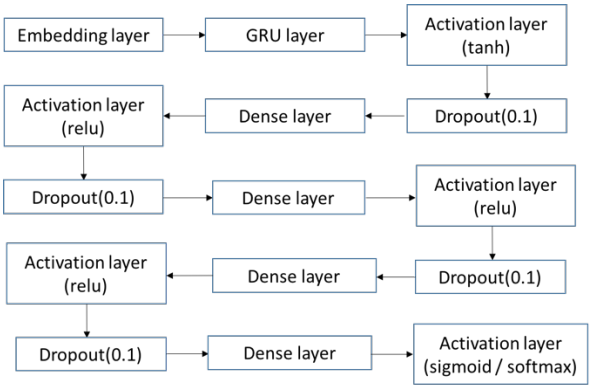
學號：B03901101 系級： 電機三 姓名：楊其昇

1. (1%)請問 softmax 適不適合作為本次作業的 output layer? 寫出你最後選擇的 output layer 並說明理由。

本題解釋所依據的模型架構（和 sample code 一樣）如下：

（loss function=categorical_crossentropy, optimizer = adam）



Softmax function 的定義如下：

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{k} e^{z_k}} \; for\; j = 1, \ldots, k$$

由 softmax 的定義甚至由字面上的意思便可以知道 softmax 的功能主要是用來挑出許多 input 中最大的那一個，並再考慮其他較小的 input，而其他較小的 input 則會被弱化。而在本次作業中，由於是要做 multi labeling，若使用 softmax function 當作 output layer，則模型往往只會挑出最有可能的選項而忽略其他的可能選項，也就是說 softmax function 並不適合用來做複選題，只適合拿來做單選題，如果堅持要使用 softmax function 的話，那麼相對應的解決辦法便可能要去調整預測機率的門檻，在所有種類全部都用單一門檻的情況下，出來的結果可能會很差。因此，我們需要一個能夠挑出所有可能選項的 function，能夠強調可能的選項而非僅僅強化最有可能的選項而弱化其他選項。而 sigmoid function 的特性為將大於 0 的 input 的往 1 的方向拉，越大越靠近 1；小於 0 的部分則往 0 的方向拉，越小越靠近 0，而這個特性並不會受到其他 input 的干擾，所以最後我用 sigmoid function 來作為 output layer。

2. (1%)請設計實驗驗證上述推論。

在同樣使用助教 sample code 的模型架構並使用相同參數的情況下，只改變 output layer 的 function，最後用來判定是否有這項 label 的機率門檻統一設為 0.4。在使用 softmax function 的情況下，很明顯便可以發現，不論是 training data 或者 validation data 的 f1_score 表現都奇差無比（如下圖）：

```
Epoch 1/55
4468/4468 [==============================] - 142s - loss: 6.7318 - f1_measure: 0.0021 - val_loss: 5.8823 - val_f1_measure: 0.0000e+00
Epoch 2/55
4468/4468 [==============================] - 140s - loss: 6.1793 - f1_measure: 0.0029 - val_loss: 5.7223 - val_f1_measure: 0.0000e+00
Epoch 3/55
4468/4468 [==============================] - 138s - loss: 6.0400 - f1_measure: 0.0023 - val_loss: 5.7054 - val_f1_measure: 0.0000e+00
Epoch 4/55
4468/4468 [==============================] - 140s - loss: 5.9030 - f1_measure: 0.0069 - val_loss: 5.5809 - val_f1_measure: 0.0000e+00
Epoch 5/55
4468/4468 [==============================] - 140s - loss: 5.7383 - f1_measure: 0.0122 - val_loss: 5.3751 - val_f1_measure: 0.0013
```

