學號：B06901053　　系級：電機三　　姓名：謝承延

1. (1%) 請說明你實作的 RNN 的模型架構、word embedding 方法、訓練過程 (learning curve)和準確率為何？ (盡量是過 public strong baseline 的 model)

   Word embedding 方式跟 sample code 相同採用了 skip-gram

   Word vector 我用了 200, Iteration 10 次

   以下是 word embedding 的結果，我隨機選了幾個詞去找出最相近的前 10 筆 word 以觀測 word embedding vector 的好壞。

	fuck	cos	love	cos	bloody	cos	sorry	cos	join	cos
0	heck	0.554775	adore	0.672369	fucking	0.480964	sry	0.670564	introduce	0.540966
1	hell	0.528393	loove	0.656429	hotttt	0.459389	srry	0.613236	recommenc	0.537057
2	eff	0.506597	luv	0.648022	mml	0.459104	soz	0.58407	contribute	0.529421
3	fucked	0.468113	looove	0.637566	frickin	0.452186	sowwie	0.525167	visit	0.496561
4	shit	0.462448	looooove	0.614203	achey	0.445726	sorrry	0.511713	sponsor	0.473205
5	fuckin	0.457852	loves	0.601316	travesty	0.444389	apologize	0.511448	cgft	0.468872
6	ughhhh	0.453585	loooove	0.598953	grrr	0.443984	sowwy	0.49822	unsubscrib	0.466975
7	gahh	0.447871	loooooove	0.592105	frigging	0.442453	jonaskevin	0.495767	psa	0.463325
8	fuuck	0.446191	loveeee	0.58787	brrrrr	0.436292	shucks	0.492009	realllyy	0.46239
9	fck	0.445431	loveeeee	0.578273	hecka	0.433096	wishhhh	0.486717	meet	0.45639

   由結果可看到，找出的辭彙都的確跟原本詞彙意義蠻相近的。

因為 overfitting 很嚴重，所以我使用了 ensemble 此技巧，想法是打算利用每個 model 預測結果的高 variance 低 bias 的特性，去將結果平均，如此便能得到接近正確答案的結果。

判斷 model 有無改善的方法是用 cross validation 的結果取平均值，若發現 validation accuracy 和 validation loss 皆有改善，代表此調整是好的。

用多個 model 的 sigmoid 出來的結果，做 soft voting(不是對預測 label 做 voting)，這樣使我的 accuracy 馬上往上了大約 1.5% 來到 81.8%左右

，然而卻產生一個新的問題。

model 需要更多的資訊量，不然無法再更進一步改善 performance，因此我發現調整 sentence length 這個參數很重要，我將 sen_length 這個參數從 20 開始往上調以後，accuracy 又明顯的上升了。

最終 kaggle 上的準確率又繼續上升了 1%左右。

Model:

我共使用了五種 model 來做 ensemble，總參數量約為 300 萬

Model 1: 5 層的 Bidirectional GRU

Embedding_dim : 200

Hidden_dim :50

```
self.lstm = nn.GRU(embedding_dim, hidden_dim,
                   num_layers=1, batch_first=True, bidirectional=bi)
self.lstm2 = nn.GRU(hidden_dim*2, hidden_dim,
                    num_layers=1, batch_first=True, bidirectional=bi)
self.lstm3 = nn.GRU(hidden_dim*2, hidden_dim,
                    num_layers=1, batch_first=True, bidirectional=bi)
self.lstm4 = nn.GRU(hidden_dim*2, hidden_dim,
                    num_layers=1, batch_first=True, bidirectional=bi)
self.lstm5 = nn.GRU(hidden_dim*2, hidden_dim,
                    num_layers=1, batch_first=True, bidirectional=bi)
```

Model 2:三層 bidirectional LSTM

Embedding_dim : 200

Hidden_dim :50

```
self.lstm21 = nn.LSTM(embedding_dim, hidden_dim,
                      num_layers=1, batch_first=True, bidirectional=bi)
self.lstm22 = nn.LSTM(hidden_dim*2, hidden_dim,
                      num_layers=1, batch_first=True, bidirectional=bi)
self.lstm23 = nn.LSTM(hidden_dim*2, hidden_dim,
                      num_layers=1, batch_first=True, bidirectional=bi)
```

