學號：B03901101 系級： 電機三 姓名：楊其昇

1. (1%)請比較有無 normalize(rating)的差別。並說明如何 normalize.

下表為針對 matrix factorization 不同 latent dimension 時，有無 Normalization 的簡單比較，而 matrix factorization 的實作方式均一樣：$r_{i,j} = U_i \cdot V_j + b_i^{user} + b_j^{movie}$，其中 U 和 V 的初始值為 random normal distribution，bias 的初始值則為 0，batch_size = 128, optimizer 和 loss 為 keras 本身提供的 adam 及 mse，訓練的 epoch 次數方面，均使用 keras 提供的 early stopping，其中 patience = 5，而 normalization 的方式為典型的 $\frac{rating-mean}{std}$，預測時再將預測結果根據 mean 及 std 還原回去：

Latent dimension	RMSE of public part on Kaggle	
	Normalized	No normalized
5	0.86606	0.87053
10	0.86636	0.86803
15	0.86882	0.87190
20	0.86513	0.87284

根據表中數據可以發現，經過 normalization 後的表現都比沒有經過 normalization 的要好，而在訓練過程中，由於 loss function 為 mean square error，所以經過 normalization 後的資料，不論是 training data 或 validation data，loss 均會比沒有 normalization 的版本低，而我想這也是滿合理的，因為 normalization 本身便縮小不同值之間的差距，所以 loss 自然比較低。

不過其實在助教的 sample code 出來之前，其實自己有先用 numpy 實作了 full batch 版本的 matrix factorization，learning rate 的設定方式是參考網路上查到的一篇論文（https://www.csie.ntu.edu.tw/~cjlin/papers/libmf/mf_adaptive_pakdd.pdf），為一種應用在 matrix factorization 的 adagrad 的改良版本(RPCS: Reduced Per-coordinate Schedule)，會隨著每次 gradient 值的大小調整 learning rate：

```
 7 def matrix_factorization(R, P, Q, K, Bu, Bi, steps=701, lr=0.1,beta=0.02):
 8     alpha_p = 0
 9     alpha_q = 0
10     alpha_bu = np.zeros((1,R.shape[1]))
11     alpha_bi = np.zeros((R.shape[0],1))
12     Q = Q.T
13     for _ in range(steps):
14         indice = np.where(R == 0)
15         #error = np.dot(P,Q) + bias -R
16         error = (np.dot(P,Q) + Bu + Bi) - R
17         error[indice] = 0
18         eva = np.sqrt((error**2).sum()/(R!=0).sum())
19         if (_%10 == 0):
20             print ('iteration:{}, loss:{}'.format(_,eva))
21         p_grad = 2*np.dot(error,Q.T)
22         q_grad = 2*np.dot(P.T,error)
23         alpha_p += np.diag(np.dot(p_grad,p_grad.T)).mean()
24         alpha_q += np.diag(np.dot(q_grad.T,q_grad)).mean()
25         alpha_bu += (2*error**2).mean(0)
26         alpha_bi += (2*error**2).mean(1).reshape(-1,1)
27         P -= lr/np.sqrt(alpha_p)*(p_grad-beta*P)
28         Q -= lr/np.sqrt(alpha_q)*(q_grad-beta*Q)
29         Bu -= lr/np.sqrt(alpha_bu.mean())*(2*error.mean(0)+beta*Bu)
30         Bi -= lr/np.sqrt(alpha_bi.mean())*(2*error.mean(1).reshape(-1,1)+beta*Bi)
31         if (eva < 0.075):
32             print ('itrration break:',_)
33             break;
34     return (P,Q.T,Bu,Bi)
```

