HW2 – Sequence Models

This assignment was done in pair:
Udita Gupta - ung200
Ameya Shanbhag – avs431

Code Specific Details:

We used Google's *word2vec* pre-trained embeddings in our embedding layer as that gave us the best results. The link can be found below.
https://code.google.com/archive/p/word2vec/

We had tried *GloVe* as well but did not get that good results.

To be able to use the pre-trained embedding we had to load the Google .bin file and process it. The file is of 3.64GB and we were running different combinations on Google collab.
So we decided to generate the embedding_matrix on our laptops and save the numpy file onto disk so that in Google collab we can directly just import the embedding_matrix.

Below is the code that we used to generate the embedding_matrix:

```
import gensim
from gensim.models import Word2Vec
from gensim.utils import simple_preprocess

from gensim.models.keyedvectors import KeyedVectors

word_vectors = KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin',
    binary=True)
NUM_WORDS = 10000
EMBEDDING_DIM=300
vocabulary_size=min(len(word_index)+1,NUM_WORDS)
embedding_matrix = np.zeros((vocabulary_size, EMBEDDING_DIM))

for word, i in word_index.items():
    if i>=NUM_WORDS:
        continue
    try:
        embedding_vector = word_vectors[word]
        embedding_matrix[i] = embedding_vector
    except KeyError:
        embedding_matrix[i]=np.random.normal(0,np.sqrt(0.25),EMBEDDING_DIM)

del(word_vectors)
np.savetxt('embedding_matrix_google_word2vec.out', embedding_matrix)
```

So, in our .ipynb collab notebook you will see the matrix being imported from a file called:
embedding_matrix_google_word2vec.out
We have attached the .out file in our zip submission.

Final Architecture:

First we used pre-trained word2vec for our Embedding layer. Max_input length was 500.

```
NUM_WORDS = 10000
EMBEDDING_DIM=300
vocabulary_size=min(len(word_index)+1,NUM_WORDS)

model.add(keras.layers.Embedding(vocabulary_size,EMBEDDING_DIM,
weights=[embedding_matrix],trainable=True))
```

Then we added 3 CNN layers followed by Max Pooling for each of them:

```
# convLayer1
model.add(keras.layers.Conv1D(filters=32, kernel_size=4, padding='same', activation='relu'))
model.add(keras.layers.MaxPooling1D(pool_size=2))

# convLayer2
model.add(keras.layers.Conv1D(filters=64, kernel_size=5, padding='same', activation='relu'))
model.add(keras.layers.MaxPooling1D(pool_size=4))

# convLayer3
model.add(keras.layers.Conv1D(filters=128, kernel_size=6, padding='same', activation='relu'))
model.add(keras.layers.MaxPooling1D(pool_size=8))
```

Then we added a LSTM layer:

```
model.add(keras.layers.LSTM(64, dropout = 0.2))
```

Followed by a Dense layer:

```
model.add(keras.layers.Dense(32))
```

Followed by a final Dense layer:

```
model.add(keras.layers.Dense(1, activation='sigmoid'))
```

Loss Function Used: **Binary Cross Entopy**

Optimizer : **RmsProp**

We received a Final Test Accuracy of 90%

Trials and Errors:

Initially we started with a single Embedding, LSTM layer followed by a Dense layer and we tried various different input max length, LSTM units etc.
We then read about how CNNs can be used as the initial layer to extract features from Natural Language as well followed by LSTM layers.
So, we decided to experiment with quite a few combinations and started visualizing using Tensorboard.

We also tried those combinations with GRU, and different optimizers like adam, rmsprop.

We also tried rmsprop with lr = 0.01 and decay = 0.001 but still we were not getting better results.