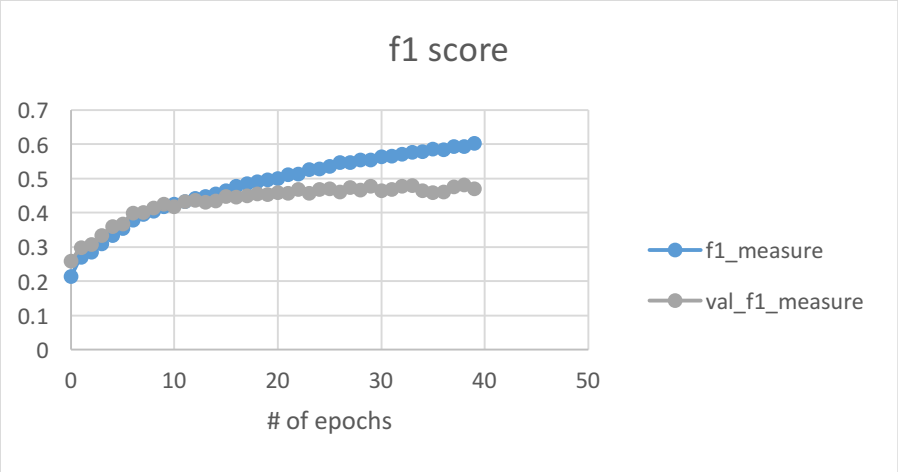
要等幾個 epoch 之後，validation data 的 f1_score（即圖中的 f1_measure，算法和衡量

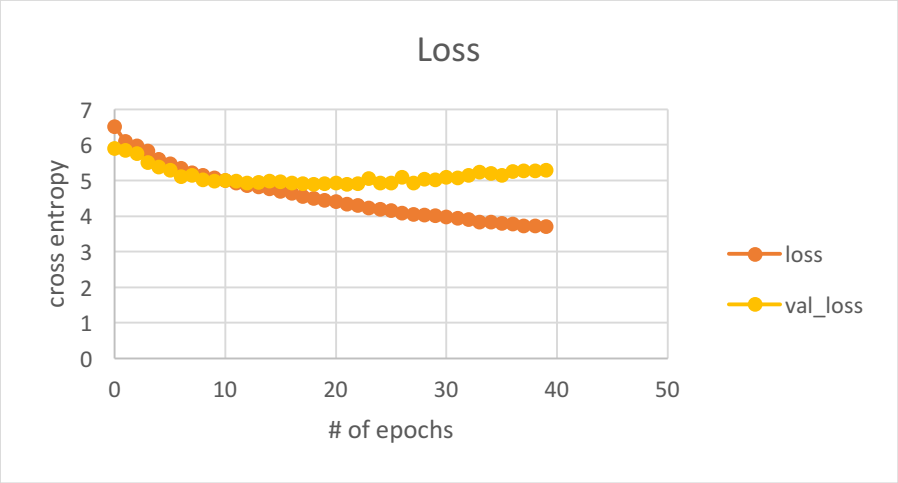
門檻和助教的 sample code 一模一樣）才會開始慢慢有非 0 的值產生，但往往又經過幾個 epoch 後，f1_score 常常會再跑到 0 附近，而 training data 的 f1_score 表現也很差，也就是說在 f1_score 這項衡量標準下，這樣訓練的過程可能是有問題的，或者說是和預期目標不符合。而如果硬著頭皮將程式整個跑完並上傳到 Kaggle 後，便發現 public 部分的成績只有 0.28275。如果簡單草率的估計一下，假設模型預設出來的結果只會有一項 label 的機率大於 0.4（機率門檻值），即每次只會有一個 label，然後有一半的機率會標成功，並再假設正確的 label 有三項：先考慮標成功的情形，precision 很直接便等於 1，$recall = \frac{True\ positive}{True\ positive + false\ negative} = \frac{1}{1+2} = \frac{1}{3}$，而 f1_score 為兩者的調和平均，因此 f1_score = 0.5；考慮猜錯的情形，無論 precision 或是 recall 皆為 0，所以 f1_score = 0。考慮這兩種情形的發生機率差不多，所以最後的 f1_score = 0.25。可以發現雖然這個估算忽略了許多種情形，但最後實際的結果其實還滿接近一開始粗略的假設及第一題的解釋。

