

SimpleCNN

2018 年 8 月 3 日



1 圖像的深度學習 - CNN

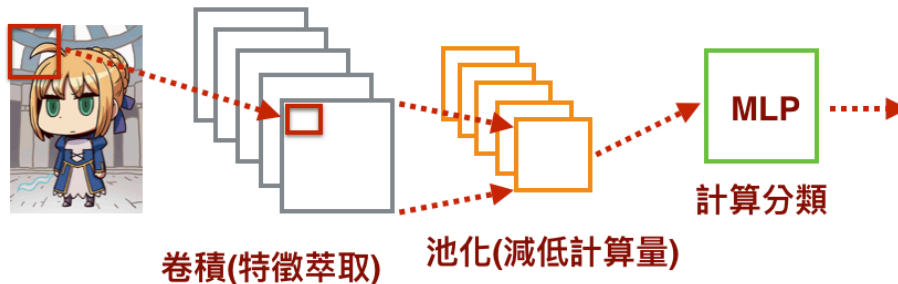
1.1 介紹

儘管之前已經使用了 MLP 來辨識簡單的手寫圖像，但是如果來到複雜的圖像，我們發現兩個問題

1. 把每一個像素平鋪開以後，在全連接下，要訓練的參數太多，要花費超級多時間訓練
2. 圖片素材取得不易，很難建立大量的資料

所以我們針對這兩個問題進行改進，發展成另外一種針對圖像的深度學習演算法: 卷積神經網路 (Convolutional Neural Network)

1.2 CNN 理論基礎



CNN 跟之前唯一不同的就是加入卷積和池化來解決上面的兩個問題

1.2.1 卷積層

我們先來解決第二個問題: 圖片素材取得不易, 難以建立大量的資料

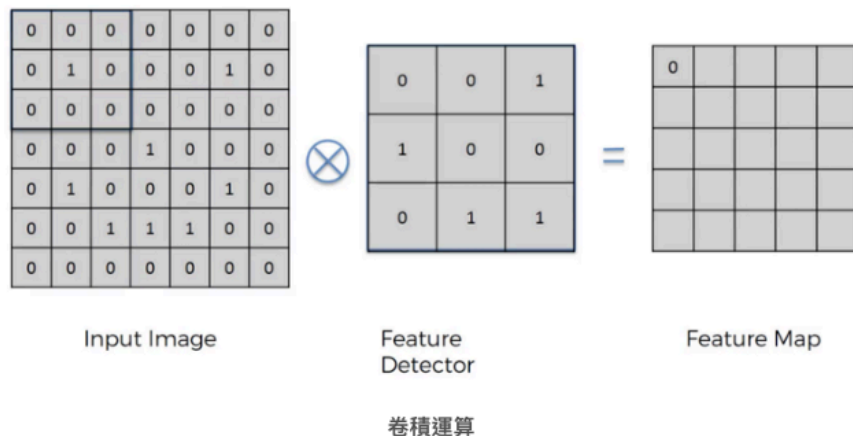
事實上, 這個問題起因應該是我們對於整個圖片的使用率太低

什麼意思呢? 一個圖片事實上應該有很多可以看的, 譬如: 顏色, 外框之類的等等

但是在我們之前的MLP, 我們並未做出這些特徵的抓取, 而是直接暴力的把所有像素進去

這裡為了讓圖片的應用效果最大化, 我們利用 **filter** 的概念, **filter** 的意思就是把原始圖片經過某些流程, 只留下我們感興趣的特徵 (ex. 邊角, 外框...之類)

有了這些特徵圖, 我們就可以改用這些特徵圖來做我們的分類, 一個具體的想像就是把圖片擴展成跟我們之前不管是鳶尾花還是鐵達尼號的分類一樣, 擁有許多許多的特徵, 利用這些特徵來分類



我們會選一個特徵圖 (filter)(ex. 可能是一個人臉特徵...之類) 不斷的一個一個像素平移, 檢查那個區域有沒有我們想要的特徵

上面的檢測 = 每一個像素相乘加起來 = $0 \times 0 + 0 \times 0 + 0 \times 1 + 0 \times 1 + 1 \times 0 + 0 \times 0 + 0 \times 0 + 0 \times 1 + 0 \times 1 = 0$
 = 沒有相關特徵在這區域

