MLP

2018 年 7 月 18 日

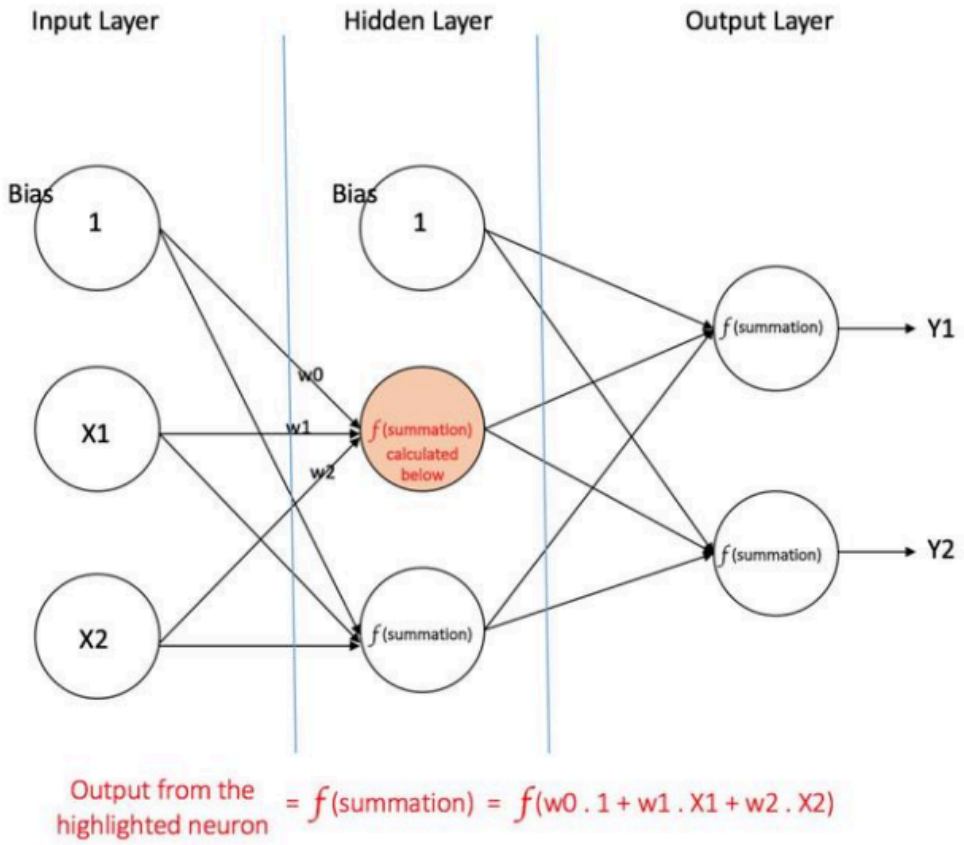
1　深度學習基礎 - MLP

1.1　介紹

上一次我們介紹到了 SVM，利用核函數來模擬非線性的分類，當時還有另外一派的學者說我們何不讓神經元變成一個多層的結構，這樣整個模型會很類似我們人類的神經系統，而且加上一層層，自然就不會是一條直線到底，非線性的問題自然就不攻而破。

1.2 多層感知器 (Multi-layer Perceptron)

1.2.1 第一步: 緣起



這圖就是當初我們設想的多層神經元，每一層的輸入都會乘上一個權重 (w)，相加以後傳入下一層神經元，不過這裡要特別注意到的是，傳進去前我們還要經過一次的激勵函數 (Activation Function)，也就是我們之前提到的 Logistic Function，簡單來說，每次往下一層的傳遞就是決定下一個神經元是否應該被 ‘‘激活’’。

這個模型看似美好，但是第一個問題要被解決，就是這麼多的係數 (w) 應該怎麼被決定呢?