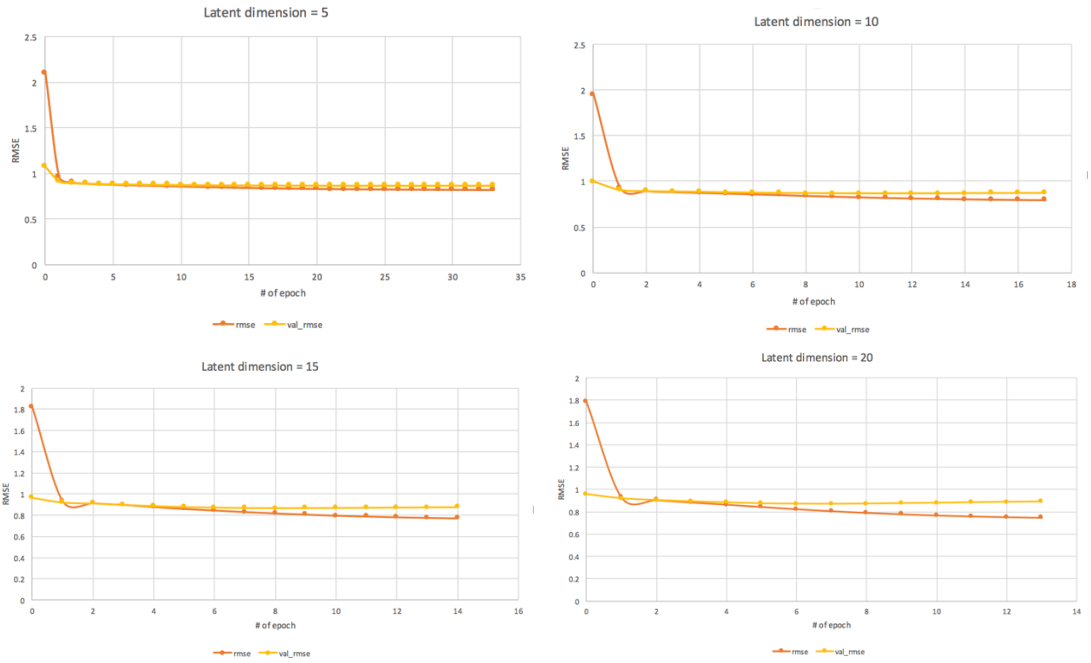
在最初的設定中，learning rate 為單純寫死的數字，為了要讓 loss 達到 global minimum，learning rate 要設很小（差不多 0.00001 左右），但若採取了如上圖的更新參數的方式，learning rate 便能一開始設大一點（可以到 0.1 左右），以加速收斂的速度。

然而在這次用 numpy 的實作中，在同樣參數的設定下（K=10，learning rate = 0.1，regularization (beta) = 0.02，iteration = 701，其餘參數均為 random normal distribution），沒有 normalization 的版本上傳到 Kaggle 後 public 的部分為 0.87792，而有 normalization 的版本則是 0.88820。而我想用

numpy 實作和用 keras 實作所得到的結果差異原因或許如下：
(1) 因為在 numpy 的版本中是使用 full batch 來更新參數，但 keras 卻有切 batch size，通常在更新參數時，經驗告訴我們做 full batch 有時候往往會發生一些意想不到的事情。
(2) keras 採用 adam 來當作更新參數的方式，而 adam 大概是目前所有方法中最通用的方式，或許 adam 的更新參數方式相較於 RPCS 更佳，畢竟 RPCS 最主要更新參數的速度仍來自 gradient 的大小，一旦經過 normalization 後，gradient 的值也會變得比較小，但 adam 除了考慮 gradient 外還有考慮 momentum 等因素，因此受 gradient 大小的影響比較不明顯。

2. (1%)比較不同的 latent dimension 的結果。

以下為不同 latent dimension 的訓練過程，除了 latent dimension 的大小有所不同外，其餘模型的建置均相同（$r_{i,j} = U_i \cdot V_j + b_i^{user} + b_j^{movie}$：U 和 V 的初始值為 random normal distribution，bias 的初始值則為 0，batch_size = 128, optimizer 和 loss 為 keras 本身提供的 adam 及 mse，訓練的 epoch 次數方面，均使用 keras 提供的 early stopping，其中 patience = 5，rating 未經過 normalization，validation data 皆相同）：