透過很多的特徵圖，我們就可以得到很多不同的特徵！最大化一張圖訓練的效益

1.2.2 池化層

池化是為了解決第一個問題，計算量太大，那怎麼讓計算量減小呢？

就是圖片的壓縮和縮小，簡單來說，假設我有一個 2×2 的像素區域，假設是

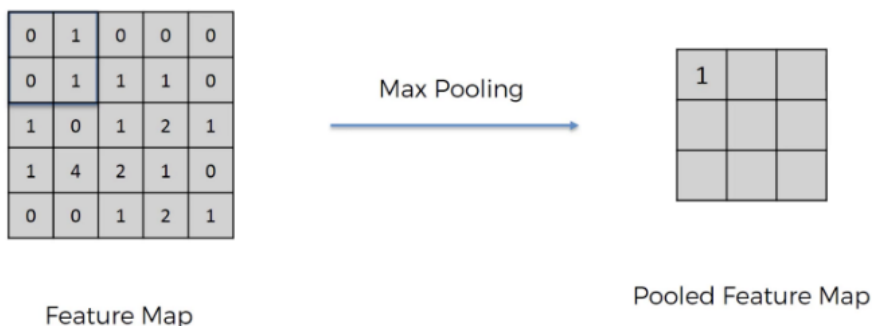
	0	1
0	4	1
1	1	0

我們如果把他壓縮成 1×1 像素，取最大值 (4)，會不會失真太多呢？答案是不會的，只要區域不要太大，理論上是不會失真太多的

池化通常有兩種：

1. Max Pooling(最大值池化): 取一個區域的最大值
2. Mean Pooling(平均值池化): 取一個區域的平均值

我們在這裡用最簡單的最大值池化就好了



1.3 CNN 架構

現在的問題是，我們究竟要幾層的卷積和池化呢？池化的 Size 應該多少呢？Filter 的數量應該是多少個呢？

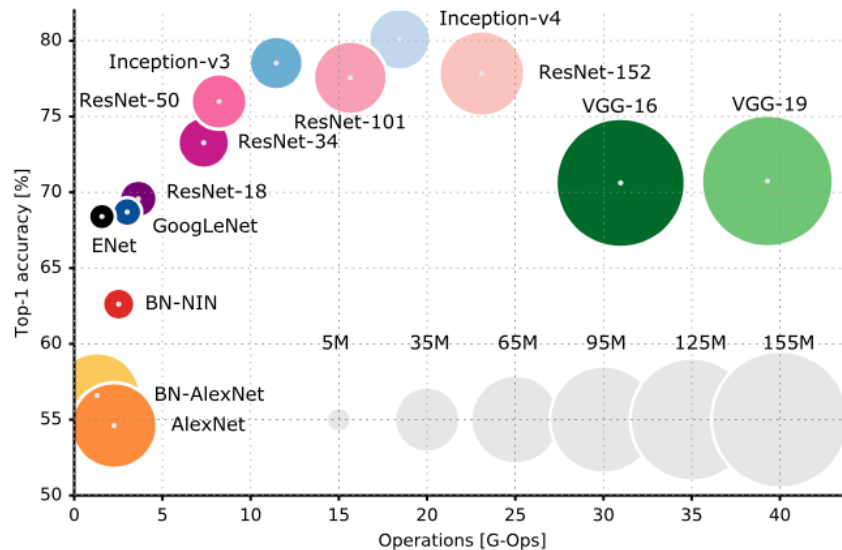
事實上在這裡完全沒有式子可以告訴你最佳答案

我們也很難自己試出最佳答案到底是多少，所以我們在圖片分類的時候通常會做一件事：參考別人的架構！

Imagenet 是 Hinton 和他的學生建立起來的一個龐大圖像資料庫，許多大公司都會在上面驗證他們的演算法好壞

每年還會舉辦比賽，參加的是全球的科技巨頭

在這些年的比賽中，出現了许多非凡的演算法，也就是下面的這些



橫軸的 Ops 你可以想成計算量，縱軸的 Top-1 Accuracy 是指說這模型只猜一個類別，正確率是多少

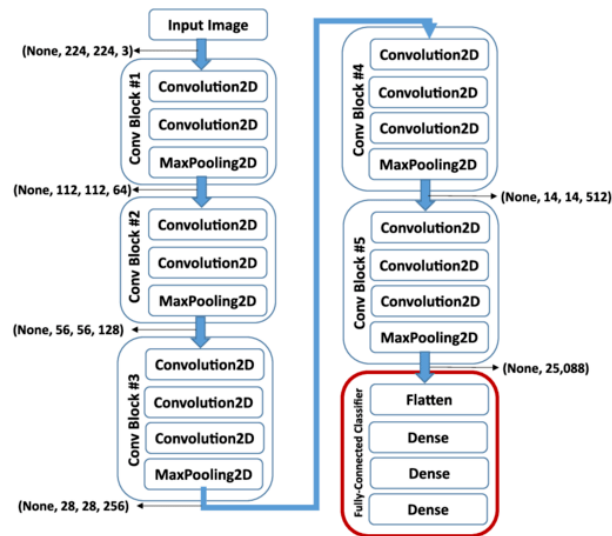
偶爾我們會看到 Top-5 Accuracy，指的是模型可以猜五個他覺得最可能的猜測，只要有一個對，就算對

1.3.1 VGG-16

VGG-16 是第一個展露頭角的 CNN 網路，而且是非常傳統的 CNN 網路，只用卷積以及池化和全連結層構成了整個架構

今天我們就參考 VGG-16 網路來建立我們的 CNN 網路

這裡我們先來一個簡化的圖



我們再來討論一下卷積和池化

首先，我們進來的圖片是 (224, 224, 3) 三個維度 -> 224 像素 (寬) x 224 像素 (高) x 3 (RGB 三通道)

再論卷積 你把 RGB 當成原圖的三個特徵，卷積其實就是特徵的萃取和再製，所以卷積只會去改變第三個維度 (在這裡就是 3)

而不會改動到寬和高這兩個維度，VGG-16 的卷積基本就是每一次的卷積把特徵數 * 2
從 3, 128, 256 到最後的 512

再論池化 池化剛好相反，他是去取像素的最大值 (最大池化)，所以反而一定不會改到第三個維度，只會更改到前面的寬和高

這裡從 224, 112, 56, 28 到最後的 14

