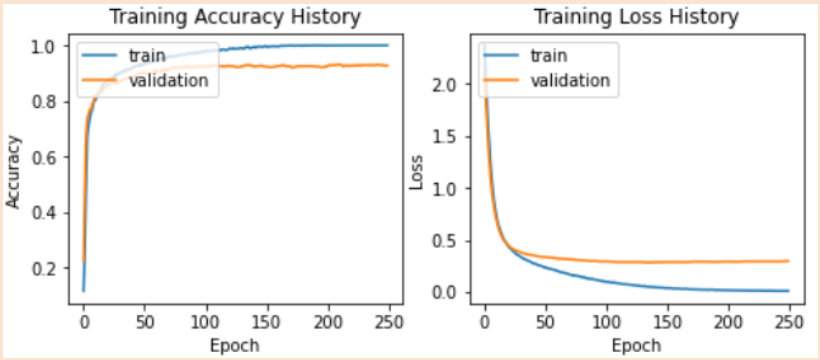
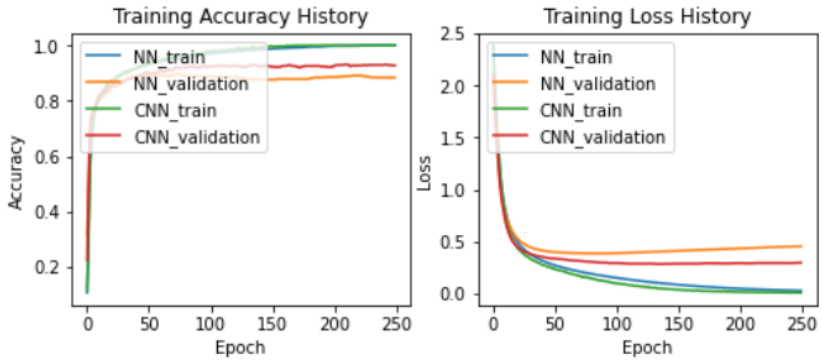
tan h 激勵函數為 Sigmoid 的變形，順利解決 Sigmoid 函數中 zero-centered 的輸出問題，中間值為 0 的特性使得 tan h 比起 Sigmoid 更適合作為激勵函數提供給下層神經元學習。tan h 的優點是收斂速度比 Sigmoid 還快，然而其梯度消失問題以及需要指數運算的問題依然存在。

以下是使用 Sigmoid 方法進行的訓練與測試效果:

1.2.4. 準確度

1.2.4.1.訓練

NN	 Test loss : 0.47279 Test accuracy : 0.88250

CNN	 Test loss : 0.26398 Test accuracy : 0.92580
NN、CNN 比較	

表六、tan h 訓練準確度

從結果圖可以驗證， tan h 方法比起 Sigmoid 收斂曲線更為穩定，在 CNN 模型中收斂曲線的穩定速度更是有明顯的改善,也因此正確率也增加了 0.4 左右，同時 NN 的正確率也相較於使用 Sigmoid 有微幅的成長。

1.2.4.2.測試

NN	10000 picture total wrong = 1175 image accuracy = 0.8825 average per execute time: 0.595085 ms
CNN	10000 picture total wrong = 742 image accuracy = 0.9258 average per execute time: 0.968436 ms

表七、tan h 測試準確度

使用 10000 張照片實際上測試後，從結果也可以驗證 tan h 的運算正確性高於 Sigmoid，不過還是低於 ReLU 的表現。以效能來說，tan h 低於 Sigmoid 也低於 ReLU。以 MNIST 訓練的表現為例，tan h 的效能約為 ReLU 的 5/3 倍。

1.2.5. 小結

從正確性以及效能評比 ReLU、Sigmoid、tan h 三種函數，我認為 ReLU 最適合作為我們的激勵函數。

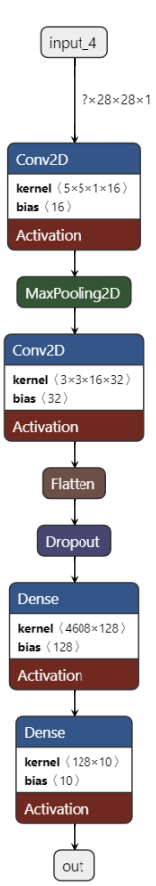
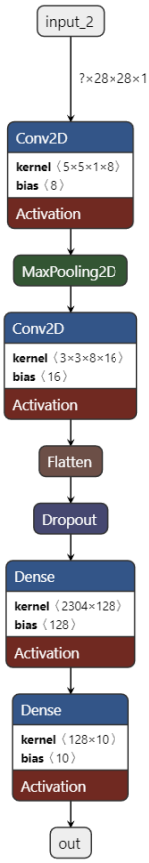
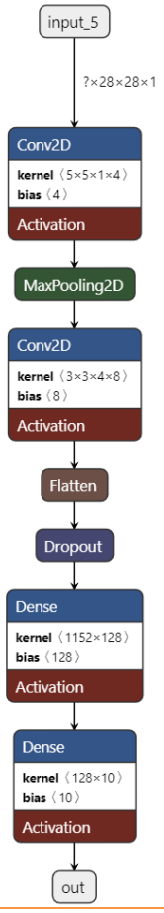
激勵函數	正確率	效能
ReLU	第一	第一
Sigmoid	第三	第二
tan h	第二	第三

表八、三種激勵函數的評比排名(由高至低)

1.3. Layer of Network

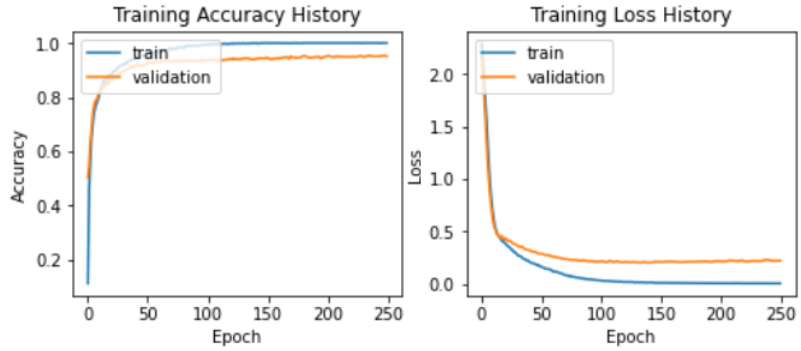
CNN 結構主要分成: 卷積層 (Convolution layer)、池化層 (Pooling layer) 以及最後一個全連接層 (Fully Connected layer)。雖然使用 MNIST 資料庫進行手寫字辨識時可以直接使用全連接層，也就是 NN 的架構進行訓練，但是更多的時候光是使用全連接層是不夠的，在進行較複雜的圖像辨識時，CNN 有助於分類與提高正確性，計算高像素圖片時使用 CNN 也能降低計算成本。以下將從 CNN 卷積層的 filter 數、filter 大小、stride 步伐，以及池化層的 pooling 方式做初步的比較:

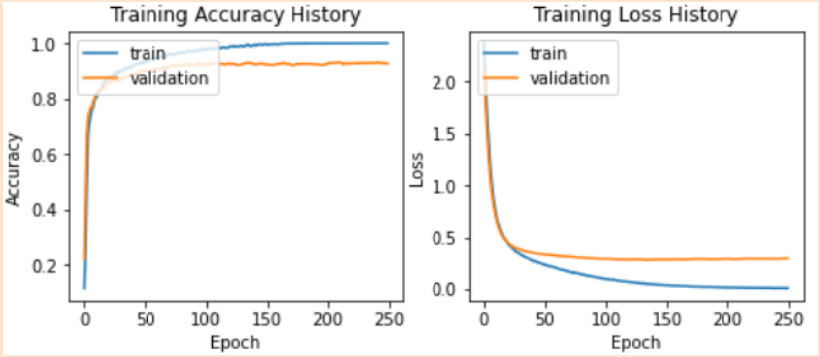
1.3.1. Filter 數

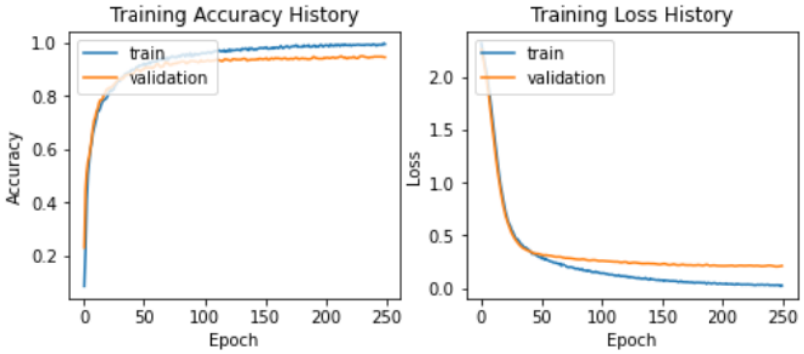
2 倍	1 倍	0.5 倍
		

表九、更改 filter 層數的網路架構

首先，我先控制網路架構的其餘變因，單就卷積層的層數做比較，將層數調整為 2 倍以及 0.5 倍。

<table>
<tr><td>2 倍</td><td>Test loss : 0.16883 Test accuracy : 0.95930
 

10000 picture total wrong = 407
image accuracy = 0.9593
average per execute time: 1.293529 ms</td></tr>
<tr><td>1 倍</td><td>Test loss : 0.26398 Test accuracy : 0.92580
 

10000 picture total wrong = 480
image accuracy = 0.952
average per execute time: 0.672524 ms</td></tr>
<tr><td>0.5 倍</td><td>Test loss : 0.17361 Test accuracy : 0.95250
 

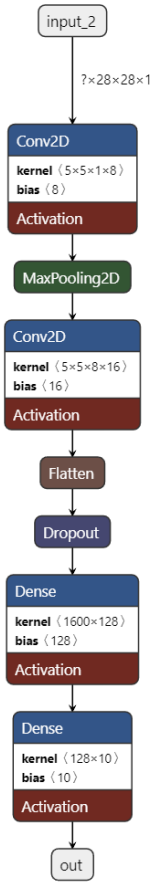
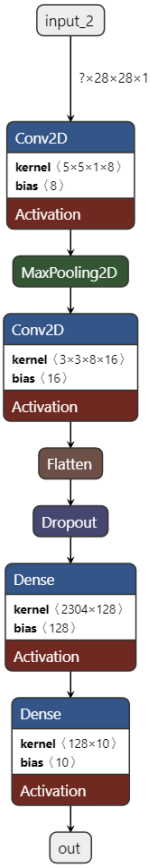
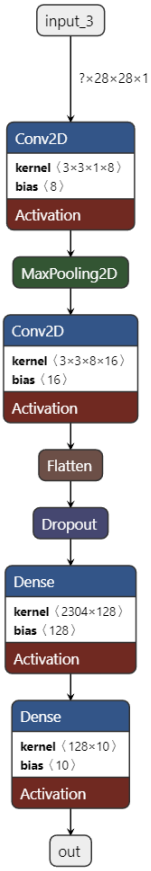
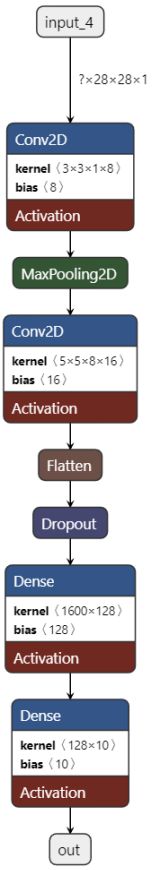
10000 picture total wrong = 475
image accuracy = 0.9525
average per execute time: 1.140163 ms</td></tr>
</table>

表十、更改 filter 層數的訓練與測試結果

從訓練與測試結果可以得知，若將卷積層層數增加為範本的兩倍，訓練的正確度可以提升 0.007，而若將層數減少兩倍，訓練的模型正確性也增加了 0.0005。

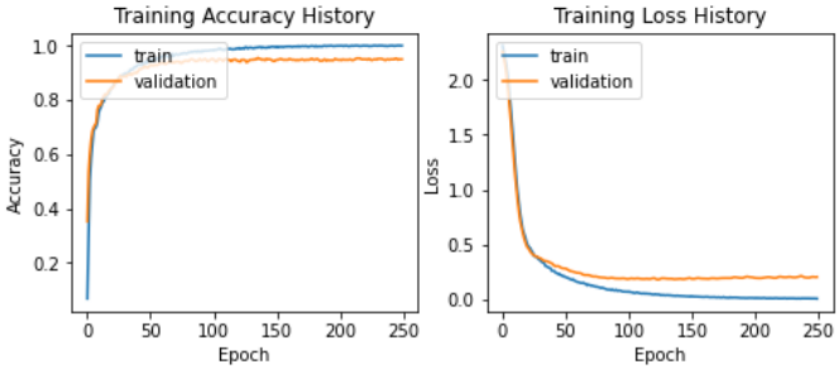
不過，不管是增加兩倍還是減少兩倍 filter 的數量，都會導致效率降低，測試時間都增加為原本對照組的兩倍。另外，若將 filter 層數增加太大會導致訓練與測試的時間增加，也容易導致 overfitting 的發生。

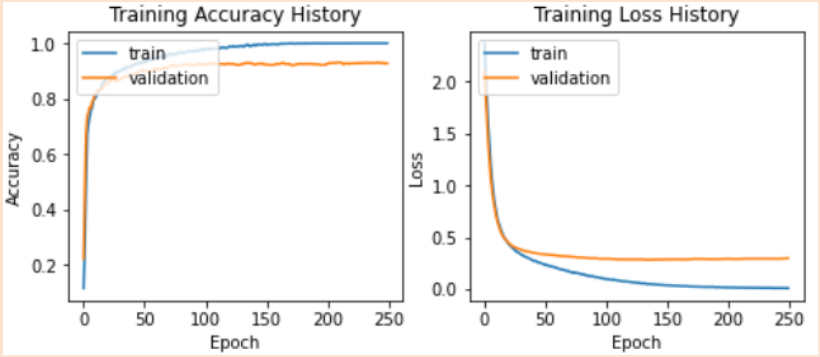
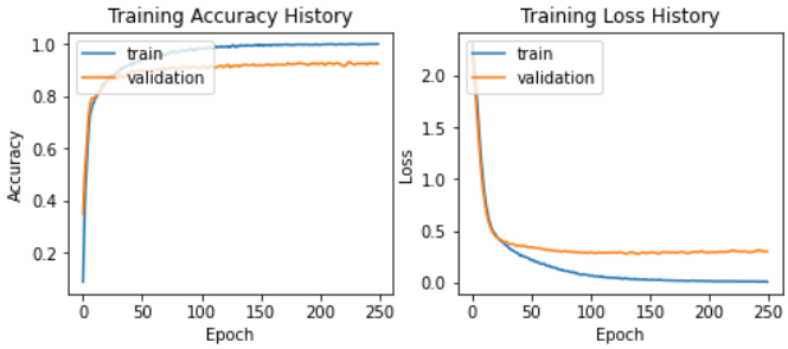
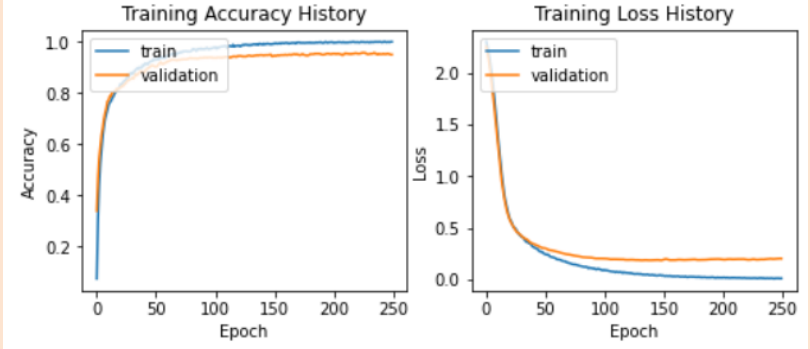
1.3.2. Filter 大小