1.2.2 第二步: 梯度下降

我們先回到一個神經元來說一下梯度的概念

斜率各位應該沒有問題，我們常說一句話，斜率 = 0 的地方就代表我們的極大值或者極小值，那如果我換一句講法，把函數當成一個 2D 山谷，你的斜率就是你傾向走的方向，走到底點或者高點的時候自然就傾向留在那了，所以斜率 = 0，傾向不走。

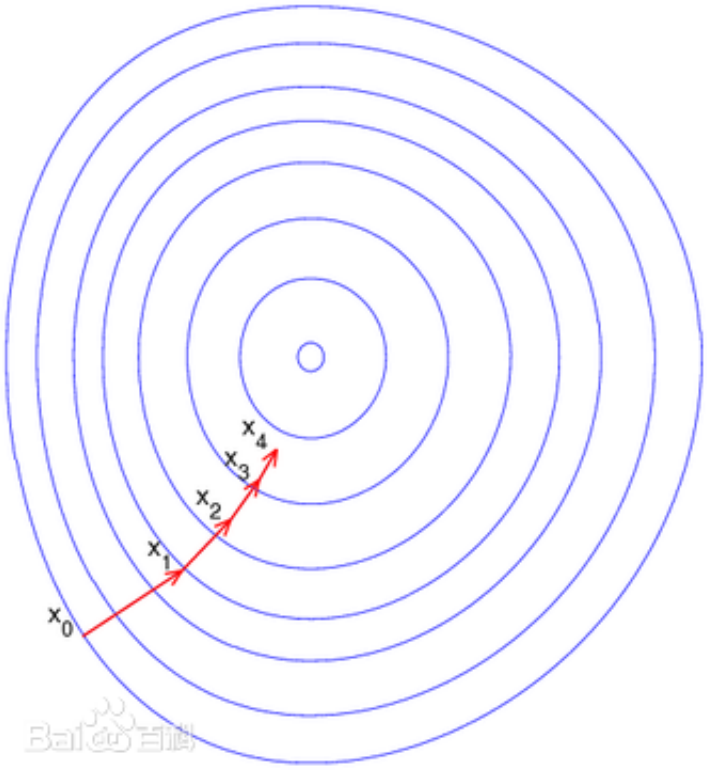
那梯度只是把斜率擴充到高維度的概念，最大的差別就是!! 我們的梯度是一個向量，是有方向性的!! 再換句話說，我們現在的維度空間提高了，所以你在走的時候要說出你每個維度要走的量。

假設我們有 f(x, y, z) -> 那我們走的傾向就是 (x 方向, y 方向, z 方向) = (對 x 偏微分, 對 y 偏微

分, 對 z 偏微分)

$$\nabla\varphi = \left(\frac{\partial\varphi}{\partial x}, \frac{\partial\varphi}{\partial y}, \frac{\partial\varphi}{\partial z} \right)$$