若改成 sigmoid function 當作 output layer，和原本使用 softmax function 的版本相比，很明顯 f1_score 的表現就好很多（如下圖）：

```
Epoch 1/55
4468/4468 [==============================] - 149s - loss: 6.8343 - f1_measure: 0.1630 - val_loss: 6.0418 - val_f1_measure: 0.2235
Epoch 2/55
4468/4468 [==============================] - 141s - loss: 6.2984 - f1_measure: 0.2253 - val_loss: 5.8503 - val_f1_measure: 0.2638
Epoch 3/55
4468/4468 [==============================] - 140s - loss: 6.1379 - f1_measure: 0.2538 - val_loss: 5.8095 - val_f1_measure: 0.2808
Epoch 4/55
4468/4468 [==============================] - 141s - loss: 6.0497 - f1_measure: 0.2664 - val_loss: 5.7808 - val_f1_measure: 0.2862
Epoch 5/55
4468/4468 [==============================] - 142s - loss: 5.9671 - f1_measure: 0.2788 - val_loss: 5.7316 - val_f1_measure: 0.2954
Epoch 6/55
4468/4468 [==============================] - 140s - loss: 5.8392 - f1_measure: 0.2944 - val_loss: 5.5237 - val_f1_measure: 0.3281
```

最後耐心訓練完模型，並上傳到 kaggle 的結果 public 的部分為 0.44238，而這結果也滿符合第一題中的預期，下表為訓練過程：





可以發現僅僅改變 output layer 的 function 之後，相較於 softmax function，sigmoid function 更符合提高 f1 score 這項衡量標準的目標，不過由於也可以明顯發現這個架構的模型很容易便 overfitting，在大約第十個 epoch 後便已無法在 validation data 上求進步，因此若要進一步提升 f1 score 的表現，便要進一步考慮別種避免 overfitting 的手段或者模型架構。

不過順帶一提，第一題中有提到若堅持用 softmax function 當作 output layer 的話，那麼想提高模型的表現，其中一個辦法便是調整每個種類的門檻，而在網路上稍微搜尋了一下，發現似乎滿常利用 Matthews correlation coefficient 當作調整門檻的依據，實作方式如下：

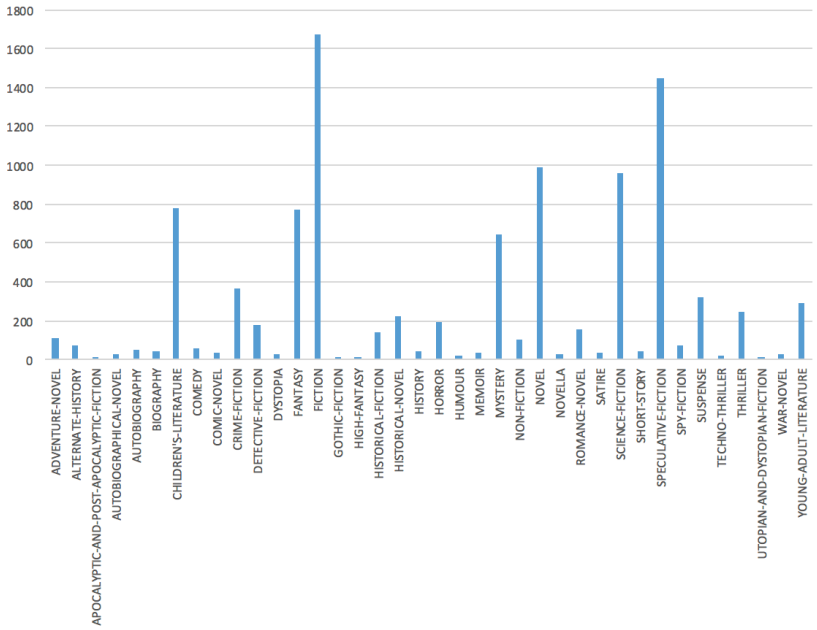
```
acc = []
accuracies = []
threshold = np.arange(0.1,0.9,0.05)
best_threshold = np.zeros(out.shape[1])
for i in range(out.shape[1]):
    y_prob = np.array(out[:,i])
    for j in threshold:
        y_pred = [1 if prob>=j else 0 for prob in y_prob]
        acc.append( matthews_corrcoef(y_test[:,i],y_pred))
    acc   = np.array(acc)
    index = np.where(acc==acc.max())
    accuracies.append(acc.max())
    best_threshold[i] = threshold[index[0][0]]
    acc = []

y_pred = np.array([[1 if out[i,j]>=best_threshold[j] else 0 for j in range(y_test.shape[1])] for i in range(len(y_test))])
```