Model 3: 3 層 bidirectional GRU

Embedding_dim : 200

Hidden_dim :50

```
self.lstm31 = nn.GRU(embedding_dim, hidden_dim*4,
                     num_layers=1, batch_first=True, bidirectional=bi)
self.lstm32 = nn.GRU(hidden_dim*8, hidden_dim*4,
                     num_layers=1, batch_first=True, bidirectional=bi)
self.lstm33 = nn.GRU(hidden_dim*8, hidden_dim,
                     num_layers=1, batch_first=True, bidirectional=bi)
```

Model 4: 2 層 bidirectional GRU

```
    self.lstm41 = nn.GRU(embedding_dim, hidden_dim*5,
                         num_layers=1, batch_first=True, bidirectional=bi)
    self.lstm42 = nn.GRU(hidden_dim*10, hidden_dim,
                         num_layers=1, batch_first=True, bidirectional=bi)
```

Model 5: 5 層 bidirectional LSTM + droppout

```
    self.lstm51 = nn.LSTM(embedding_dim, hidden_dim,
                          num_layers=1, batch_first=True, bidirectional=bi, dropout=0.3)
    self.lstm52 = nn.LSTM(hidden_dim*2, hidden_dim,
                          num_layers=1, batch_first=True, bidirectional=bi, dropout=0.3)
    self.lstm53 = nn.LSTM(hidden_dim*2, hidden_dim,
                          num_layers=1, batch_first=True, bidirectional=bi, dropout=0.3)
    self.lstm54 = nn.LSTM(hidden_dim*2, hidden_dim,
                          num_layers=1, batch_first=True, bidirectional=bi, dropout=0.3)
    self.lstm55 = nn.LSTM(hidden_dim*2, hidden_dim,
                          num_layers=1, batch_first=True, bidirectional=bi)
```

其他相關參數:

```
# 定義句子長度、要不要固定 embedding、b
sen_len = 26
fix_embedding = True  # fix embeddin
batch_size = 128
epoch = 15
lr = 0.001
w2v_vector_dim = 200
```

Learning Curve:

我採用 cross validation 的方法，做三次的 validation，並且將

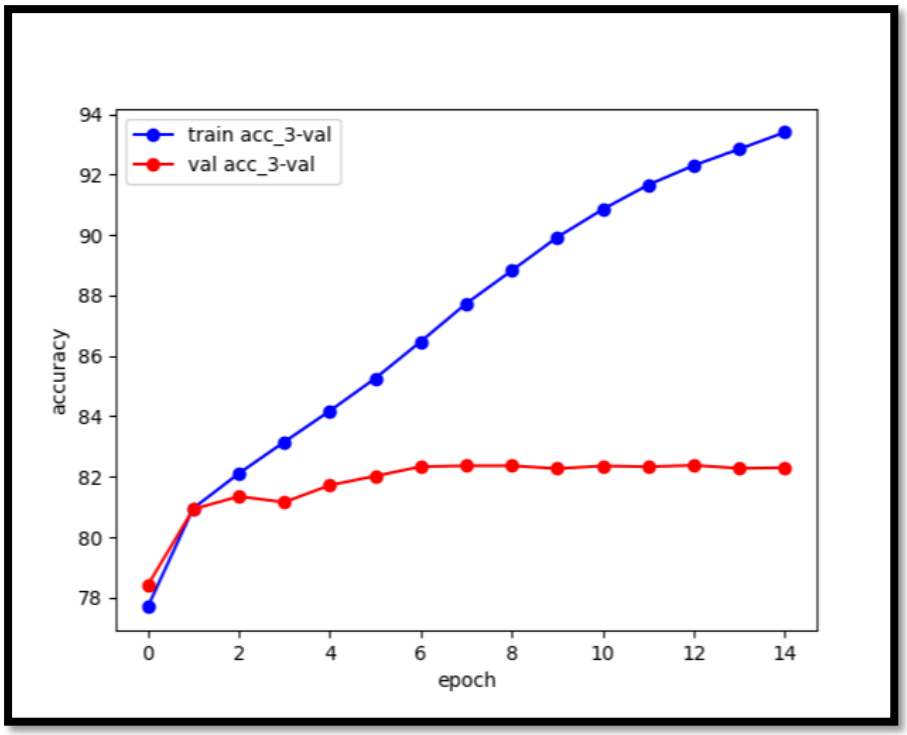
train accuracy

train loss

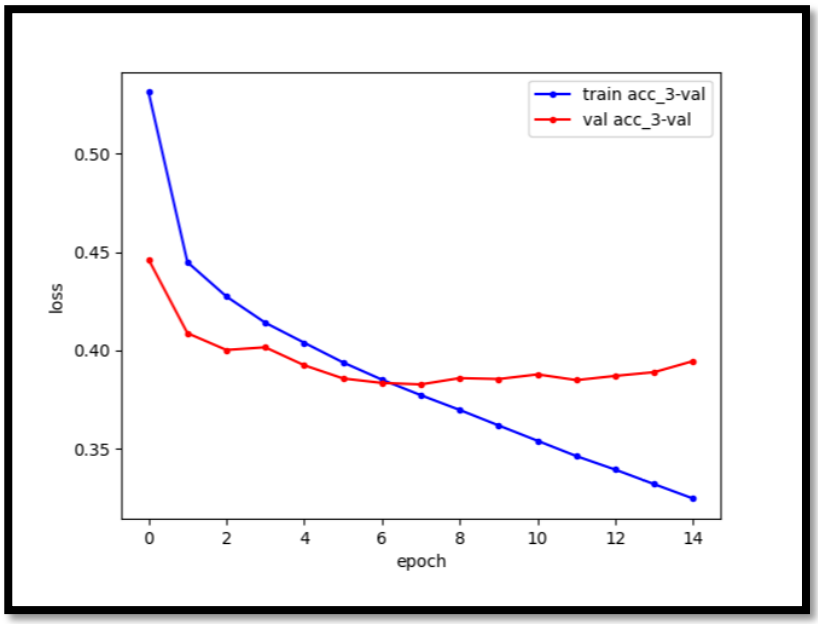
validation accuracy

validation loss 的平均畫成圖。

Accuracy:



LOSS:



2. (2%) 請比較 BOW+DNN 與 RNN 兩種不同 model 對於"today is a good day, but it is hot"與"today is hot, but it is a good day"這兩句的分數(過 softmax 後的數值)，並討論造成差異的原因。

RNN:

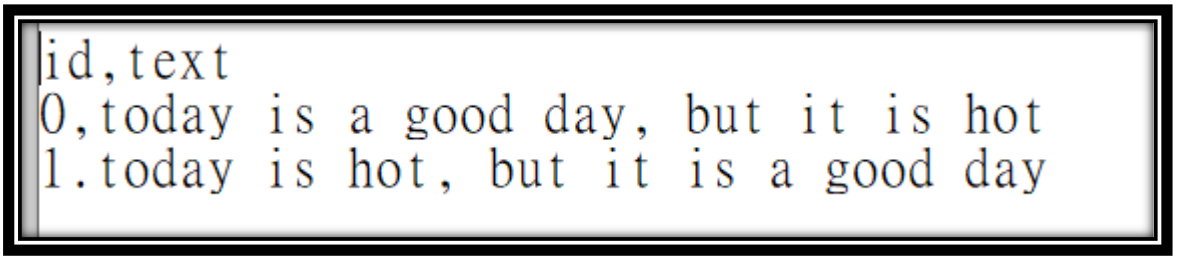
作法:因為在 testing data 中沒有找到這兩個句子，我另外建立了一個 txt 檔案紀錄這兩個句子，並將執行 binarization 前的 output 記錄下來(也就是 softmax 的數值)。