回到一層的神經元，如何求出一層神經元的 w(權重) 呢？很簡單，假設把我們所有 w 的選擇和 Loss(跟目標的差距) 做出一個圖，我們開始擁有一個等高線圖了



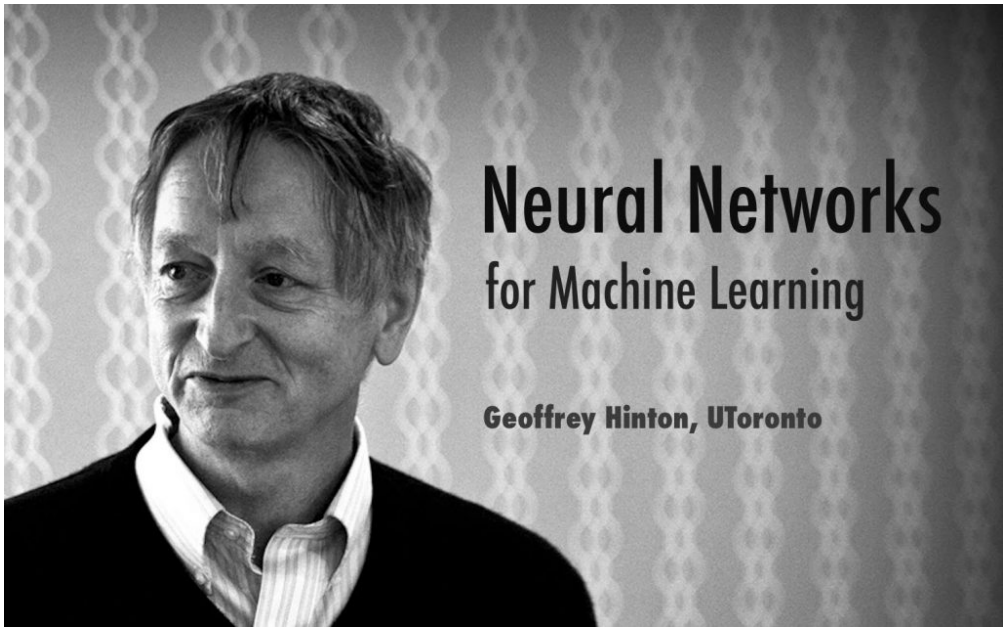
假設比較裡面的圈圈是我們的損失最小，我們要往它前進，該如何做呢？

很簡單，隨便選個 w，這時候你的損失一定不是最小，但是我們可以沿著 “負梯度” 的方向前進，我們就可以走到整個等高線的最低點，這個就叫做我們的踢度下降！

1.2.3 第三步: 反向傳播

剛剛我們理解了一個神經元如何決定 w，那怎麼決定多層的 w 呢？

這裡就不得不提到我們深度學習的鼻祖 Hinton 教授了



Hinton 教授在二十年前根據梯度的概念提出了誤差的反向傳播 (Backpropagation) 造成轟動

我們先聊聊簡單微分，假設今天我們有 f(x) = 4x ^ 2，微分應該告訴你微分後 f′(x) = 8x

但是如果我換一個方法問你呢?

我們的 x 先經過 g 函數 g(x) = 2x 再把整個 g 算出的結果丟進 f f(g) = g ^ 2

這樣我如果 f 函數對 x 的微分是多少呢?

微分原理的鏈式法則告訴你 ***f 對 x 微分 = f 對 g 微分 g 對 x 微分*** *

我們先算 g 對 x 微分: 2 再來我們算 f 對 g 微分 = 2g

所以 f 對 x 微分 = 4g 囉 g = 2x，一樣 = 8x

為什麼我們會說到這呢? 因為我們的多層神經元不就是這樣寫的嗎? W: 權重加總函數 L:Logistic

result = L2(W2(L1(W1(數據特徵))))

利用上面的鏈式法則，你可以輕鬆算出你想要更新權重層的梯度

W1 層更新: result 對 W1 微分

W2 層更新: result 對 W2 微分

Hinton 教授在 20 年前提出誤差反向傳播後到今天我們都還在使用這方法來更新我們的權重，但深度學習在 20 年前卻式微了，為什麼呢?

因為有一個嚴重的問題沒有被解決

你想想看上面的式子，我們總有一天會算到 Logistic 函數的微分，麻煩的是邏輯函數的微分你可以看到上面的兩條基本上微分 = 0

中間的曲線微分後大概介於 0 ~ 1 之間

一層的神經元的時候，這個微分乘上去還沒有什麼

但你能想像多層的時候，我們就必須把這 0 ~ 1 之間的值乘上多次!

大概就是你每層的更新幅度到下一層都打個八折的感覺！總有一天你的更新幅度會趨近於 0
這就是我們所說的踢度消失，也就是最前面幾層的權重更新根本沒在動!!!
也因此當初大家都認為就算多層，我們也只能做到兩三層，兩三層我們還不如使用之前所說的ＳＶＭ，還可以達到更好的精確度!

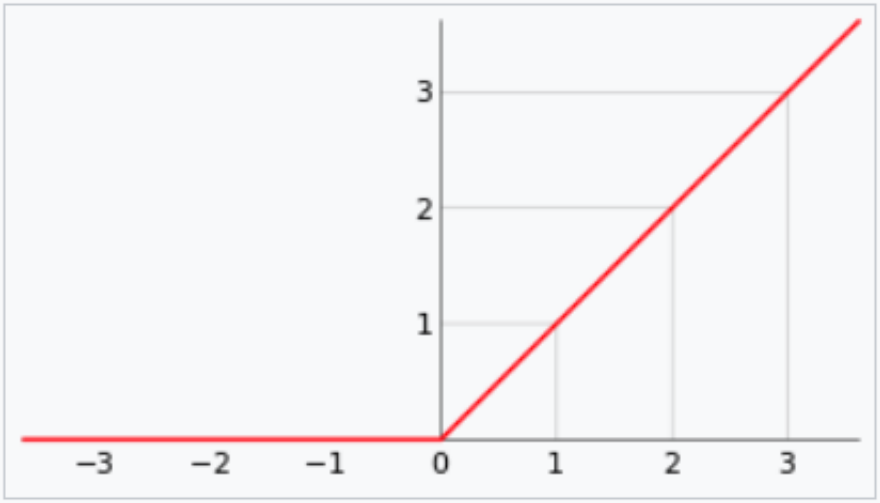
1.2.4 第四步: 梯度消失的解決

Hinton 教授在經過十數年的深度學習黑暗期，始終沒有放棄他的多層神經網路的想法，終於，他提出了深度信念網路來解決問題。終於讓深度學習再度露出一絲曙光，站在他的肩膀上，我們再度開始探討如何解決梯度消失的問題，雖然今天我們已經不用 Hinton 教授當初提出的方法，但我們始終感謝 Hinton 教授這二十年對於深度學習的貢獻。