全連接 做完特徵的萃取和計算量的減低，一樣攤開經由我們的 MLP 開始判斷

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool1 (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool1 (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool1 (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool1 (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool1 (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102764544
fc2 (Dense)	(None, 4096)	16781312
predictions (Dense)	(None, 1000)	4097000
dense_1 (Dense)	(None, 16)	16016
Total params: 138,373,560		
Trainable params: 138,373,560		
Non-trainable params: 0		

完整參數

1.4 簡化版 VGG-16

在這裡我們不可能從頭開始訓練 VGG-16 的所有參數，所以我們先取前面的幾層來訓練得到一個初步的印象

在實務上，有時候我們為了時間考慮，會拿取已經訓練好的模型，再 **fine-tune** 其中幾層的參數

理論上，有了基礎的分辨，稍加訓練，雖然不及從頭訓練來得好，但是已經有還不錯的分辨能力了

1.5 Step1. 資料預處理

這裡我們選用 cifar10 圖像資料庫來，cifar10 提供共 60000 筆較為小的圖片，並且總共有 10 種分類

你甚至可以把它當成不清晰版的 imagenet 了

```
In [1]: from keras.datasets import cifar10
```

```
# MAC 一定要加入此行, 才不會把對方伺服器的 SSL 證書視為無效
import ssl
ssl._create_default_https_context = ssl._create_unverified_context

import matplotlib.pyplot as plt
%matplotlib inline

(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

Using TensorFlow backend.

把測試資料和訓練資料印給你看, 訓練資料是 50000 萬筆的 32 x 32 的 RGB 圖片