(5x5)、(5x5)	(5x5)、(3x3)	(3x3)、(3x3)	(3x3)、(5x5)
			

表十一、更改 filter 大小的網路架構

再來，我將 filter 的大小調整成(5x5)、(5x5)；(3x3)、(3x3)；(3x3)、(5x5)與對照組的(5x5)、(3x3)比較。

(5x5)、(5x5)	Test loss : 0.15057 Test accuracy : 0.96140 

	10000 picture total wrong = 386 image accuracy = 0.9614 average per execute time: 1.022703 ms
(5x5)、(3x3)	Test loss : 0.26398 Test accuracy : 0.92580  10000 picture total wrong = 480 image accuracy = 0.952 average per execute time: 0.672524 ms
(3x3)、(3x3)	Test loss : 0.21959 Test accuracy : 0.94310  10000 picture total wrong = 569 image accuracy = 0.9431 average per execute time: 1.156285 ms
(3x3)、(5x5)	Test loss : 0.14364 Test accuracy : 0.95720  10000 picture total wrong = 428 image accuracy = 0.9572 average per execute time: 1.218010 ms

表十一、更改 filter 大小的訓練與測試結果

從訓練與測試結果可以得知，當 filter 大小設為(5x5)、(5x5) 與範本相比可以提升 0.007 的正確性，但要犧牲 0.33 毫秒左右的執行效率；當 filter 大小設為(3x3)、

(3x3)降低了 0.01 的正確性，也增加了兩倍左右的執行效率；當 filter 大小設為(5x5)、(3x3)增加了 0.014 的正確性，不過也增加了兩倍左右的執行效率。

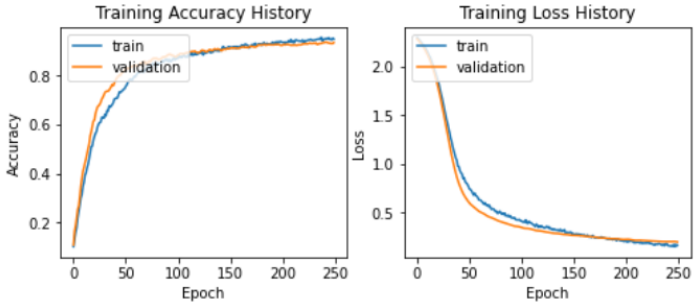
1.3.3. Stride 步伐

這裡，我嘗試將 strides 從原本的(1, 1)調整成(2, 2)，也就是將 filter 滑動的速度在 x，y 方向都增加兩倍。

strides = (2, 2)

```
W1 = Conv2D(8, (5, 5), strides = (2, 2), activation = 'relu', name = 'conv1',padding='same')(X_input)
W1 = MaxPooling2D((2, 2), name='max_pool1_W1')(W1)
W1 = Conv2D(16, (3, 3), strides = (2, 2), activation = 'relu', name = 'conv2')(W1)
```

```
Test loss :   0.19129  Test accuracy :  0.93970
```

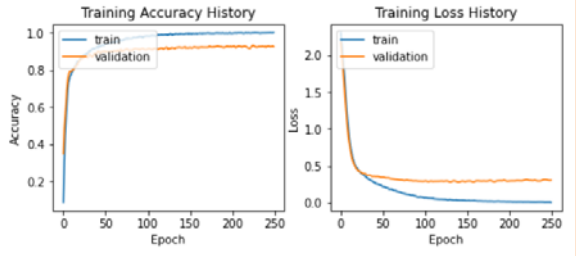



```
10000 picture total wrong = 603
image accuracy =  0.9397
average per execute time: 0.902723 ms
```

strides = (1, 1)

```
W1 = Conv2D(8, (5, 5), strides = (1, 1), activation = 'relu', name = 'conv1',padding='same')(X_input)
W1 = MaxPooling2D((2, 2), name='max_pool1_W1')(W1)
W1 = Conv2D(16, (3, 3), strides = (1, 1), activation = 'relu', name = 'conv2')(W1)
```

```
Test loss :   0.21959  Test accuracy :  0.94310
```



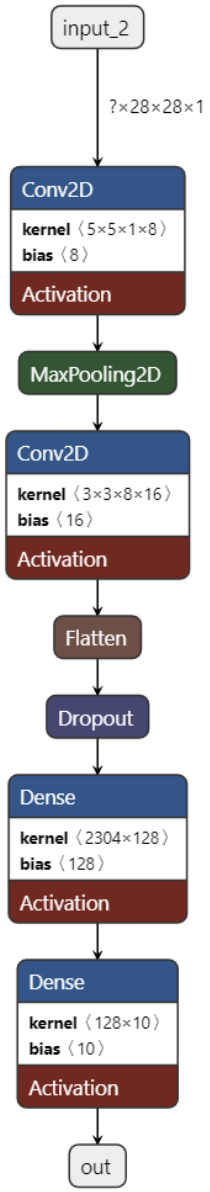
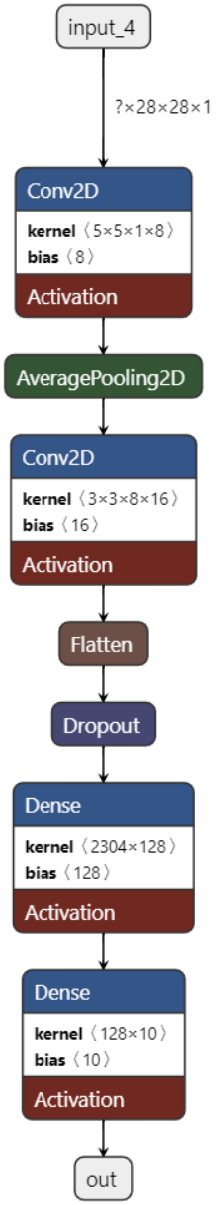

```
10000 picture total wrong = 480
image accuracy =  0.952
average per execute time: 0.672524 ms
```

表十二、更改 strides 大小的訓練與測試結果

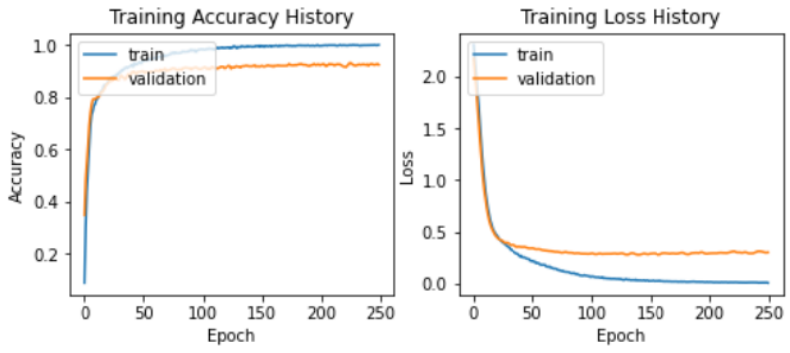
從訓練與測試結果可以得知，增加 filter 滑動的速度不僅不一定能增加模型訓練的正確性，效能也不一定能提升。不過這並不代表滑動速度的增加就一定對模型的訓練無益，只能說滑動速度增加為(2, 2)對 MNIST 資料集的訓練不如(1, 1)的速度。