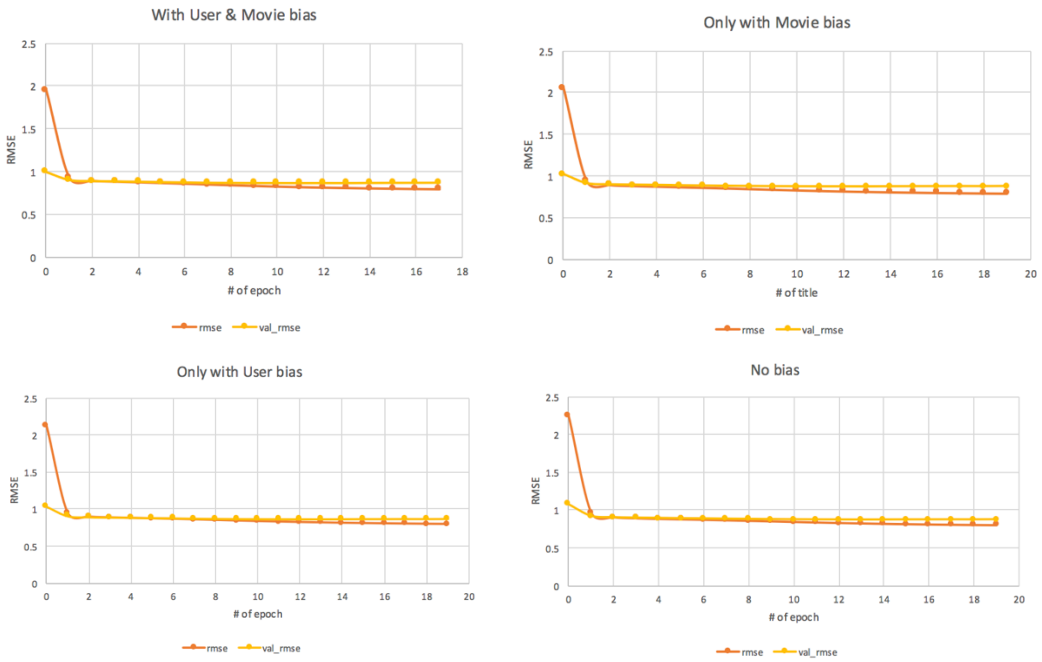


藉由觀察圖表，可以明顯發現 latent dimension 愈大，愈容易 overfitting，同時由於 early stopping 的關係，所以愈容易 overfitting 的模型，訓練的 epoch 數愈少，而這也符合以往的認知：參數過多，容易造成 overfitting。此外可以發現 matrix factorization 大部分在前幾個 epoch 後，validation data 的 root mean square error 便會進步至 0.9 附近，然後再來便是緩慢的下降，藉由第一題中的表格可以知道大概都會降到 0.86 ~ 0.88 左右，其中 k=10 的表現最佳。

3. (1%)比較有無 bias 的結果。

下圖為有無 bias 時，latent dimension = 10 的 matrix factorization 的訓練過程。若有 bias，初始值一律為 0，其餘條件皆相同（$r_{i,j} = U_i \cdot V_j + b_i^{user} + b_j^{movie}$：U 和 V 的初始值為 random normal distribution，batch_size = 128, optimizer 和 loss 為 keras 本身提供的 adam 及 mse，訓練的 epoch 次數方面，均使用 keras 提供的 early stopping，其中 patience = 5，rating 未經過 normalization，

validation data 皆相同）：



可以發現不論有無 bias，其影響並不至於太大。若再看詳細一點的數據（訓練過程中最後一個 epoch 的結果），表格如下：

Type	RMSE on training data	RMSE on validation data
Both	0.79381	0.87167
Only Movie bias	0.79387	0.87257
Only User bias	0.79672	0.87278
No bias	0.80142	0.87288