Model 是用 LSTM，word Embedding 是使用 skip-gram。

Batch size:128

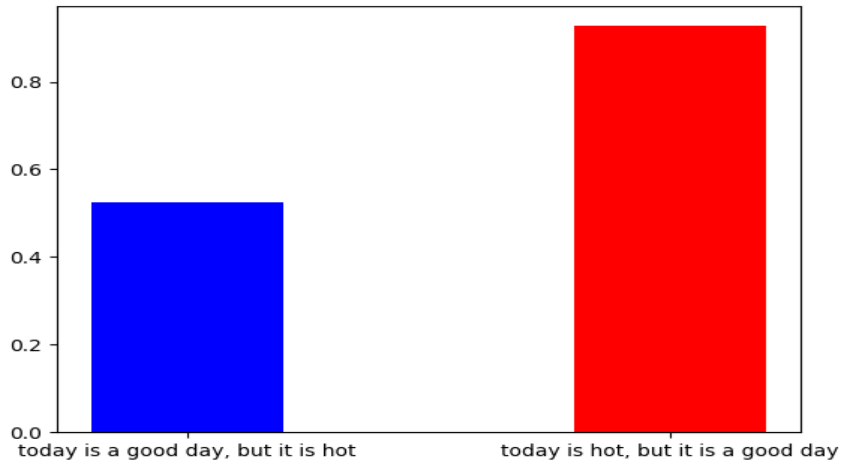
Epoch:5

結果如下:



id	score
0	0.524465
1	0.927484

可以看到”Today is hot, but it is a good day”的分數很高，比起另外一句更接近 postivie，”Today is a good day , but it is hot”的 score 則介於 0.52 左右，雖然最後被歸類在 class 1，但從分數看起來，分類效果是不太好的。



BOW+DNN:

我將每個 word 對應到一個 index，在 train 的時候把每個 batch 的 sentence list 變成 bag of word 的形式(從#sentence length 個 word 變成 dimension 為總 word 數的 vector)，然後接上 DNN。

```
self.classifier = nn.Sequential(
    nn.Linear(24696, hidden_dim),
    nn.Dropout(dropout),
    nn.Linear(hidden_dim, 1),
    nn.Sigmoid())
```

Batch size Epoch 皆跟 RNN 相同，差別僅在 DNN 多了一層 layer。

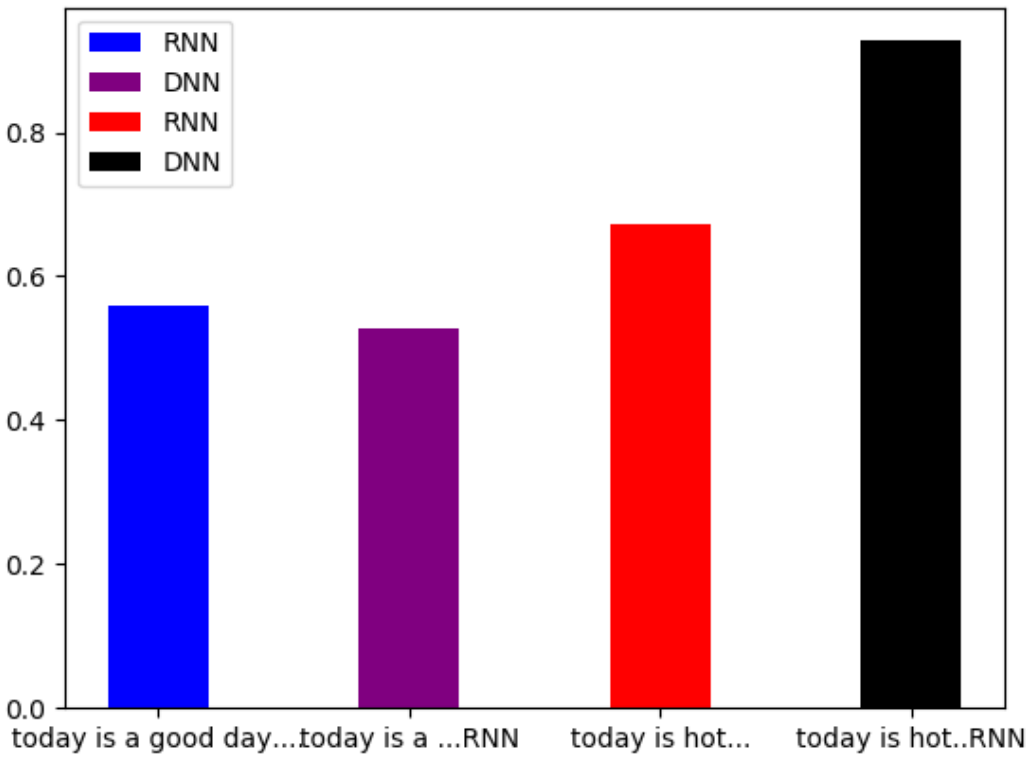
參數量變化:因為多了一層 layer 要將 bag of word 的 vector 作轉換，參數量從 241351->3945901，train 的時間變長了很多。

```
start training, parameter total:10119901, trainable:3945901
```