那我們今天怎麼解決梯度消失的問題呢？有兩種想法

1. 想法 1: 讓你的激勵函數的微分等於 1 就不會打折了
2. 想法 2: 打折就好像我們人在忘記東西，那不要讓事情太容易忘就好，也就是跨層的神經元連接

想法 1: 微分等於 1 於是我們使用 relu 函數來替代 Logistic 函數



relu 簡單來說就是讓函數 (負數) = 0 函數 (正數) = 數自己
你可以看到在正數的範圍我們都是微分 1!! 解決
但是 relu 也是有幾個小問題

1. 激勵幅度的中心不是 0，不平均
2. 負類別微分永遠 =0，代表很多神經元不會更新，相當於死掉了！所以後續有一些改進，如 leakly relu

但是這些小問題對於他的表現影響很小!! 也是我們今天採用的重要方法!!

想法 2: 跨層連接 我們後續會在 LSTM(長短期記憶模型) 和 ResNet(進階的圖像辨識模型) 看到這種特別的想法

1.3 需要函式庫

機器學習的函式庫有很多, ex. Caffe, PyTorch, Keras, Theano…等等，我們的選擇如下:

1. TensorFlow: 請使用 PyCharm 或者 pip 安裝 tensorflow，我們後續才會教你如何直接使用 tensorflow
2. Keras: 請使用 PyCharm 或者 pip 安裝 keras，將底層實作再包一層，讓你快速建立模型，當然在彈性上會有些損失，支援 TensorFlow 和 Theano 兩種底層

1.4 Step1. 資料預處理

這裡我們選用內建的 mnist 手寫數字資料庫來，mnist 提供共 70000 筆手寫數字，而且用 keras 讀取的時候會直接幫你分成訓練和測試兩份資料

```
In [1]: import keras
        from keras.datasets import mnist
        from keras.models import Sequential
        from keras.layers import Dense,Dropout
        import matplotlib.pyplot as plt
        %matplotlib inline
        # 我們會使用到一些內建的資料庫, MAC 需要加入以下兩行, 才不會把對方的 ssl 憑證視為無效
        import ssl
        ssl._create_default_https_context = ssl._create_unverified_context
```

```
Using TensorFlow backend.
```

這裡讀取的時候比較不一樣，load_data 給你的是 tuple 中的 tuple，load_data 回傳值 ((訓練特徵, 訓練目標), (測試特徵, 測試目標))

所以你要接的時候必須再加上 ( ) 來接裡面的 tuple

```
In [2]: # 回傳值: ((訓練特徵, 訓練目標), (測試特徵, 測試目標))
        (x_train, y_train),(x_test, y_test) = mnist.load_data()
```

```
In [3]: print("訓練資料筆數:", len(y_train))
        print("測試資料筆數:", len(y_test))
```

```
訓練資料筆數: 60000
測試資料筆數: 10000
```

```
In [4]: print("特徵的維度:", x_train.shape)
```

```
特徵的維度: (60000, 28, 28)
```

稍微用 dataframe 看一下我們資料，我選擇其中的第一筆特徵來看，這是一個黑白圖像
所以你可以看到我們每張圖是 28 * 28 pixel，每個 pixel 最大值 255(白) 最小值 0(黑)

```
In [5]: import pandas as pd
        # 為了顯示的漂亮, 我刻意的把印出來的 row 只顯示 15 個和 column 只顯示 10 個
        # 大家練習的時候可以去掉下面兩行
        pd.set_option('display.max_rows', 15)
        pd.set_option('display.max_columns', 10)
        pd.DataFrame(x_train[0])
```

Out[5]:

	0	1	2	3	4	...	23	24	25	26	27
0	0	0	0	0	0	...	0	0	0	0	0
1	0	0	0	0	0	...	0	0	0	0	0
2	0	0	0	0	0	...	0	0	0	0	0
3	0	0	0	0	0	...	0	0	0	0	0
4	0	0	0	0	0	...	0	0	0	0	0
5	0	0	0	0	0	...	127	0	0	0	0
6	0	0	0	0	0	...	64	0	0	0	0
..	..	..	..	..	...	...	...	..	..	..	..
21	0	0	0	0	0	...	0	0	0	0	0
22	0	0	0	0	0	...	0	0	0	0	0
23	0	0	0	0	55	...	0	0	0	0	0
24	0	0	0	0	136	...	0	0	0	0	0
25	0	0	0	0	0	...	0	0	0	0	0
26	0	0	0	0	0	...	0	0	0	0	0
27	0	0	0	0	0	...	0	0	0	0	0

[28 rows x 28 columns]

這裡我們會習慣做標準化 (把所有的資料縮放到 0 - 1 間)，標準化的兩個原因

1. 希望所有特徵影響的幅度一樣，如果有個特徵區間明顯比別人大，那在梯度下降的時候就比較會傾向往它走，調整它
2. 如果你的特徵區間太大，一個梯度下降的步幅相對來說就比較大，很容易超過你的最低點

這裡我們標準化的原因主要是 2，不過建議你不管如何，都養成標準化的好習慣

另外記得將你的輸出做成 One-Hot Encoding，因為我們在使用深度學習的時候，最後一層可以是多個神經元

每個神經元就代表了一件事的正負，像我們現在的判定有 0 - 9，最後一層就可以是 10 個神經元，看誰被激發的幅度比較大，答案就是誰!

```
In [6]: from keras.utils import np_utils
        # reshape 讓他從 32 * 32 變成 784 * 1 的一維陣列
        # 除以 255 讓我們標準化到 0-1 區間
        x_train_shaped = x_train.reshape(60000, 784).astype("float32") / 255
        x_test_shaped = x_test.reshape(10000, 784).astype("float32") / 255
        # keras 要求你的分類輸出必須換成 One-hot 模式
        y_train_cat = np_utils.to_categorical(y_train)
        y_test_cat = np_utils.to_categorical(y_test)
```

取一個目標讓你看看 One-Hot Encoding 前後的改變

```
In [7]: print("One-hot 前:", y_train[0])
        print("One-hot 後:", y_train_cat[0])
```

```
One-hot 前: 5
One-hot 後: [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

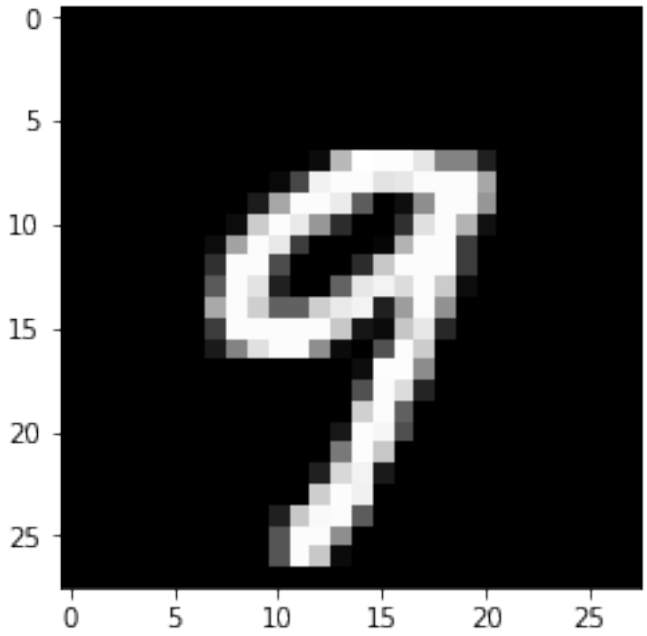
利用 input 讓你看看你想看的圖片，記得輸入完要按下 Enter 才可以繼續往下跑 Cell

```
In [8]: a = int(input("請輸入你想可視化的圖片 [0-59999]:"))
        print("你想可視化的圖片號碼是", a)
        print("圖片答案是", y_train[a])
```