根據上表，可以發現在這次的實驗中，有加 bias 的表現會變得比較好一點，即 root mean square error (RMSE)會比較低，而其中在這次的實驗中，單純加 movie bias 的效果會比單純加 user bias 的效果要來得好。

4. (1%)請試著用 DNN 來解決這個問題，並且說明實做的方法(方法不限)。並比較 MF 和 NN 的結果，討論結果的差異。

在這次實驗中考量到 NN 模型的複雜度變化可以很大，因此決定只使用助教 sample code 中的模型架構，並將此 matrix factorization 視為 DNN 上的 Regression 問題，並以此和 latent dimension = 10 的 matrix factorization（有 bias，無 normalization）做比較。

DNN 的模型如下圖：

經過 embedding（latent dimension = 10）並且利用 keras 中函式使 user 和 movie 的 id vector 併在一

起後，便直接送入上圖的 DNN 模型中進行訓練，其中每層 DNN 的 activation function = relu，Dropout 比例均為 0.3，loss function 和 optimizer 和 matrix factorization 一樣為 mse 及 adam，訓練過程中的 training data 和 validation data 均和 matrix factorization 一樣。不過為了讓這次的實驗顯得比較有效，在實驗中有稍微調整 DNN 的參數盡量使得兩者的參數量差不多，兩者參數量如下：

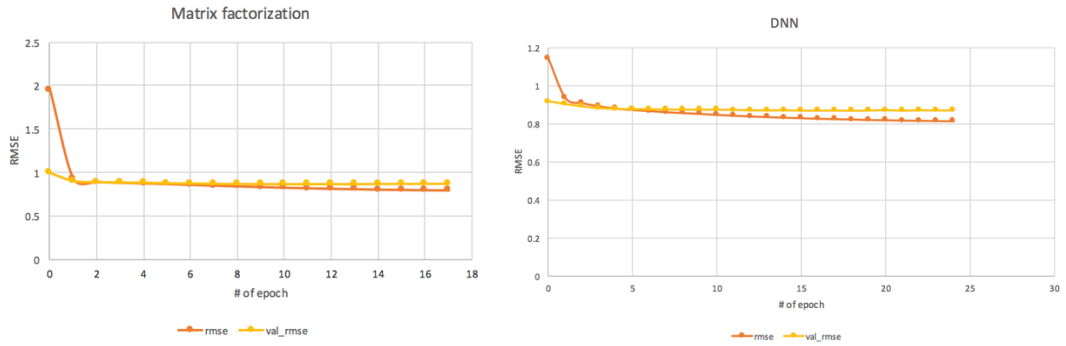
• matrix factorization（latent dimension=10, with movie & user bias）：

```
=================================================
Total params: 109,934.0
Trainable params: 109,934.0
Non-trainable params: 0.0
_________________________________________________
```

• DNN：

```
=================================================
Total params: 107,741.0
Trainable params: 107,441.0
Non-trainable params: 300.0
_________________________________________________
```

下面為兩者的訓練過程：



再看看兩者最後一個 epoch 的訓練結果：

Type	RMSE on training data	RMSE on validation data
Matrix factorization	0.79381	0.87167
DNN	0.81340	0.87004