```
In [2]: print(x_train.shape)
        print(x_test.shape)
```

```
(50000, 32, 32, 3)
```

```
(10000, 32, 32, 3)
```

十種類別分別如下, 我先創造出來, 等等可以把圖片的類別印給你看

```
In [3]: label = {0:"飛機", 1:"車", 2:"鳥", 3:"貓", 4:"鹿",
                 5:"狗", 6:"青蛙", 7:"馬", 8:"船", 9:"卡車"}
```

利用 input 你可以決定你要秀出哪張照片

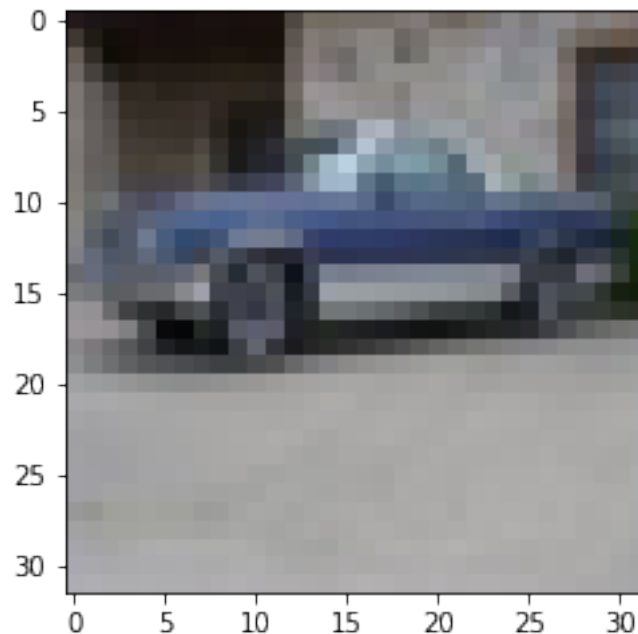
```
In [4]: a = int(input("請輸入你想可視化的圖片 [0-49999]:"))
        print("你想可視化的圖片號碼是", a)
        print("圖片答案是", label[y_train[a][0]])
        plt.imshow(x_train[a])
```

請輸入你想可視化的圖片 [0-49999]:10000

你想可視化的圖片號碼是 10000

圖片答案是 車

```
Out[4]: <matplotlib.image.AxesImage at 0x124a0c630>
```



在上一張我們提過，使用深度神經網路的一個好習慣就是標準化，這樣對於我們的 w 的更新會比較好！

另外一個你需要做的是，如果判斷是多個類別，請把它轉成 One-hot Encoding 形式

```
In [5]: from keras.utils import np_utils
        x_train_shaped = x_train.astype("float32") / 255
        x_test_shaped = x_test.astype("float32") / 255
        y_train_cat = np_utils.to_categorical(y_train)
        y_test_cat = np_utils.to_categorical(y_test)
```

1.6 Step2. 建立模型

這裡我們一樣使用 Sequential 模型

但我們開始用三種不同的 Layer

1.6.1 Conv2D (卷積層)

我們使用 Conv2D 層來卷積，要設定的最重要參數是 `filters`，我們這裡參照了上面的 VGG16 並做了一點小小的簡化，我們只做了兩次的卷積，並且第一次只擴充到 32 通道，`kernel_size` 則是你的過濾器的寬和高，我們設值成 (3, 3)，也就是跟上面的介紹一樣的過濾器寬高

啟動函數跟 MLP 一樣，只要在中間層我們就會選擇 `relu`，避免梯度的消失，這裡你會發現一件事情

原來我們的 filter 也是透過訓練 w 決定的!!!

這正是深度神經網路和之前的不一樣，我們的過濾器不再是提早決定好了，而是透過深度神經網路的訓練來決定

Padding 的設置是為了讓卷積後的寬高保持不變 (最後一次的卷積會只遇到 1 寬度，我們會幫他 Padding 成 2 寬度)

1.6.2 MaxPooling2D (最大池化層)

我們選擇池化窗是 (2, 2) 的話也就意味著 2 個寬度取一個最大值，2 個高度取一個最大值就相當於本來的寬高 (w, h) -> 池化後 -> (w/2, h/2)

1.6.3 Dropout (Dropout 層)

在訓練中，我們很怕遇到過擬合的情況，那在我們的神經網路裡，怎麼防止模型對於資料過擬合呢?

Hinton 提出了一個很簡單但卻非常有效的方法，就是每一次在訓練的時候，不要用全部的神經元來訓練

而是隨機的斷開一些神經元 (斷開不代表消失，下次斷開的不一定是他)，也就是我們每次訓練的模型稍微都有所不同 (很像隨機森林裡的每棵樹對吧)

這樣就會自然的不這麼擬合了!

依照前人的經驗，通常在這裡 drop 掉 25%(0.25)~50%(0.5) 個神經元會是一個不錯的選擇

```
In [1]: from keras.models import Sequential
        from keras.layers import Dense, Dropout, Activation, Flatten
        from keras.layers import Conv2D, MaxPooling2D, ZeroPadding2D

        model = Sequential()