Result:
Train accuracy 變化:81.530-→80.655
Validation accuracy 變化:79.563->76.363

```
id,text
0,today is a good day, but it is hot
1,today is hot, but it is a good day
```

id	score
0	0.558852
1	0.672923



可以看到使用 RNN 時，兩個句子 score 相差較大，使用 DNN 時候，兩個句子 score 相差較小。
這個結果跟預想差不多，因為 Bag of word 並沒有考慮詞彙的順序，他僅僅是考慮了詞彙在這個句子中的含量，也就是說即使意義不太相同的兩個句子，如果其中包含的詞彙相似，就會產生相似的結果。

RNN 則考慮了時間，在這個 case 就是詞彙的順序性，以這兩個句子而言，雖然”hot”都有出現，但一個在前一個在後，就會產生不同預測結果，所以可以看到 score 預測相差較多。

3. (1%) 請敘述你如何 improve performance（preprocess、embedding、架構等等），並解釋為何這些做法可以使模型進步，並列出準確率與 improve 前的差異。（semi supervised 的部分請在下題回答）

因為 model 架構的敘述已經在第一題描述了，這裡就簡單描述一下改 model 的過程

起初甚麼都沒調整的時候 Validation Accuracy 約為 80.2%

1	train acc	val acc	train loss	val loss	Kaggle	e
2	82.2528	80.19672	0.389179	0.421225		
3	81.42139	80.27966	0.403412	0.419568	0.80596	

因為我認為這個 model 容易 overfitting，於是我使用 ensemble，會 work 的原因是因為對於 overfitting 的 model，他的 variance 很大，bias 很小，所以利用很多 model 的結果去做平均，就能得到小 varaince 小 bias 的結果。

最開始我將三個 model 做 ensemble，每個 model 都是多層且 Bidirectional 的。

Validation Accuracy 上升到大約 81.12% Kaggle 的結果也上升大約 1%

	Train acc	Val acc	train loss	val loss	Kaggle	
21	83.97484	81.12726	0.348721	0.403702	0.81526	

我繼續加入新的 2 個 model 做 ensemble，結果來到 81.4% Kaggle 來到 81.8%

28	87.17758	81.4026	0.316081	0.402052	0.81825	

到了這個階段，由於訓練時間已經非常久了我就沒有繼續加入 model，然而為了給 model 更多資訊量我提高了 sentence length。

當我將 sentence_length 從 20 提高到 26 的時候，準確率又快速上升了約 1%。

32	92.30114	82.37792	0.339517	0.387107	**0.82766**	

在我所有調整的參數當中，我認為 sentence length 是影響最巨大的，我認為原因是因為假如 sentence length 不夠大的話，model 得到的資訊量不夠多，因此他本身 bias 還是太大，這樣做 ensemble 的效果就不好，因此提高 sentence length 能夠有效改善 performance。