簡單來說便是設定幾種不同高低的門檻，並再利用 validation data 在訓練好模型中所預測出來的機率和正確解答相比較，最後挑出適合的門檻。最後根據這個方式，應用到 softmax function 所預測出來的機率上，丟到 Kaggle 後，發現 public 部分的 f1_score = 0.47330，意外的還滿好的，不過若將這個方法也應用到 sigmoid function 所預測出來的機率上，其實效果跟直接全部設成 0.4 差不了太多。

3. (1%)請試著分析 tags 的分布情況(數量)。

以下為各種類數量的長條圖：



下表為所有的種類的數量分布：

label	number	label	number	label	number
ADVENTURE-NOVEL	109	HIGH-FANTASY	15	SPECULATIVE-FICTION	1448
ALTERNATE-HISTORY	72	HISTORICAL-FICTION	137	SPY-FICTION	75
APOCALYPTIC-AND-POST-APO	14	HISTORICAL-NOVEL	222	SUSPENSE	318
AUTOBIOGRAPHICAL-NOVEL	31	HISTORY	40	TECHNO-THRILLER	18
AUTOBIOGRAPHY	51	HORROR	192	THRILLER	243
BIOGRAPHY	42	HUMOUR	18	UTOPIAN-AND-DYSTOPIAN-FIC	11
CHILDREN'S-LITERATURE	777	MEMOIR	35	WAR-NOVEL	31
COMEDY	59	MYSTERY	642	YOUNG-ADULT-LITERATURE	288
COMIC-NOVEL	37	NON-FICTION	102		
CRIME-FICTION	368	NOVEL	992		
DETECTIVE-FICTION	178	NOVELLA	29		
DYSTOPIA	30	ROMANCE-NOVEL	157		
FANTASY	773	SATIRE	35		
FICTION	1672	SCIENCE-FICTION	959		
GOTHIC-FICTION	12	SHORT-STORY	41		

可以簡單地發現，這次的 training data，各種種類的分佈數量極不平均，而我想這大概也是這次作業困難的地方：部分種類的樣本數相較於其他主要的數量過於稀少，造成訓練出來的模型往往最後容易在出現頻率較高的種類預測出較高的機率，最常出現的種類前三名為：FICTION、SPECULATIVE-FICTION、CHILDREN’S-LITERATURE，最少出現的前三名為 UTOPIAN-AND-DYSTOPIAN-FICTION、GOTHIC-FICTION、APOCALYPTIC-AND-POST- APOCALYPTIC-FICTION。平均分配下來每筆資料會約有 2.07 項 label。

4. (1%)本次作業中使用何種方式得到 word embedding?請簡單描述做法。

在這次作業中原本一開始是想要自己從 training data 中自己訓練出 word vector，不過後來發現效果實在是很差，因此在聽完 TA 時間後，便決定從網路上下載別人已訓練好的 word vector，我選的是 GloVe 裡用 2014 年英文維基百科訓練好的結果（https://nlp.stanford.edu/projects/glove/）。

在這次作業中我得到 word embedding 的流程如下，先利用 keras 內建的 Tokenizer 將

training data 和 testing data 中所有的字作斷詞並且編號然後將 testing data 從文字轉成 sequences、把每筆資料同樣利用 Tokenizer 內建的函式 padding 成一樣的長度，再載入從網路上別人以訓練好的 word vector，並根據 training data 中所有文章的數量、training data 和 testing data 中不同詞的總數以及最多的字數（即之前 padding 的長度）得到 Embedding matrix，再以此作為 Embedding layer 中的 weights（將 trainable 設為 false），最後將以處理好的 sequences 輸入 Embedding layer 即可以得到 word embedding。