```
請輸入你想可視化的圖片 [0-59999]:45
你想可視化的圖片號碼是 45
圖片答案是 9
```

```
In [9]: plt.imshow(x_train[a],cmap='gray')
```

```
Out[9]: <matplotlib.image.AxesImage at 0x1052cb7f0>
```



1.5 Step2. 建立模型

在建立模型的時候你必須選擇一個 Model 然後將你的一層一層神經元疊上去，我們接下來先解釋一下我們重要的選擇和參數設定

1.5.1 Model 選擇

Keras 的 Model 有兩種模型，Sequential 和 Functional，Sequential 使用上比較簡單，就一層一層 Layer 疊上去就可以，也可以滿足我們大部分的需要，Functional 的模型可以做出更多特別的設置，在使用上會比較彈性

1.5.2 Layer 選擇

我們現在只用到最簡單的一個層，叫做全連接層 (Dense Layer)，就是每一個神經元都會貢獻給下一層的神經元

這裡一個比較大的問題就是你的每一層應該有多少神經元，我們分三個部分討論

1. 輸入層 (最初層): 無庸置疑，你有多少個特徵就應該有多少神經元
2. 輸出層 (最後層): 無庸置疑，你有多少個分類要判斷就應該有幾個最後的神經元

3. 隱藏層 (中間層): 這裡沒有公式，有的只有經驗法則，只能試出一個比較佳的值，還好前人已經給你一些經驗可以借鑑，見下圖的三種選擇你都可以先試試看再微調

$m = \sqrt{n+l} + \alpha$
$m = \log_2 n$
$m = \sqrt{nl}$
m：隱含层节点数
n：输入层节点数
l：输出层节点数
α：1--10 之间的常数。

1.5.3 W 權重初始

因為我們要使用梯度下降，所以我們需要選擇一組隨機的權重開始，通常我會喜歡使用常態分佈 (random_normal) 來選擇我的初始權重

1.5.4 激勵函數

如同我們之前所說的，在層和層之間我們會選擇 relu 當我們的激勵函數，因為這樣才可以解決梯度消失的問題

問題在最後一層我們要選擇什麼激勵函數呢？以前我們選擇 sigmoid 也就是 Logistic 函數來激勵，Logistic 在單一的判斷的時候沒有問題

但在多個判斷的時候會有個小問題，就是所有的類別的判斷值加起來不等於 1，跟我們對於機率的認知不同 (所有機率加起來 = 1)

所以我們改使用 softmax 函數，長個跟 sigmoid 有 87% 像，但是加起來會 = 1

假設我們有兩個神經元，算出分數分別是 a 和 b，softmax(a) = e^a / (e^a + e^b)，softmax(b) = e^b / (e^a + e^b)

讀者可以輕易看出 softmax(a) + softmax(b) = 1

在這裡列出詳細的數學定義

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

```
In [10]: from keras.models import Sequential
         from keras.layers import Dense
         model = Sequential()
         # 一層隱藏層的模型，第一個隱藏層記得要寫出特徵的數目: input_dim
```

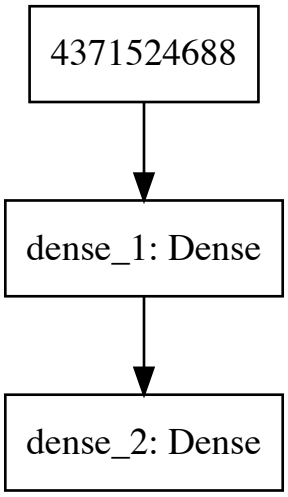
```
h_layer = Dense(units = 256,
                input_dim = 784,
                kernel_initializer = "random_normal",
                activation = "relu")
model.add(h_layer)
o_layer = Dense(units = 10,
                kernel_initializer = "random_normal",
                activation = "softmax")
model.add(o_layer)
```

你可以畫出方塊圖 (記得要安裝 graphviz)

```
In [11]: 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[11]:



更仔細的參數設定你可以藉由 summary 來看到，這裡解釋一個東西，權重 (Param) 的個數
因為是全連接，所以至少需要
第一隱藏層: 784 * 256 = 200704 輸出層: 256 * 10 = 2560
但是每一個輸出，應該要配置一個 bias 常數 (f = wx + b 的 b 常數) 來讓你的激勵函數還是可以保持一樣，所以總數是
第一隱藏層: 784 * 256 + 256(bias 個數) = 200960 輸出層: 2560 + 10(bias 個數) = 2570

```
In [12]: model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 256)               200960
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2570
=================================================================
Total params: 203,530
Trainable params: 203,530
Non-trainable params: 0
_________________________________________________________________
```