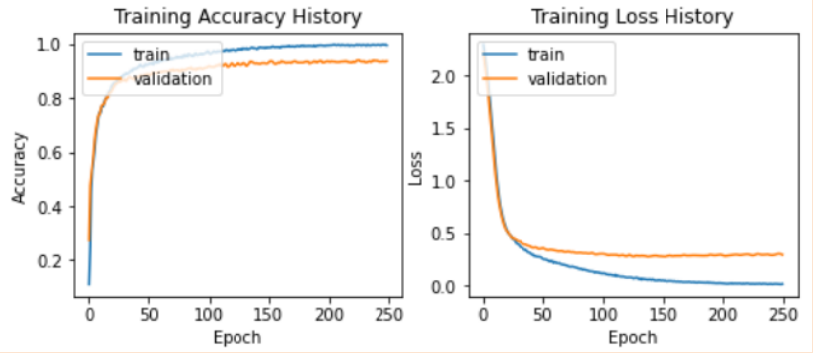
1.3.4. Pooling 方式

池化層的運算方法除了 MaxPooling 求 filter 中的最大值之外，也能使用 AveragePooling 求得 filter 範圍中的平均值，以下將兩種 pooling 方式做比較:

MaxPooling	AveragePooling
	

表十三、兩種不同的池化方式

MaxPooling	Test loss : 0.21959 Test accuracy : 0.94310 

	10000 picture total wrong = 480 image accuracy = 0.952 average per execute time: 0.672524 ms
AveragePooling	Test loss : 0.23996 Test accuracy : 0.94130  10000 picture total wrong = 587 image accuracy = 0.9413 average per execute time: 1.265115 ms

表十三、更改池化層的訓練與測試結果

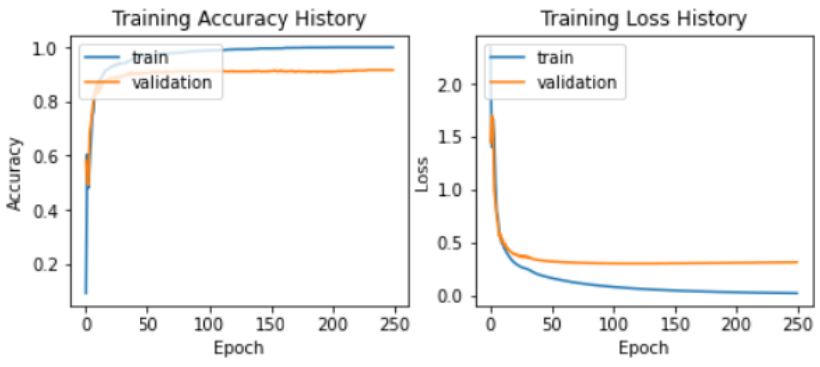
從訓練與測試結果可以得知，不管是正確率還是執行效能，MaxPooling 都表現得比 AveragePooling 還要好，事實上，現在大多數的 CNN 模型也是使用 MaxPooling 比較多，從 MNIST 模型的訓練結果又將此結果做了一次驗證。

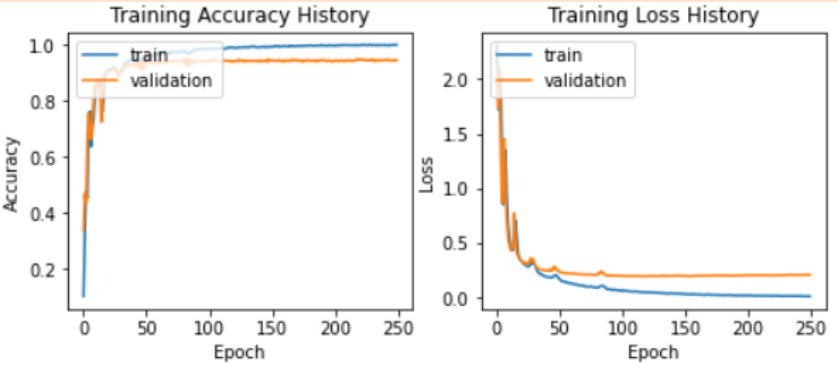
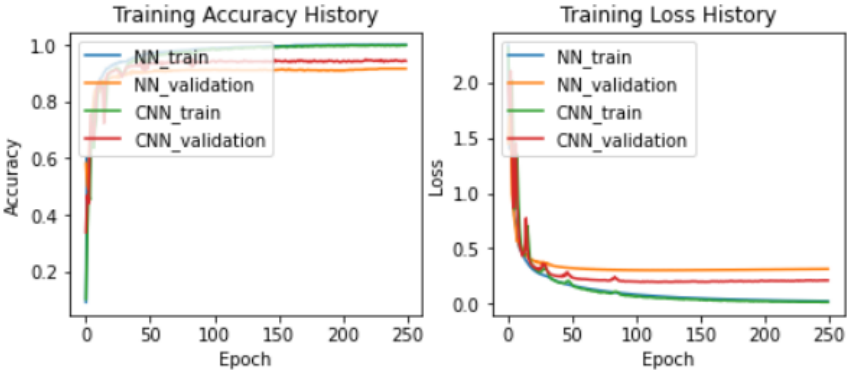
1.3.5. Optimizer

常見的優化器有 AdaGrad，RMSProp，及 Adam 等。他們都是基於梯度下降改良的版本，一般來說都有助於訓練模型。範本中我們使用的是 Adam，那麼接下來我會就另外兩種優化器進行測試。