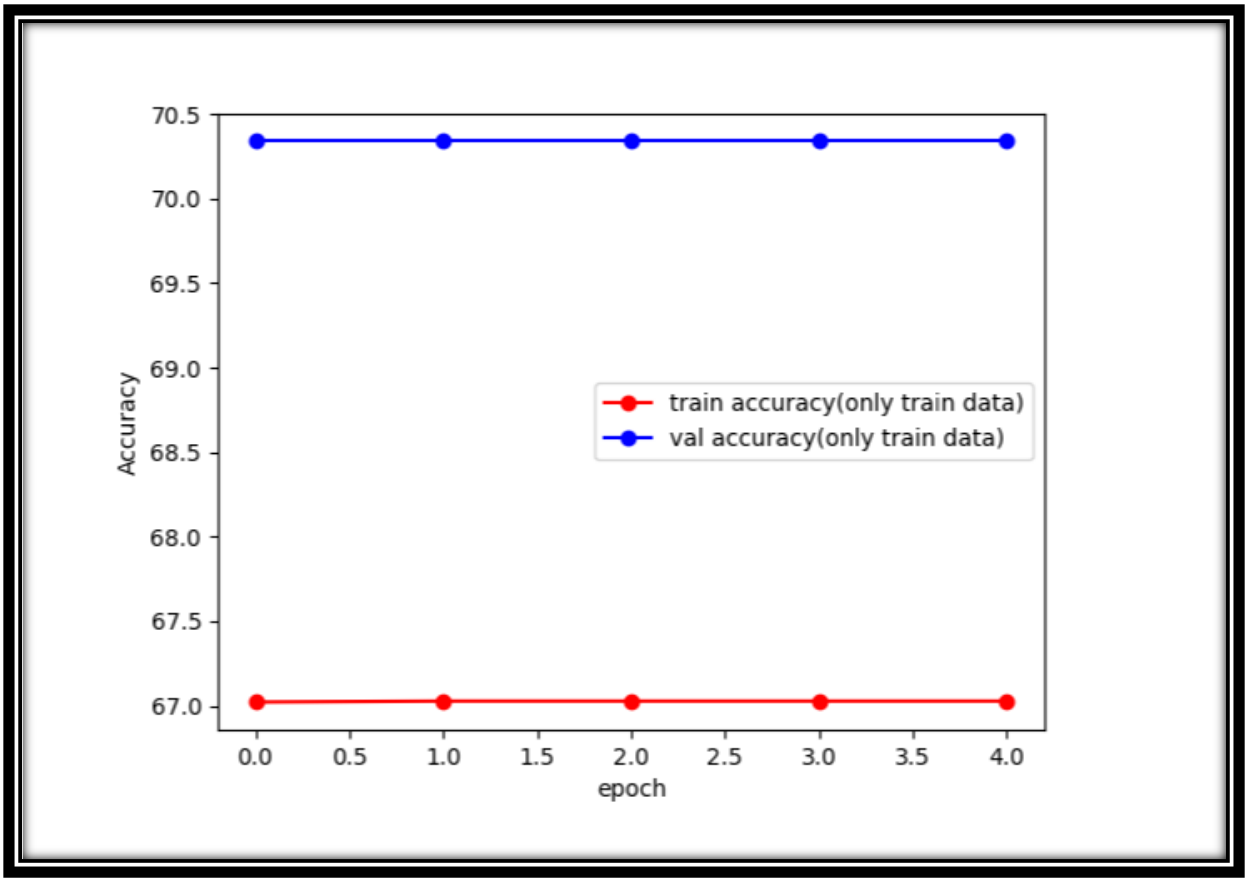
而 Ensemble 之後的結果也確保了提升 sentence length 之後的 performance，每個 model 的結果平均之後能夠得到比較小的 bias 以及比較小的 variance，因此 performance 才有不錯的表現。

4. (2%) 請描述你的 semi-supervised 方法是如何標記 label，並比較有無 semi-supervised training 對準確率的影響並試著探討原因（因為 semi-supervise learning 在 labeled training data 數量較少時，比較能夠發揮作用，所以在實作本題時，建議把有 label 的 training data 從 20 萬筆減少到 2 萬筆以下，在這樣的實驗設定下，比較容易觀察到 semi-supervise learning 所帶來的幫助）。

我將 train data 取了 2 萬筆 validation data 取了 18 萬筆，unlabeled data 則取用全部。
Threshold 為 0.8