5. (1%)試比較 bag of word 和 RNN 何者在本次作業中效果較好。

在 bag of word的實作方面，我是利用 sklearn 這個 Python 套件裡面的 TFidfVectorizer 將文字轉成 word vector。Tf-idf 全名為 term frequency - inverse document frequency，一個字詞的重要性會隨著它在每一篇文章中出現的次數呈正相關增加，但同時又會隨著在 corpus 中出現的頻率成反比下降。舉例來說，某篇科幻小說的內容中相較於其他文章可能常常出現外星人、太空船這一類的詞，但在其他種類的文章中卻會很少出現，但像是我、你、他這一類的詞幾乎在全部的文章中都很常見，因此我們便會認為太空船、外星人這一類的詞對於判斷這篇文章的種類有著一定的重要性，但我、你、他這類的詞大概對於判斷文章的種類並沒有什麼幫助。Tf-idf 便是在這樣的理論之下所發展出來的其中一種 bag of word，不過還有另外考慮這個詞還有在其他文章中出現的比率。在得到 word embedding 之後，在出於考量得到的 word vector 維度有點大，直接丟進去 RNN 跑會滿曠日費時的，我並沒有使用 neural network 的架構來進行分類，而是直接使用作業四中助教曾經在 sample code 中示範過的 SVC 來進行分類。實作方面同樣是利用 sklearn 裡面的 SVR 這個套件。在簡單試了幾種 SVC 之後，我發現 Linear SVC 出來的效果最好，再簡單的調了一下 penalty 再上傳到 Kaggle 後，public 的部分的 f1 score = 0.5011，其實還滿不錯的，輕輕鬆鬆就過了 simple baseline。不過這個方法也有它的極限，不論怎麼調整參數，比較好的表現大概都在 0.50 ~ 0.51 徘徊，很難再得到更好的結果，更別說超過 strong baseline 了。SVR 所使用的參數如下：
SVR(kernel='linear', degree=3, tol=0.001, C=0.28, epsilon=0.1, shrinking=True, max_iter=-1)

在 RNN 的實作方面，從第二題中便可以發現這次作業還滿容易發生 overfitting 的問題的，因此在用了第四題中所提到的方式得到 word embedding 後，為了提高在 validation data 上的表現，除了提高 dropout 的比例、增加每層的 neuron 數目、加深 DNN 或者 RNN 的 layer 深度，甚至參考網路上所提到的做法：在 embedding layer 後面接幾層 CNN 希望能達到 ngram 的效果，之後再接 RNN 及 DNN 來進行分類，不過最後無論怎麼做，上傳到 kaggle 後的 public 部分的 f1 score 大概都只在 0.44 ~ 0.49 間徘徊，不僅沒有超過 simple baseline，還慘輸 bag of word 裡用的 SVR。最後在到處和同學討論，以及去網路上翻老師以前的上課內容後，最終使用了 ensemble 中的 Bagging 來提升種類辨識的表現：也就是說訓練出幾個在 validation data 上還過得去的模型，然後預測的時候再根據這些模型的預測結果投票出最終結果。實作上，根據 validation data 上的 f1 score 及部分曾經上傳到 Kaggle 的模型表現之後，審慎地挑了五個模型來預測並投票最後結果，最後上傳到 Kaggle，在 public 部份的 f1 score = 0.52068，雖然仍然尚未超過 strong baseline，不過總算是高過了 bag of word 的表現。

Bag of word 和 RNN 在這次作業的實作上 embedding layer 的實作難度都差不多，都是使用別人寫好的套件或函式，簡單調整一下參數便能得到 word embedding。不過，由於在 bag of word 是使用 SVR 的關係，因此訓練時間以及每次訓練出來的穩定度都比 RNN 要

好上許多，但卻很難再更上一層樓。而 RNN 雖然有許多諸如訓練時間費時、想要調整到最佳的模型架構和參數相當困難等缺點之外，其實若能克服或者容忍前面所提到的幾項缺點，RNN 的表現還是可以超過使用 bag of word 的實作方式。不過，在這次作業中，RNN 表現不盡理想的原因，我想除了 training data 本身資料分布就很不平均導致很難得到好的訓練模型之外，我想或許在這次作業中仍然有一些能夠提升 RNN 表現的關鍵方法尚未被我發現吧。