        # 第一次卷積和第一次池化
        model.add(Conv2D(filters=32,
                          kernel_size=(3, 3),
                          input_shape=(32, 32, 3),
                          activation='relu',
                          padding='same'))
        model.add(MaxPooling2D(pool_size=(2, 2)))

        # 斷開 25% 的連接
        # 並且加入第二次卷積和第二次池化
```

```

model.add(Dropout(0.25))
model.add(Conv2D(filters=64,
                  kernel_size=(3, 3),
                  activation='relu',
                  padding='same'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# 把你處理過的東西攤開成為一維
model.add(Flatten())
model.add(Dropout(rate=0.25))

# 全連接層
model.add(Dense(128, activation='relu'))
model.add(Dropout(rate=0.25))

model.add(Dense(10, activation='softmax'))
model.summary()

```

Using TensorFlow backend.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 32, 32, 32)	896
max_pooling2d_1 (MaxPooling2D)	(None, 16, 16, 32)	0
dropout_1 (Dropout)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(None, 8, 8, 64)	0
flatten_1 (Flatten)	(None, 4096)	0
dropout_2 (Dropout)	(None, 4096)	0

```

dense_1 (Dense)                (None, 128)                524416
-----
dropout_3 (Dropout)            (None, 128)                0
-----
dense_2 (Dense)                (None, 10)                 1290
=====
Total params: 545,098
Trainable params: 545,098
Non-trainable params: 0
-----

```

1.6.4 Param 個數

我們一起看看上方 summary 的參數個數

1. 卷積層 1 的 896: 32 個濾波器，每一個 $3(\text{寬度}) \times 3(\text{高度}) \times 3(\text{input 通道 RGB}) + 1(\text{bias}) = 28$ 個參數， $32 \times 28 = 896$ 個參數
2. 卷積層 2 的 18496: 64 個濾波器，每一個 $3(\text{寬度}) \times 3(\text{高度}) \times 32(\text{input 通道數}) + 1(\text{bias}) = 289$ 個參數， $64 \times 289 = 18496$ 個參數
3. 全連接層的 524416: $4096 \times 128 + 128(\text{bias}) = 524416$ 個參數

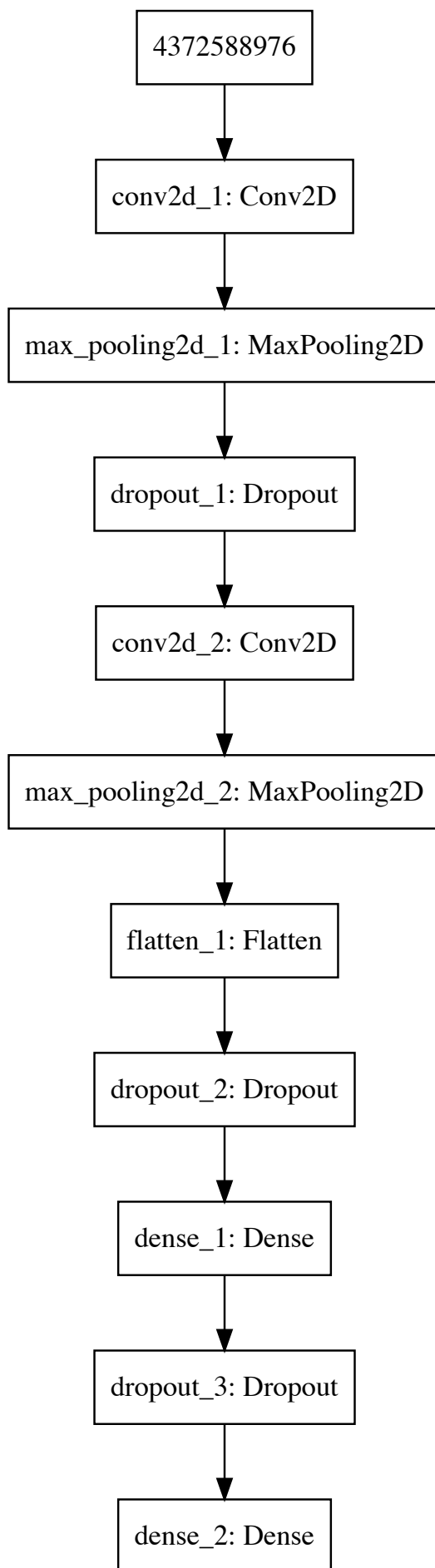
```