我將這個實驗分成兩種
實驗一、

利用原本 model 預測 unlabeled data，然後結合 labeled data 跟 unlabeled data 再做一次 train，但是 train 的時候 word embedding 完全相同於只用 labeled data + test data 的 word embedding。



可以看到 train accuracy 跟 validation 幾乎沒有甚麼變化，而且呈現 under fitting 的狀態，原本在單純取用 labeled data 做 train 的時候，train accuracy 跟 validation accuracy 大概都落在約 78%左右，而使用了 semi-supervised learning 之後 train accuracy 掉到 67%左右，validation accuracy 則掉到約 70%。

```
[ Epoch1: 9365/9365 ] loss:0.597 acc:12.500
Train | Loss:0.63514 Acc: 67.022
Valid | Loss:0.61108 Acc: 70.339
saving model with acc 70.339
-----------------------------------------------
[ Epoch2: 9365/9365 ] loss:0.655 acc:10.938
Train | Loss:0.63422 Acc: 67.028
Valid | Loss:0.61013 Acc: 70.339
-----------------------------------------------
[ Epoch3: 9365/9365 ] loss:0.625 acc:11.719
Train | Loss:0.63400 Acc: 67.028
Valid | Loss:0.61083 Acc: 70.339
-----------------------------------------------
[ Epoch4: 9365/9365 ] loss:0.663 acc:10.938
Train | Loss:0.63394 Acc: 67.028
Valid | Loss:0.61021 Acc: 70.339
-----------------------------------------------
[ Epoch5: 9365/9365 ] loss:0.495 acc:14.844
Train | Loss:0.63385 Acc: 67.028
Valid | Loss:0.60998 Acc: 70.339
-----------------------------------------------
```

Semi-supervised learning

整體而言，performance 差了很多。

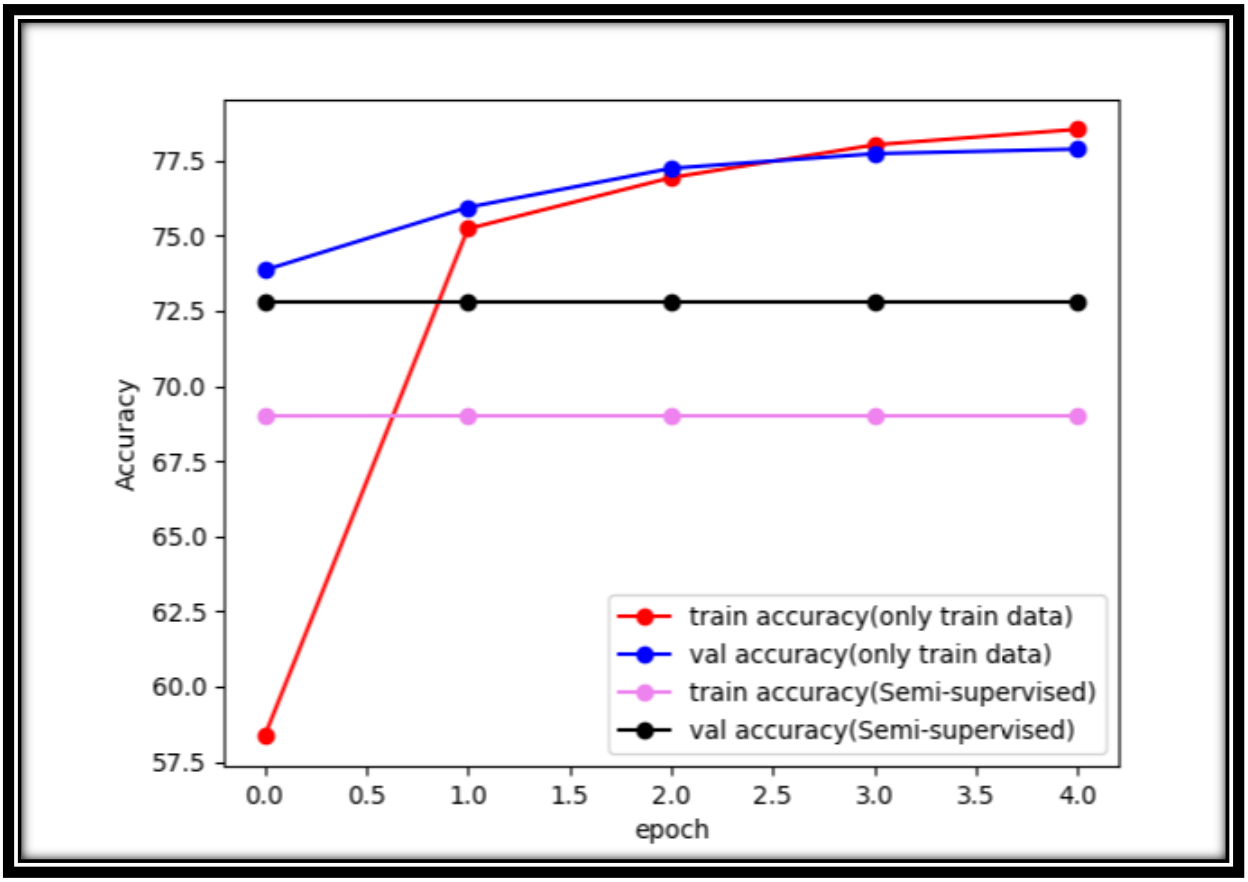
我認為造成 under fitting 的原因是因為 word embedding 裡面沒有使用到 unlabeled data 的 vector，所以在 train unlabeled data 的時候，每個句子中的 word index 很有可能都是 UNK 的狀態，這樣自然無法判斷詞彙的涵義以及詞彙之間的關係，使得整個 model 不足以預測出正確的結果