1.3.5.1.AdaGrad

1.3.5.1.1. 訓練

NN	 Test loss : 0.34150 Test accuracy : 0.90710

CNN	 Test loss : 0.17845 Test accuracy : 0.95310
NN、CNN 比較	

表十四、訓練準確度

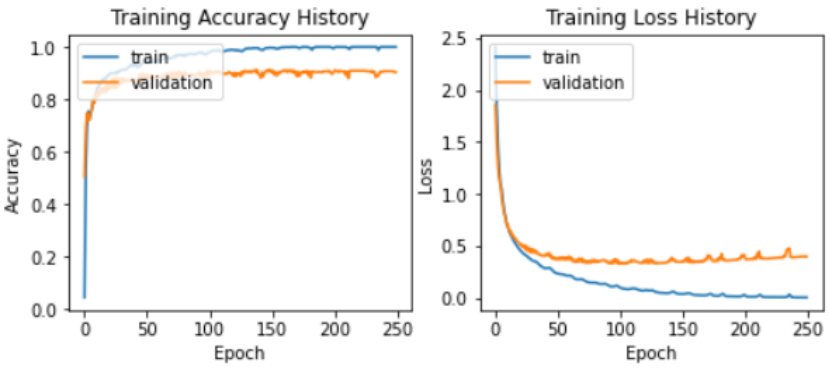
1.3.5.1.2. 測試

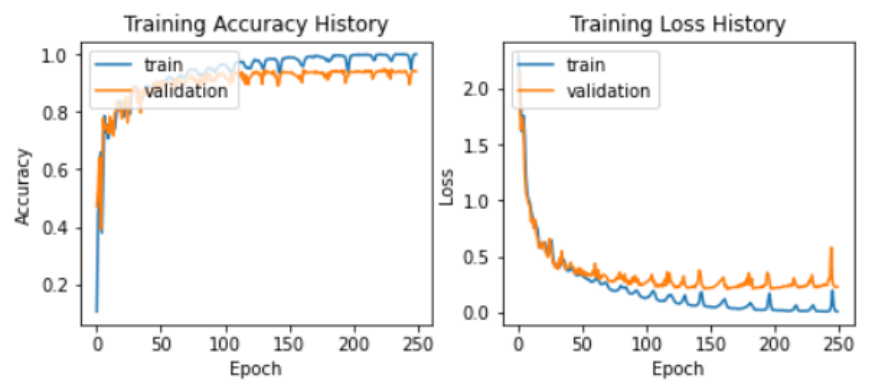
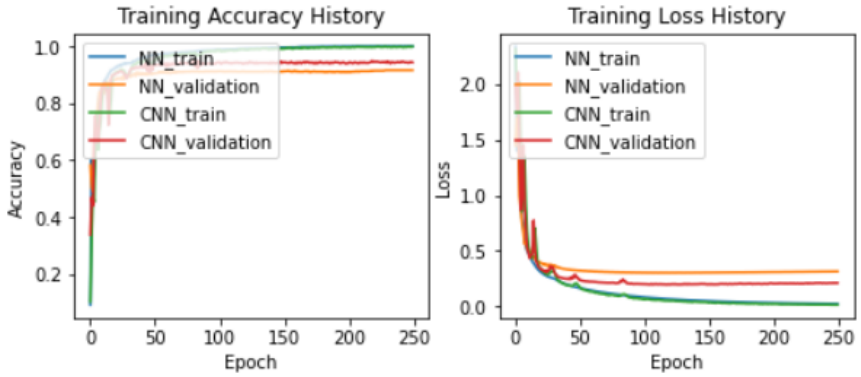
NN	10000 picture total wrong = 929 image accuracy = 0.9071 average per execute time: 0.776817 ms
CNN	10000 picture total wrong = 469 image accuracy = 0.9531 average per execute time: 0.821827 ms

表十五、測試準確度

1.3.5.2.RMSProp

1.3.5.2.1. 訓練

NN	

	Test loss : 0.47129 Test accuracy : 0.89470
CNN	 Test loss : 0.19516 Test accuracy : 0.94970
NN、CNN 比較	

表十六、訓練準確度

1.3.5.2.2. 測試

NN	10000 picture total wrong = 1053 image accuracy = 0.8947 average per execute time: 0.791603 ms
CNN	10000 picture total wrong = 503 image accuracy = 0.9497 average per execute time: 1.186966 ms

表十七、測試準確度

綜合以上結果，我認為 AdaGrad 綜合表現最佳。

2. 高鐵驗證碼

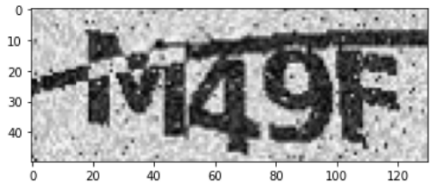
2.1. 預處理方式

範本內總共提供了三種預處理選項供選擇，分別是不做預處理的 nopre，去噪方法一 dn1，以及去噪方法 dn2。下面將呈現三種方法的處理結果與成效:

2.1.1. nopre

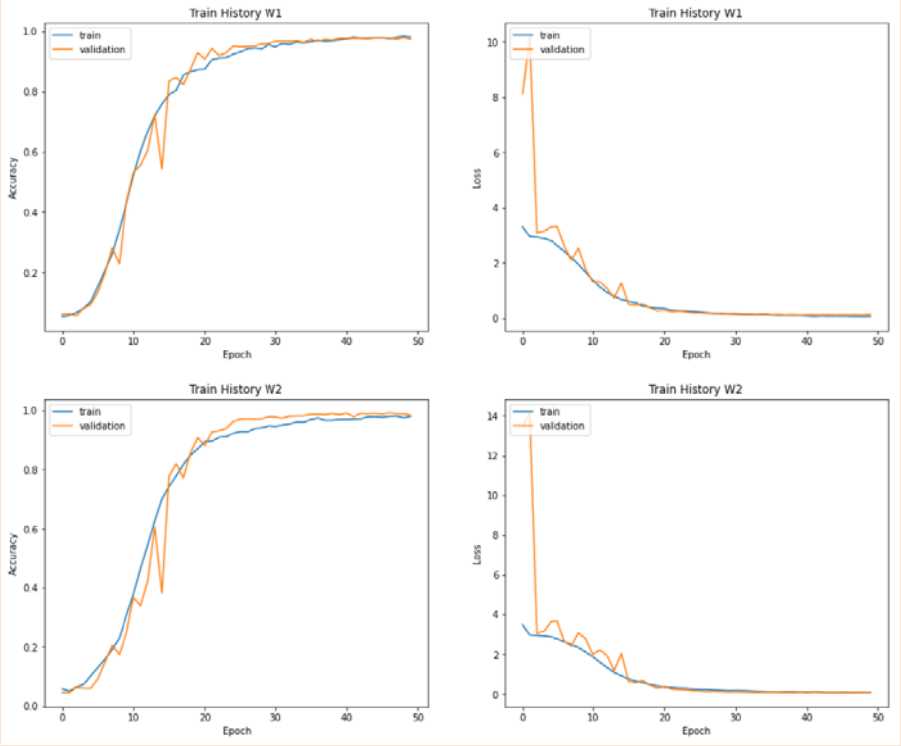
處理效果

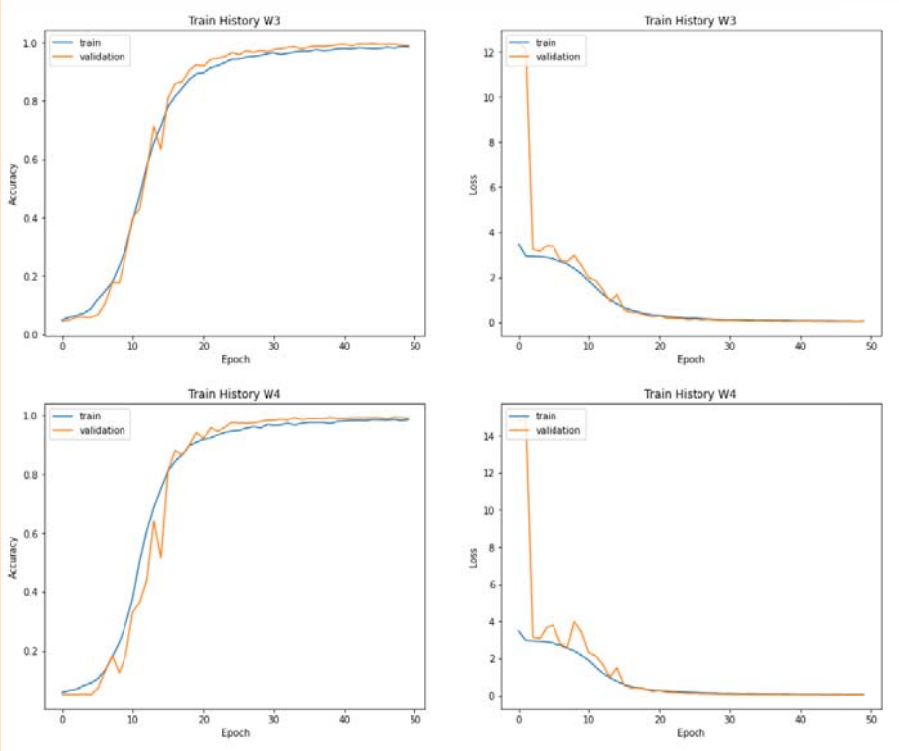
```
origin_X[0] shape: (49, 122, 3)
(5000, 50, 130, 3)
resize_x shape: (5000, 50, 130, 3)
resize_x[0] shape: (50, 130, 3)
```