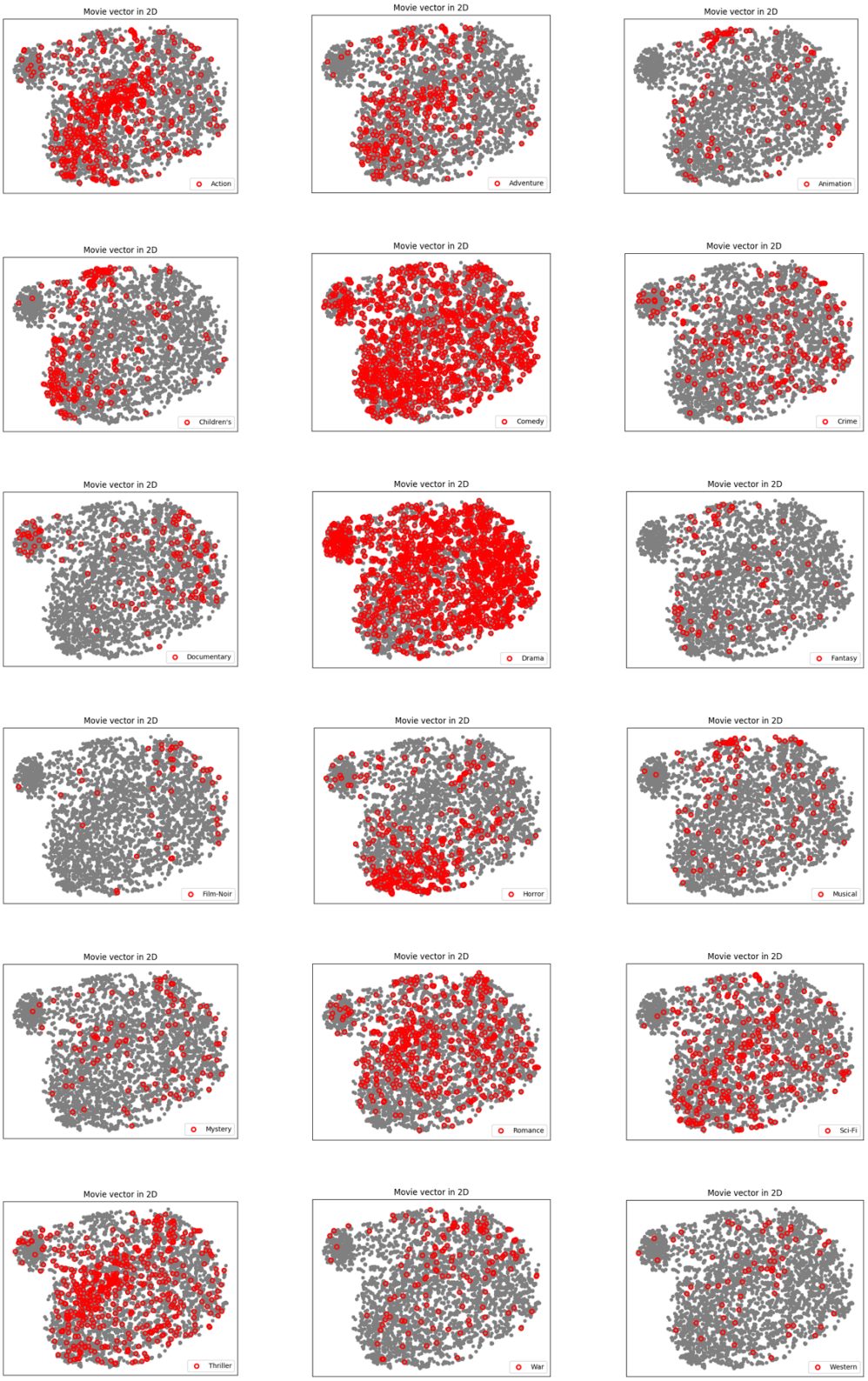
可以發現 DNN 最後訓練出來的表現雖然比 matrix factorization 要好，但其實並沒有太大的差異。在實驗中也能發現，DNN 在 RMSE 的下降幅度比較緩慢，而且或許是因為實驗中所用的 DNN 模型比較深，每一個 epoch 的訓練所需時間也明顯較 matrix factorization 長。此外在這次實驗中，matrix factorization 的 overfitting 狀況會略微嚴重一些，而原因可能有如下幾種：

(1) matrix factorization 的參數量比 DNN 的參數量略多。

(2) DNN 的模型架構中，每層最後都有 Dropout 來降低 overfitting 的門檻，但實驗中 matrix factorization 的架構並沒有加防止 overfitting 的手段（如 Dropout、Regularization 等）