In [7]: from IPython.display import SVG
        from keras.utils.vis_utils import model_to_dot

        SVG(model_to_dot(model).create(prog='dot', format='svg'))

```

Out[7]:



1.6.5 開始訓練

你可以看到我們 10 個 epoch 大概花了 10 分鐘訓練，最後 val_loss 差不多已經趨於平穩，所以其實 10 個 epoch 已經差不多了

```
In [8]: model.compile(loss="categorical_crossentropy",
                      optimizer = "adam",
                      metrics = ['accuracy'])
train_history = model.fit(x = x_train_shaped, y = y_train_cat,
                          validation_split = 0.1,
                          epochs = 10,
                          batch_size = 128,
                          verbose = 2)
```

Train on 45000 samples, validate on 5000 samples

Epoch 1/10

- 57s - loss: 1.7013 - acc: 0.3832 - val_loss: 1.3603 - val_acc: 0.5134

Epoch 2/10

- 53s - loss: 1.3428 - acc: 0.5182 - val_loss: 1.1724 - val_acc: 0.5946

Epoch 3/10

- 54s - loss: 1.2093 - acc: 0.5706 - val_loss: 1.1136 - val_acc: 0.6146

Epoch 4/10

- 52s - loss: 1.1301 - acc: 0.5984 - val_loss: 1.0042 - val_acc: 0.6474

Epoch 5/10

- 58s - loss: 1.0636 - acc: 0.6219 - val_loss: 0.9457 - val_acc: 0.6702

Epoch 6/10

- 55s - loss: 1.0149 - acc: 0.6396 - val_loss: 0.9024 - val_acc: 0.6942

Epoch 7/10

- 53s - loss: 0.9735 - acc: 0.6567 - val_loss: 0.8784 - val_acc: 0.7012

Epoch 8/10

- 52s - loss: 0.9385 - acc: 0.6712 - val_loss: 0.8536 - val_acc: 0.7104

Epoch 9/10

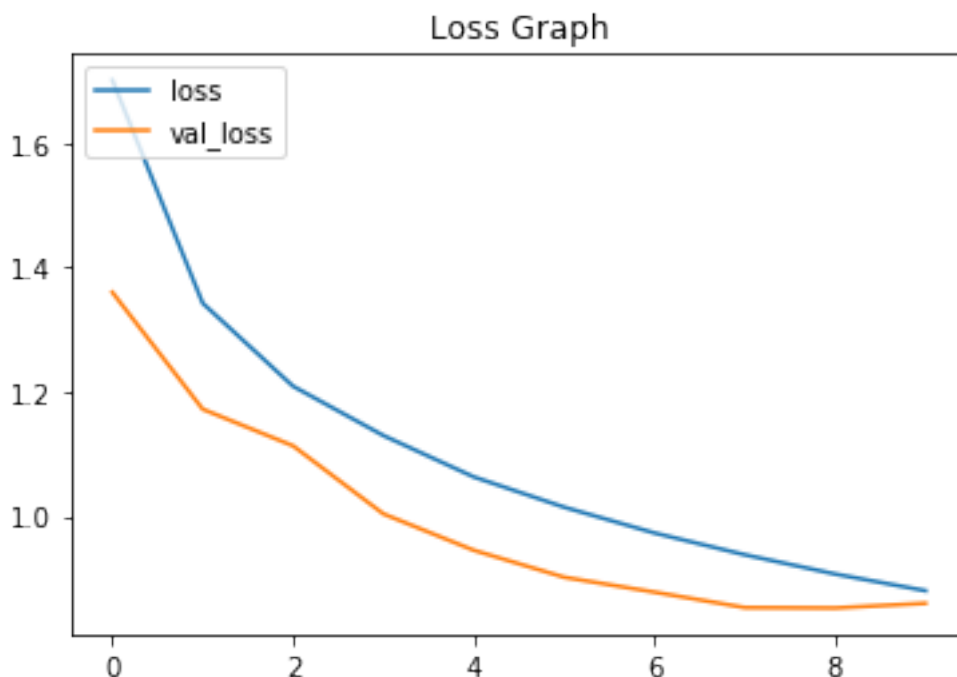
- 53s - loss: 0.9075 - acc: 0.6806 - val_loss: 0.8530 - val_acc: 0.7042

Epoch 10/10

- 68s - loss: 0.8807 - acc: 0.6892 - val_loss: 0.8605 - val_acc: 0.7118