訓練結果

```
Test W1 loss: 0.07709530081599951  Test W1 accuracy: 0.984
Test W2 loss: 0.07274479222670198  Test W2 accuracy: 0.986
Test W3 loss: 0.026128847817191853  Test W3 accuracy: 0.99
Test W4 loss: 0.06474165044724942  Test W4 accuracy: 0.984
```





測試結果

```
3000 picture total wrong = 219
image accuracy =  0.927
word accuracy =  0.9794166666666667
average per execute time: 11.694158 ms
total execute time = 44.779081 s
```

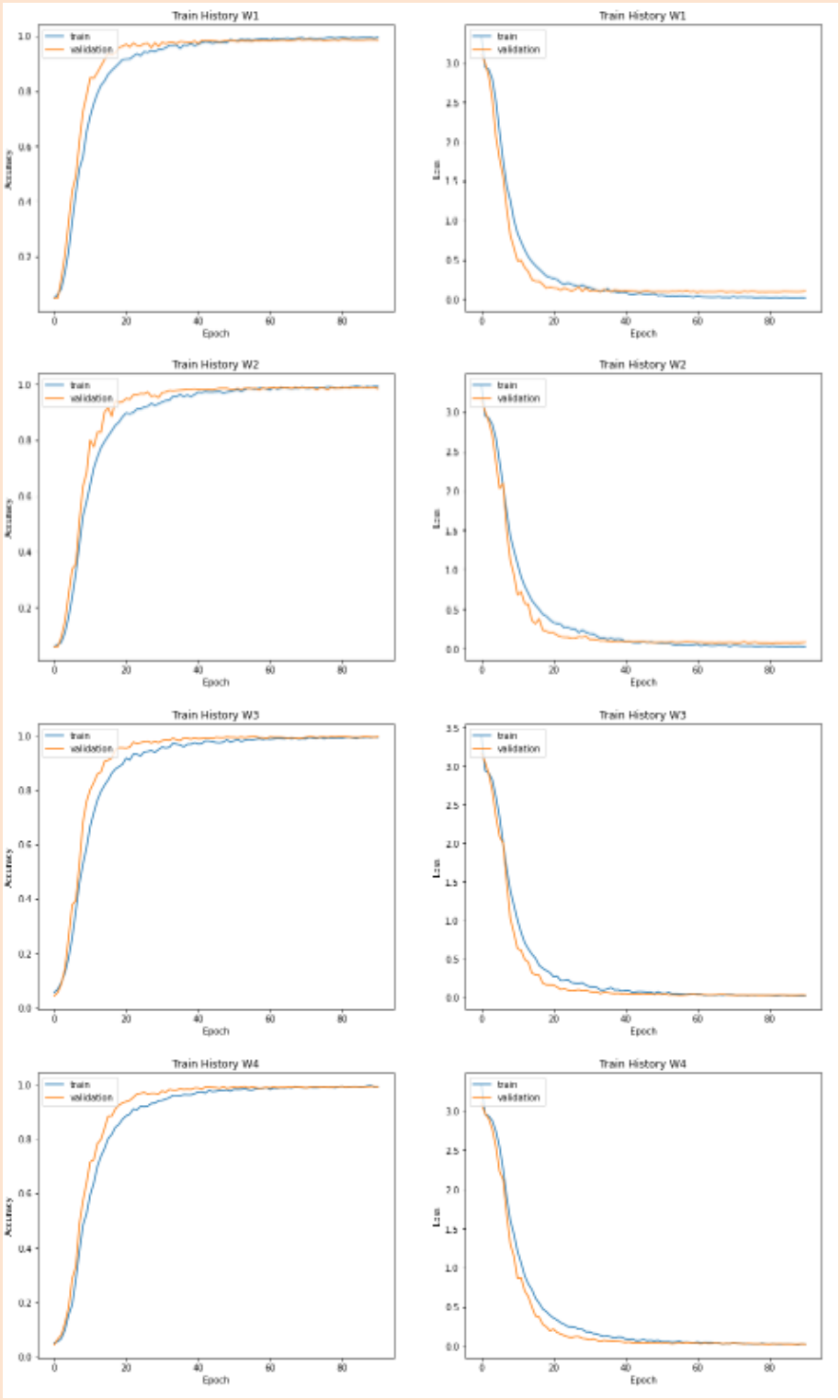
2.1.2. dn1

處理效果

```
origin_X[0] shape: (49, 122, 3)
dict_keys(['__header__', '__version__', '__globals__', 'denoise2_x'])
denoise2_x shape: (5000, 50, 130)
resize_x shape: (5000, 50, 130)
resize_x[0] shape: (50, 130)
```

訓練結果

```
Test W1 loss: 0.0695661993175745  Test W1 accuracy: 0.9839999995231629
Test W2 loss: 0.07656530291051604  Test W2 accuracy: 0.982
Test W3 loss: 0.006032318020472303  Test W3 accuracy: 0.998
Test W4 loss: 0.08874986103922129  Test W4 accuracy: 0.986
```



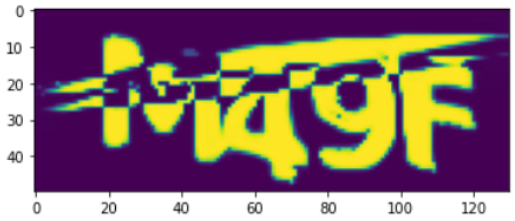
測試結果

```
3000 picture total wrong = 358
image accuracy =  0.8806666666666667
word accuracy =  0.9655
average per execute time: 173.600325 ms
total execute time = 547.334728 s
```