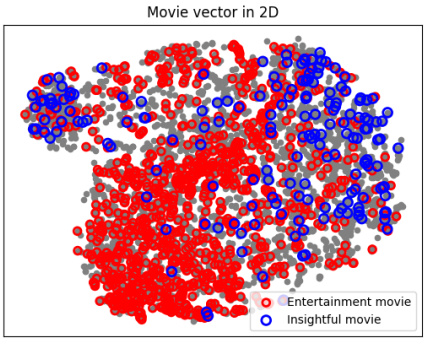
5. (1%)請試著將 movie 的 embedding 用 tsne 降維後，將 movie category 當作 label 來作圖。

以下作圖用的 movie embedding 是根據 latent dimension = 10 的 matrix factorization 而來。我一開始先按照自己一般的認知簡單畫了一下圖，結果卻整個爛掉，完全分不開。因此便決定，先按每一個種類各自畫圖（下列的圖只畫出在 movies.csv 裡面標有 label 的電影，可能會有多種 label 的情形），灰色為所有標記過的電影，用紅色圈起來的表示為目標種類：



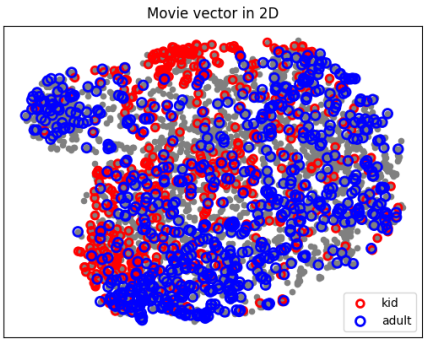
觀察完各個種類的分佈，其實不難發現為什麼一開始根據自己認知的分類會很難畫分開，因為其實就算單一種類的分佈往往也看起來很分散，而有的數目眾多的種類，如 Drama、Comedy 之類的幾乎更是涵蓋大半的電影，因此若要做出有區分效果的圖，勢必得在分類的群組上下一點巧思。

稍微經過觀察後，我發現在電影分佈中似乎娛樂性較高的電影（俗稱爽片）往往有集中在左半部的趨勢，而比較有深度的電影似乎都會在右半部，因此在排除掉數目過多且均勻分散的種類（如 Drama、Romance、Comedy、Sci-Fi、Thriller 等），將 Action、Adventure、Fantasy、 Horror 等四個種類歸在爽片類；Documentary、Film-Noir（黑色電影）歸在深度片類，並依此分類作圖，結果如下：



圖中紅點為爽片類，藍點為深度片，灰點為其他類。根據作圖結果，這樣的分類方式似乎還滿不錯的，基本上都可以大約地看到這兩類分布的區域大概便是圖中的左下和右上角，至於左上角有兩類混在一起的情況，推測應該是電影中沒被訓練到的種類，因此便呈現一開始的初始隨機狀態（左上推測沒被訓練到的資料群在圖中相較於其他有被訓練到的資料其實也有獨自成群的現象）。

除了利用爽片和深度片的方式來進行分類外，也嘗試利用別的分類方式來進行分類，例如大人和小孩喜歡看的電影種類雖然會有重複的部分，但其實基本上還是會有一些種類比較特殊的電影是大部分只有小孩會喜歡看（小孩市場導向）或者只有大人才會看（大人市場導向）的電影，因此依然按照前述的方式一樣剔除數目過多以及分佈過於平均的種類並將電影 label 為 Children’ s、Animation、Musical、Adventure 歸到小孩市場導向的電影；將 Documentary、Film-Noir、Crime、Horror 歸為大人市場導向的電影，作圖結果如下：



途中紅色為小孩市場導向類電影，藍色為大人導向電影。扣除左上角那群沒被訓練的電影種類，其實基本上仍可以觀察到圖中左上部大概為小孩市場導向的電影，右下為大人市場導向的電影。雖然這樣依照自己認為的市場導向來將電影做分類或許有稍嫌武斷的嫌疑，不過就結果而言其實分類效果其實還算能夠接受。