因此下個實驗我重新 train 一個 word embedding，並以此 embedding layer 再重複一次 self-training 的過程。

實驗二、重新做 word embedding(納入考慮 unlabeled data):

加入 unlabeled data 的 word embedding 後 word 數量翻倍。

```
get words #55777
total words: 55779
sentence count #1378614
```



紅線及藍色的線，代表那 2 萬筆 train data 的結果，紫色以及黑色的線，代表 self-training 的結果。

結果仍然是 under fitting，train accuracy 以及 validation accuracy 皆有上升了 2%，然而皆沒有隨著 epoch time 而穩定上升，我認為造成的原因是:

1.data 數量多了好幾倍，原本的 model 已經不足以預測正確的結果，所以 accuracy 無法隨著 epoch 跟著上升

後來我減少了 unlabeled data 拿來 train 的數目，發現此現象仍然存在，所以可能這個 model 剛好就是會卡在 local minimum 的地方。

```
[ Epoch1: 1563/1563 ] loss:0.594 acc:35.938
Train | Loss:0.58912 Acc: 72.662
Valid | Loss:0.58592 Acc: 72.818
saving model with acc 72.818
-----------------------------------------------
[ Epoch2: 1563/1563 ] loss:0.565 acc:37.500
Train | Loss:0.58757 Acc: 72.672
Valid | Loss:0.58554 Acc: 72.818
-----------------------------------------------
[ Epoch3: 1563/1563 ] loss:0.538 acc:39.062
Train | Loss:0.58738 Acc: 72.672
Valid | Loss:0.58494 Acc: 72.818
-----------------------------------------------
[ Epoch4: 1563/1563 ] loss:0.670 acc:31.250
Train | Loss:0.58715 Acc: 72.672
Valid | Loss:0.58644 Acc: 72.818
-----------------------------------------------
[ Epoch5: 1563/1563 ] loss:0.594 acc:35.938
Train | Loss:0.58651 Acc: 72.672
Valid | Loss:0.58581 Acc: 72.818
```

2. threshold 的挑選也會影響結果，要是結果因為 threshold 的挑選預測錯誤，而後又被重新帶進去 train，錯誤只會越來越大，所以也可能讓結果不理想

結論:

Semi-supervise learning 可以幫助在 train data 太少的狀況下，給予更多參考資料，然而要讓整個 model 能夠 train 的起來，要費更多苦心去維持預測 label 的品質，self-training 的理論很簡單，但也因此很難控制預測出的 label 的品質。