2.1.3. dn2

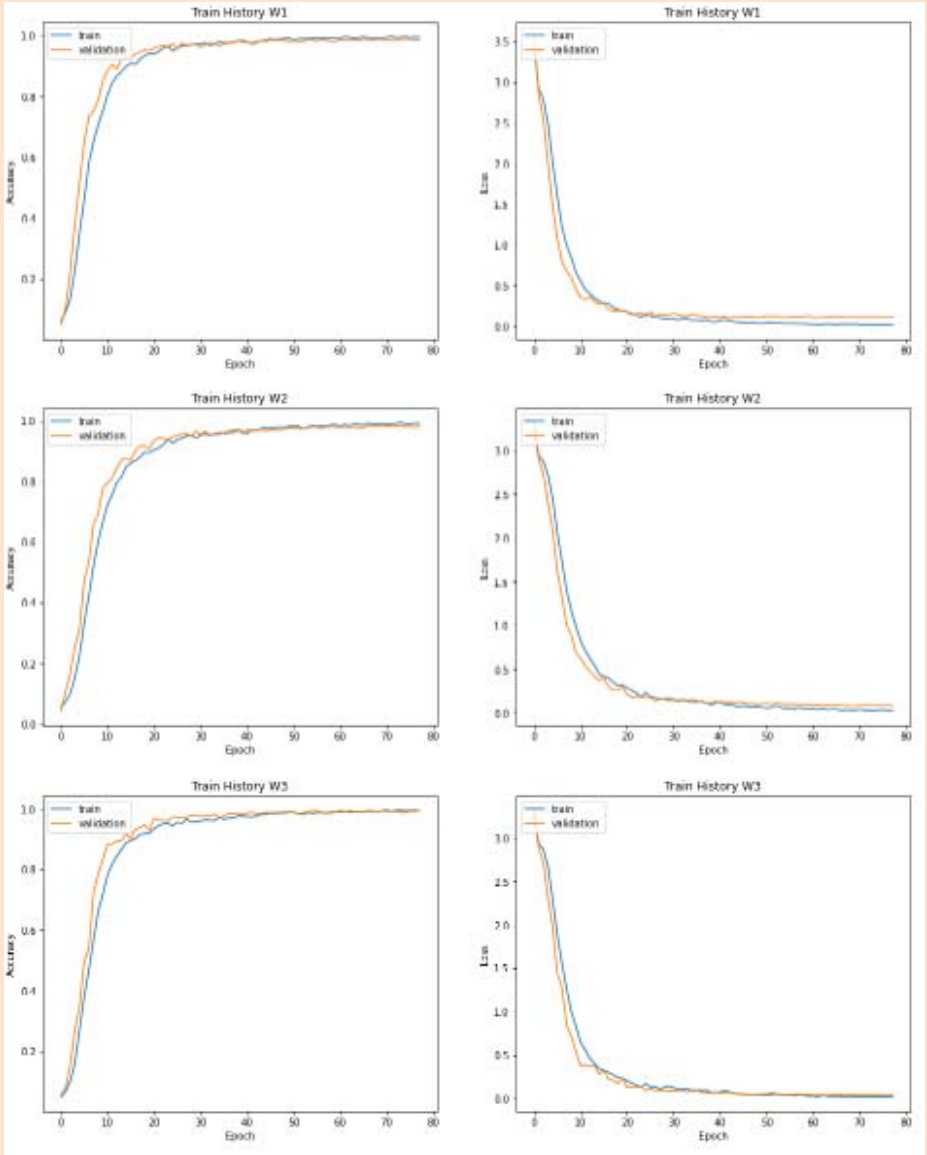
處理效果

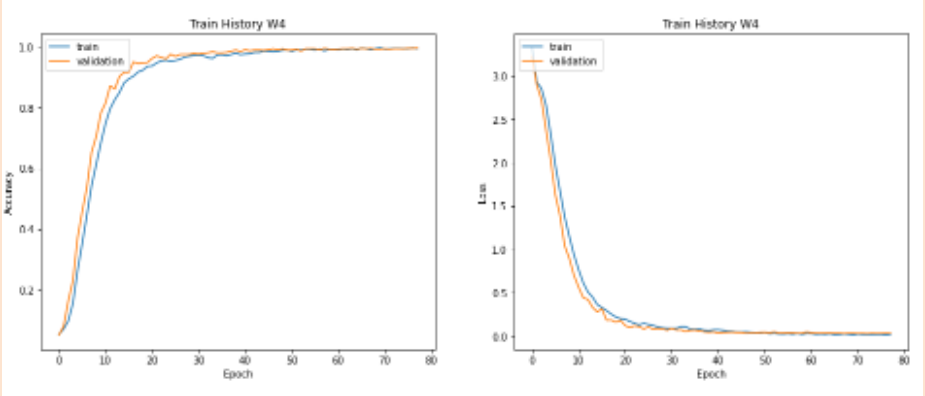
```
origin_X[0] shape: (49, 122, 3)
dict_keys(['__header__', '__version__', '__globals__', 'denoise_x'])
denoise_x shape: (5000, 50, 130)
resize_x shape: (5000, 50, 130)
resize_x[0] shape: (50, 130)
```



訓練結果

```
Test W1 loss: 0.0695661993175745  Test W1 accuracy: 0.9839999995231629
Test W2 loss: 0.07656530291051604  Test W2 accuracy: 0.982
Test W3 loss: 0.006032318020472303  Test W3 accuracy: 0.998
Test W4 loss: 0.08874986103922129  Test W4 accuracy: 0.986
```





測試結果

```
3000 picture total wrong = 234
image accuracy =  0.922
word accuracy =  0.9784166666666667
average per execute time: 84.030596 ms
total execute time = 339.646858 s
```

從結果可以看出，不管是 dn1 還是 dn2 的預處理方式，都會大幅的增加執行的時間，並且，在正確率上，nopre 反而有最佳的表現。因此，我認為在變是高鐵驗證碼時不做多餘的預處理是最好的選擇。

2.2. 網路架構

我使用的高鐵辨識碼 CNN 模型結構是由三層卷積層加上四層池化層，配合 BatchNormalization，最後再加上與全連接層組成。

最靠近輸入端的卷積層 Conv2D(64, (5, 5), strides = (1, 1), activation = 'relu', padding='same', name = 'conv11_W1')(X_input)，使用 64 個 5x5 的 filter 配合固定的權重做到影像處理，提取特徵。並且將 strides 設為(1, 1)，同時，除了增加 ReLU 激勵函式外，也使用 padding 補邊方法，針對輸入的維度做擴增，使得最後的輸出結果與原輸入大小相同。

接著 MaxPooling2D((2, 2), name='max_pool1_W1')(W1)使用最大值方法 maxpool-pooling，保留每個 2x2 資訊中的最大值，再將其餘資訊丟棄，將資訊壓縮簡化以增加模型的收斂及穩定性。

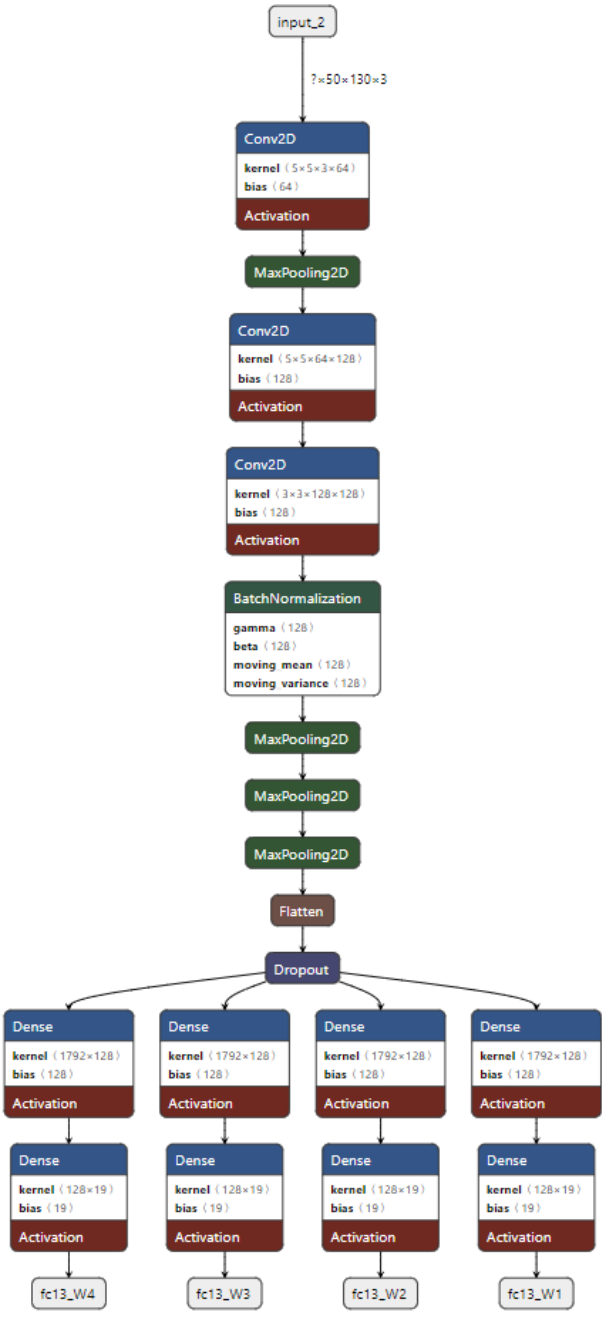
然後再經過兩次的卷積層 W1 = Conv2D(128, (5, 5), strides = (1, 1), activation = 'relu', padding='same', name = 'conv23_W1')(W1) 與 W1 = Conv2D(128, (3, 3), strides = (1, 1), activation = 'relu', name = 'conv25_W1')(W1)，也就是使用 128 層 5X5 的 filter 和 128 層 3x3 的 filter 做影像處理。兩個卷積層的 strides 都設為(1, 1)，激勵函式也都使用 ReLU。除此之外，5x5 的 filter 有做 padding 補邊。