6. (BONUS)(1%)試著使用除了 rating 以外的 feature, 並說明你的作法和結果，結果

好壞不會影響評分。

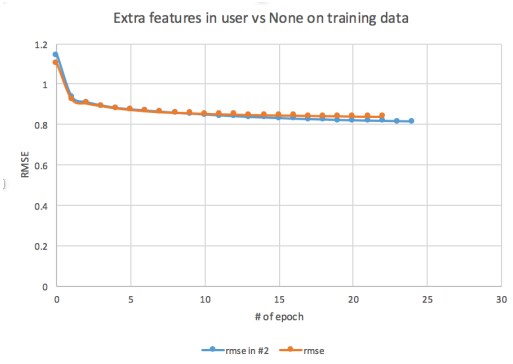
這本題中我的做法是將 users.csv 中的 Gender、Age、Occupation 等資訊加入 embedding layer 中，如下圖：

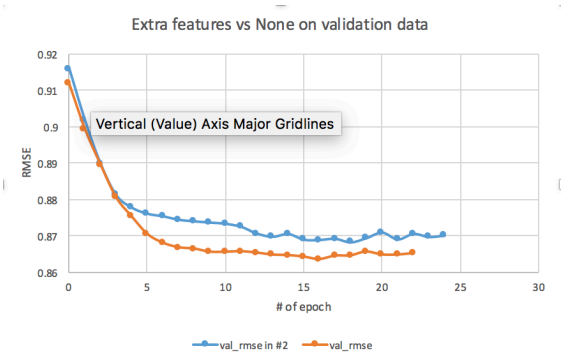
```
user = np.concatenate((user,user_info[ind,1].reshape(-1,1)),axis=1) # gender
user = np.concatenate((user,user_info[ind,2].reshape(-1,1)),axis=1) # age
user = np.concatenate((user,user_info[ind,3].reshape(-1,1)),axis=1) # occupation
```

```
def nn_model(n_users, n_items,latent_dim=10):
    user_input = Input(shape=[5])
    user_vec = Embedding(n_users, latent_dim, embeddings_initializer='random_normal',
                         embeddings_regularizer=l2(0.000001))(user_input)
    user_vec = Flatten()(user_vec)

    hidden = Dense(256, activation='relu')(user_vec)
    hidden = BatchNormalization()(hidden)
    hidden = Dropout(0.3)(hidden)
    hidden = Dense(128, activation='relu')(hidden)
    hidden = BatchNormalization()(hidden)
    hidden = Dropout(0.3)(hidden)
    hidden = Dense(64, activation='relu')(hidden)
    hidden = BatchNormalization()(hidden)
    hidden = Dropout(0.3)(hidden)
    output = Dense(1)(hidden)
    model  = keras.models.Model(user_input, output)
    model.compile(loss= rmse, optimizer='adam',metrics=[rmse])
    model.summary()
    return model
```

並以和第二題同樣的 DNN 模型（所有參數設定皆一樣）下去做訓練（embedding 的 latent dimension = 10），以下為訓練過程並將其和第二題做比較：





在 training data 上的訓練表現，有沒有加入 user 的其他資訊似乎影響不大，不過在 validation data 上就看得出區別了，有加入 user 其他資訊的模型表現明顯較未加來的好，雖然兩者的 root mean square error 差距其實不大（< 0.01）。藉由觀察兩個模型在 validation data 上的表現，也可以發現兩個模型一開始其實 RMSE 下降的速度是差不多的，而我想這也不讓人意外，因為基本上這兩者的是相同的模型，差別只在於 input 資訊量的多寡，並沒有因為 input 資訊量的改變而使得模型本身的能力變強，基本上加入其他 user 資訊的模型表現得比較好，我想單純就是獲得有用的資訊比較多而已。