設置完記得 compile 一下你的模型

optimizer 參數：梯度下降的方式，傳統的梯度下降有一個小缺點，就是你如果走太小步，要訓練很久，走太大步容易錯過最小值，所以我們做出動量的修改，也就是在可以走大步 (你可以想像成斜率還很大的時候) 的時候盡量大步，走小步的時候小步一點，這裡我們通常使用 adam 優化過的梯度下降就可以了

參考: https://zhuanlan.zhihu.com/p/22252270

metrics 參數：預設在輸出的時候只會告訴你 loss 是多少，你需要什麼量度的話要額外告訴他，我這裡告訴他的量度是正確率

```
In [13]: model.compile(loss="categorical_crossentropy",
                       optimizer = "adam",
                       metrics = ['accuracy'])
```

開始 fit 你的模型，這裡也解釋一下重要的參數

validation_split：多少用來訓練，多少用來測試

batch_size：我們使用的是梯度下降，所以現在的問題是，到底看過多少個樣本更新一次梯度，如果只看一個樣本就更新一次，會有亂走的問題，因為一個樣本跟你整體的趨勢差距太大了！如果看完全部樣本才更新一次，也會有一點問題，第一個問題是內存可能沒辦法一次載入全部樣本，所以計算時間會變得很大，第二個問題是全部樣本才更新一次，但你還是需要足夠的更新次數才能走到低點，假設你需要 10 次更新次數，那就是要把整個樣本看十次才能走到你的低點，這是一個非常非常大的計算量！所以我們做了一個折衷，我們一次看完很多個 (但不是全部) 就做一次權重更新，這裡我設置成看完 200 個樣本做出一次權重更新，建議你在這裡設置成 100~200 間的數值

epochs：每一個樣本要看過幾次，這裡我設成 10，也就是整份樣本我必須看過 10 次，建議從 10~20 開始試

verbose：0(不輸出訓練 log) 1(只輸出進度條) 2(每一個 epoch 輸出訓練信息)，我們通常就選擇 2

```
In [14]: train_history = model.fit(x = x_train_shaped, y = y_train_cat,
                                   validation_split = 0.1,
                                   epochs = 10,
                                   batch_size = 200,
                                   verbose = 2)
```

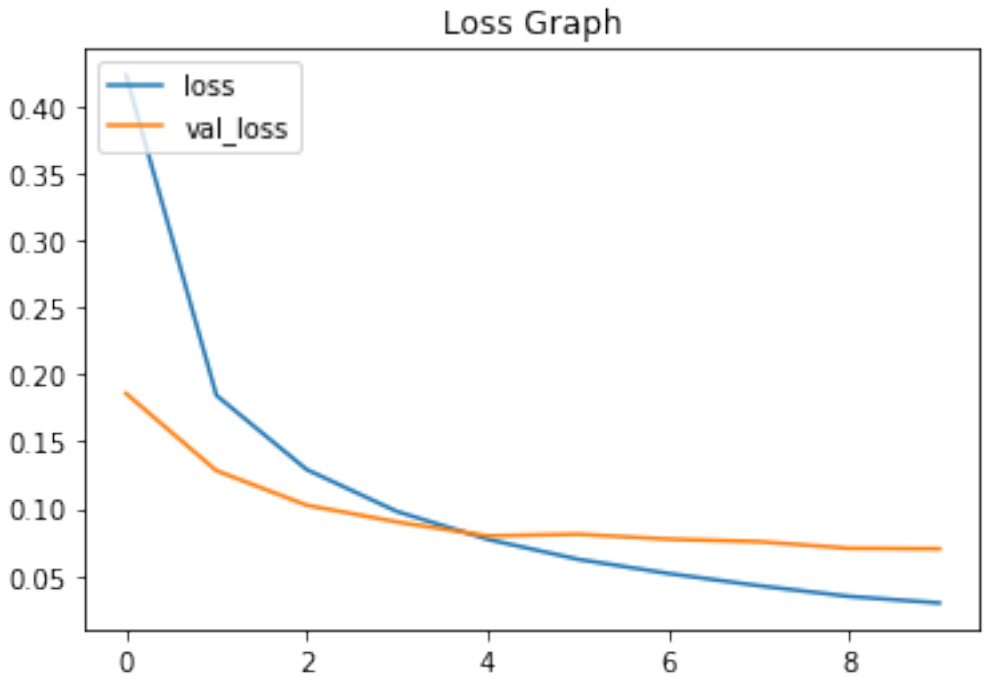
```
Train on 54000 samples, validate on 6000 samples
Epoch 1/10
 - 1s - loss: 0.4229 - acc: 0.8873 - val_loss: 0.1857 - val_acc: 0.9503
Epoch 2/10
 - 1s - loss: 0.1843 - acc: 0.9474 - val_loss: 0.1282 - val_acc: 0.9658
Epoch 3/10
 - 1s - loss: 0.1290 - acc: 0.9632 - val_loss: 0.1025 - val_acc: 0.9700
Epoch 4/10
 - 1s - loss: 0.0978 - acc: 0.9724 - val_loss: 0.0900 - val_acc: 0.9745
Epoch 5/10
 - 1s - loss: 0.0772 - acc: 0.9786 - val_loss: 0.0797 - val_acc: 0.9760
Epoch 6/10
 - 1s - loss: 0.0623 - acc: 0.9826 - val_loss: 0.0809 - val_acc: 0.9768
Epoch 7/10
 - 1s - loss: 0.0517 - acc: 0.9852 - val_loss: 0.0773 - val_acc: 0.9773
Epoch 8/10
 - 1s - loss: 0.0426 - acc: 0.9883 - val_loss: 0.0754 - val_acc: 0.9768
Epoch 9/10
 - 1s - loss: 0.0346 - acc: 0.9913 - val_loss: 0.0705 - val_acc: 0.9790
Epoch 10/10
 - 1s - loss: 0.0297 - acc: 0.9925 - val_loss: 0.0701 - val_acc: 0.9800
```