接下來，我使用 Batch Normalization 正規化方法將深度學習模型優化。Batch Normalization 在進行學習時以 mini-batch 為單位，依照各個 mini-batch 進行正規化。為了增加神經網絡的穩定性，它透過減去 batch 的均值並除以 batch 的標準差來正規化前面激勵層的輸出。Batch Normalization 的使用時積除了改善收斂速

度慢的問題或是梯度爆炸無法訓練，在一般的情況下也可以使用，用以加快模型速度提高模型效能。

正規化後，接下來我再使用三次的池化層 W1 = MaxPooling2D((2, 2), name = 'max_pool2_W1')(W1)、W1 = MaxPooling2D((2, 2), name = 'max_pool3_W1')(W1)、W1 = MaxPooling2D((2, 2), name='max_pool4_W1')(W1)，使用 maxpool-pooling 最大值方法保留每個 2x2 資訊中的最大值，再將其餘資訊丟棄，重複執行將資訊壓縮簡化的步驟以增加模型的收斂及穩定性。

在連接到全連接層之前，我使用 Flatten()函數，將 3 維的資料轉換為 1 維。之後，W1、W2、W3、W4 的全連接層都是先使用 ReLU 再使用 Softmax 做計算。



圖一、網路架構

2.3. 超參數設置

這裡，我將每次迭代時從訓練集中提取，送入神經網路的資料數量設為256，提取其中的0.9作為訓練資料的比例，其中的0.1作為驗證的資料比例，進行總計200次的迭代訓練。也就是說，共有230份資料做為資料集，26份資料作為驗證集。

```
BATCH_SIZE = 256
NUM_EPOCHS = 200 #change to 100

#WEIGHTS_FINAL = 'model-cat-final2.h5'
```

```
train_rate=0.9 #change to 0.9
num_train_data=int(5000*train_rate)
```

2.4. 預處理方式

由於使用 nopre 的方式訓練正確率以及效率都最佳，所以這裡我使用 nopre 作為我的預處理。

```
origin_X[0] shape: (49, 122, 3)
(5000, 50, 130, 3)
resize_x shape: (5000, 50, 130, 3)
resize_x[0] shape: (50, 130, 3)
```

2.5. 準確度

2.5.1. 訓練

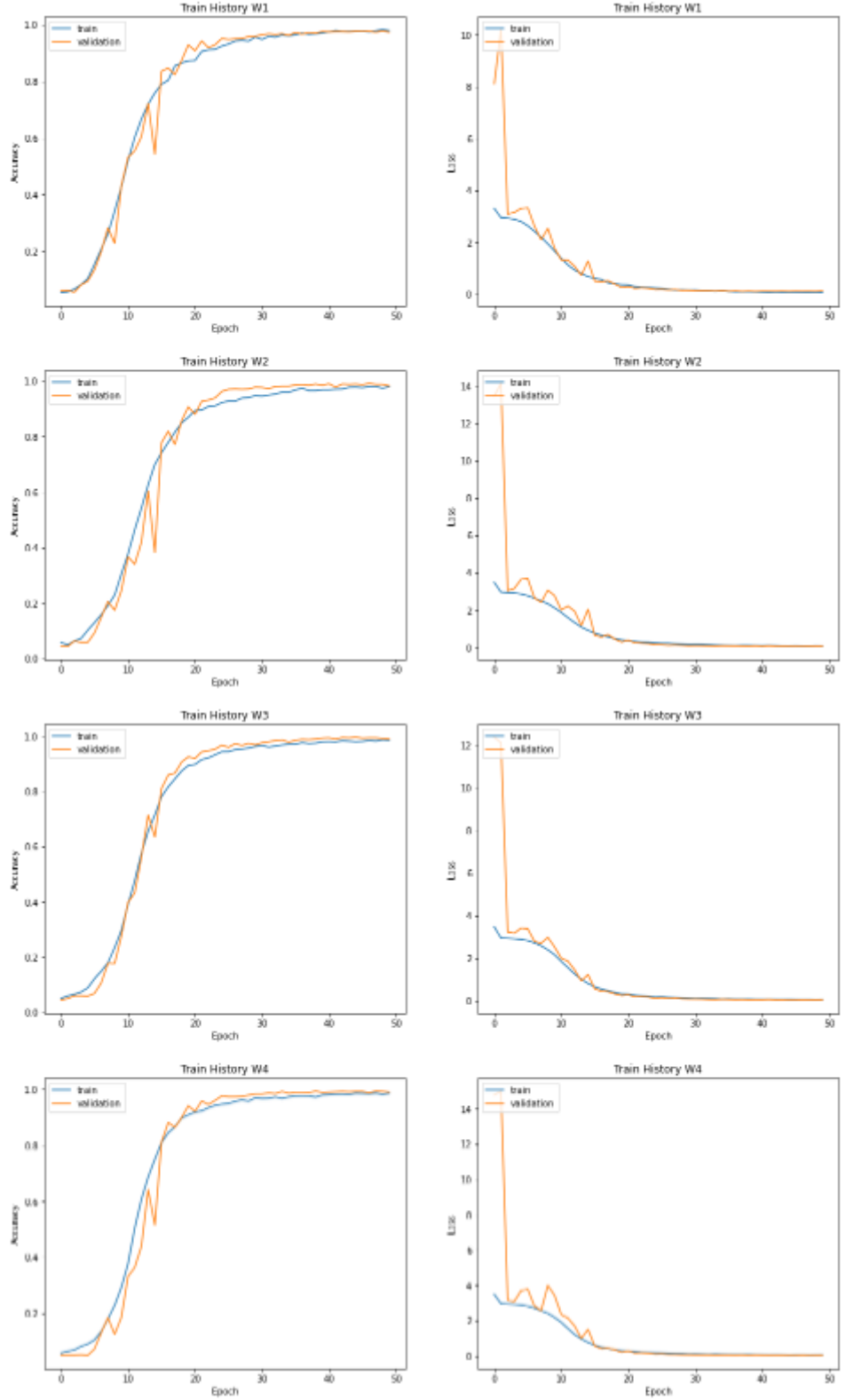
在訓練上，CNN 模型在第 97 輪迭代時便因為無法找到函數中更小的 minimum 位置，迭帶回數也超過設定的 patience 回述而 early stopped 了。

```
Epoch 00097: val_loss did not improve from 0.21286

Epoch 00097: ReduceLROnPlateau reducing learning rate to 8.235429777414538e-
05.
Epoch 00097: early stopping
```

訓練後的成效如下:

```
Test W1 loss: 0.07709530081599951  Test W1 accuracy: 0.984
Test W2 loss: 0.07274479222670198  Test W2 accuracy: 0.986
Test W3 loss: 0.026128847817191853  Test W3 accuracy: 0.99
Test W4 loss: 0.06474165044724942  Test W4 accuracy: 0.984
```



2.5.2. 測試

最終我的高鐵辨識碼模型在圖像及文字辨識上分別獲得 0.927 以及 0.979 的正確率，四個字的平均執行時間為 44.779 秒。

```
3000 picture total wrong = 219
image accuracy =  0.927
word accuracy =  0.9794166666666667
average per execute time: 11.694158 ms
total execute time = 44.779081 s
```

模型還有能再完善的空間，TensorFlowe 官方證實現今最好的模型正確率高達 0.92。如果想提升我的模型正確率，我想可以試著從增加層數或式調整參數下手，不過相對地執行效率的犧牲也可能因此造成。