```
In [9]: plt.plot(train_history.history["loss"])
        plt.plot(train_history.history["val_loss"])
        plt.title("Loss Graph")
        plt.legend(['loss', 'val_loss'], loc="upper left")
```

```
Out[9]: <matplotlib.legend.Legend at 0x150e054a8>
```



List 的第二個是我們的正確率，你可以看到正確率大概是 70% 左右，以這麼簡單的卷積網路來說已經不錯了

```
In [10]: model.evaluate(x_test_shaped, y_test_cat)

10000/10000 [=====] - 5s 520us/step
```

```
Out[10]: [0.8914391684532166, 0.6947]
```

1.7 Step3. 儲存模型

我們可以藉由 `save` 直接儲存成 `hdf5` 類型的檔案，`hdf5` 是一種資料庫形式，他會以階層式 (就像我們電腦的資料夾) 來儲存你的資料，你的一個一個儲存的資料就會像電腦裡的檔案位於不同的資料夾，之後你就可以使用 `load` 把你當初訓練好的所有參數載入回來

```
In [11]: model.save('cnn1.h5')
```

我們可以藉由軟體 `hdfview` 來開啟 `hdf5` 檔案格式

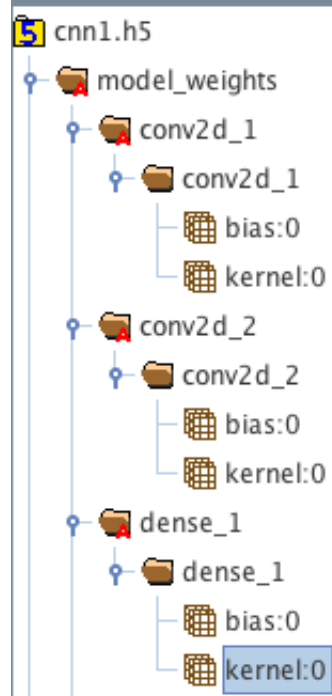
<https://support.hdfgroup.org/products/java/release/download.html>

請在這裡下載




以下展示一下我開啟 `cnn1.h5` 檔案的畫面

我們可以看到每一層都被儲存下來了



點進去你可以看到確實訓練出來的參數都被儲存了

kernel:0 at /model_weights/dense_1/dense_1/ [cnn1.h5 in /Users/Elwi

Table 

	0	1	2	3	4
4030	0.055459...	-0.07749...	-0.05894...	-0.02302...	-0.05851...
4031	-0.00340...	0.055477...	-0.00523...	-0.04247...	-0.01333...
4032	0.016113...	0.021479...	0.026292...	-0.00316...	0.014386...
4033	-0.01049...	-0.01994...	0.004785...	0.014793...	0.008700...
4034	0.020475...	-0.03334...	0.004371...	0.002015...	0.029571...
4035	0.030232...	-0.02547...	-0.00751...	0.024257...	0.009110...
4036	-0.00291...	0.060014...	0.004878...	-1.24107...	-0.03728...
4037	0.034150...	0.020552...	-0.00290...	-0.01285...	-0.02622...
4038	0.007243...	-0.06513...	-0.03890...	0.004310...	0.009421...
4039	-0.03771...	0.073945...	0.035994...	-0.01278...	-0.00329...
4040	-0.02035...	0.009032...	-0.00185...	0.002331...	0.015627...
4041	-0.00887...	-0.14406...	-0.03788...	-0.03566...	0.007929...
4042	0.036383...	-0.21050...	0.008463...	0.019900...	0.007325...
4043	-0.01477...	0.042035...	-0.00531...	0.017164...	-0.03203...
4044	-0.00780...	-0.02041...	0.007992...	1.789570...	-0.03511...
4045	-0.00716...	-0.02020...	0.026667...	-0.05026...	-0.01205...
4046	-0.00755...	-0.10623...	-0.01257...	0.020442...	-0.00273...
4047	0.015341...	-0.08324...	-0.01765...	-0.00374...	-0.00394...
4048	-0.00733...	0.068243...	-0.02780...	0.010626...	0.027988...
4049	-0.01486...	0.056750...	-0.01469...	-0.02104...	-0.02754...

1.8 結語

我們已經完成了第一個簡單的 CNN，但要說深度，是完全完全不夠的，我們下一章繼續增加深度看一下我們的模型的轉變