畫個很重要的圖，loss 和 val_loss 的圖，簡單來說，loss 是你的模型跟所有訓練資料 (我們剛剛切出 9/10 訓練) 正確答案的差距，val_loss 是你的模型跟你切出來的驗證資料 (我們剛剛切出 1/10 驗證) 正確答案的差距，別忘了我們在前面的課程跟你強調過 ‘‘過擬合’’ 的問題，就是過度訓練的意思，那通常我看 val_loss 沒什麼大的變動的時候 (再訓練也對沒看過的資料沒啥幫助) 我就會停下

來了，去避免過度訓練，你可以看到其實我們的 10 epoch 是差不多的，val_loss 已經達到他能下降的極致了。

```
In [15]: 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[15]: <matplotlib.legend.Legend at 0x1283c3be0>
```



你可以輸出預測的分類

```
In [20]: # 取五筆給你看
         pre = model.predict_classes(x_test_shaped)
         print("預測標籤:", list(pre[:5]))
         print("正確標籤:", list(y_test)[:5])
```

```
預測標籤: [7, 2, 1, 0, 4]
正確標籤: [7, 2, 1, 0, 4]
```

```
In [18]: e = model.evaluate(x_test_shaped, y_test_cat)
         print("衡量係數:", e)
         print("正確率:", e[1] * 100, "%")
```

```
10000/10000 [==============================] - 0s 17us/step
衡量係數: [0.07355162749246229, 0.9778]
正確率: 97.78 %
```

看看是什麼被分類錯，我們使用之前說過的混淆矩陣，你也可以使用之前說過 sklearn 的一些衡量唷!

```
In [22]: from sklearn.metrics import confusion_matrix
         pd.DataFrame(confusion_matrix(y_test, pre))
```

Out[22]:

	0	1	2	3	4	5	6	7	8	9
0	971	0	1	1	1	1	1	1	2	1
1	0	1119	4	0	0	1	2	2	7	0
2	5	1	1011	1	1	0	2	5	6	0
3	2	0	5	992	0	1	0	4	4	2
4	1	0	6	0	964	0	2	1	0	8
5	4	0	0	14	1	864	4	0	2	3
6	4	2	1	1	4	3	941	0	2	0
7	2	3	10	3	0	0	0	1002	4	4
8	4	0	5	4	4	2	1	3	949	2
9	3	4	0	8	15	3	0	6	5	965

列標籤是正確的標籤，行標籤是你的預測

你可以看到 9 由於和 7 和 4 長得非常像，所以被判斷錯的機率就還蠻高的！0 長得獨樹一幟，所以分類錯誤的機率真的還蠻小的!