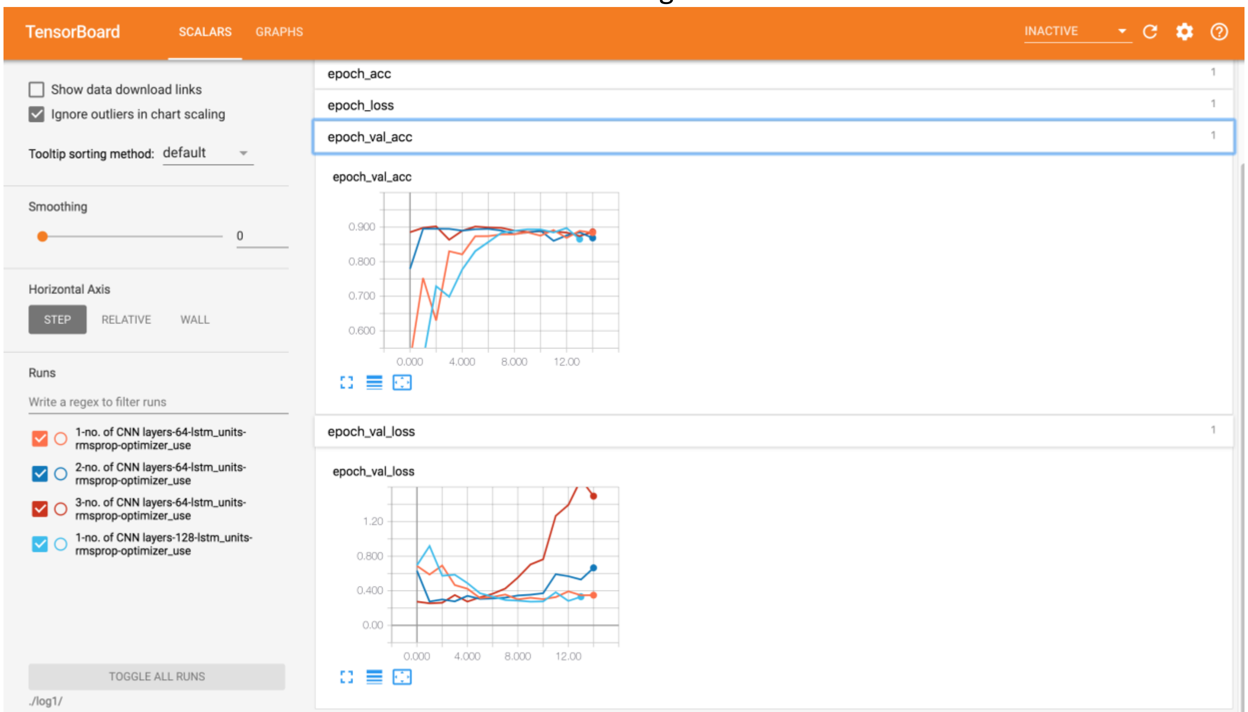
Best was to keep lr as default 0.001 and use 3 CNNs followed by 1 LSTM and 2 Dense.
Also, LSTM is pretty slow but when combined with CNN layers, it runs much faster!

We were also getting descent results using Bi-directional LSTM but there’s more scope for trying out different number of units etc with it.
We read that Bi-directional LSTMs are the new baseline architecture. As it can use past and future words both to get better context.

This is just a small code snippet of what we were trying to achieve.

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Embedding,LSTM,Conv1D,Dense,Dropout,MaxPooling1D, Flatten, GRU

dense_layers = [0]
layer_sizes = [32, 64, 128]
conv_layers = [1, 2, 3]
lstm_sizes = [256,300]
lstm_layers = [1,2]
optimizer_used = ['adam']

for optimizer_use in optimizer_used:

  for dense_layer in dense_layers:

    for lstm_size in lstm_sizes:

      for lstm_layer in lstm_layers:

        for layer_size in layer_sizes:

          for conv_layer in conv_layers:

            NAME = "{}-conv-{}-nodes-{}-gru_layer-{}-gru_size-{}-dense-{}-optimizer_use".format(conv_layer, layer_siz

            model = Sequential()

            model.add(keras.layers.Embedding(10000, 50, input_length=300))

            model.add(Conv1D(filters = layer_size, kernel_size = 3, padding='same', activation='relu'))
            model.add(MaxPooling1D(pool_size= 2))

            for _ in range(conv_layer-1):
              model.add(Conv1D(filters = layer_size, kernel_size = 3, padding='same', activation='relu'))
              model.add(MaxPooling1D(pool_size= 2))

            model.add(GRU(lstm_size,dropout = 0.2, return_sequences = True))


            for l in range(lstm_layer-1):
```

This is screenshot of all these combinations running from Tensorboard:



It was still running when we took this screenshot so all the combinations weren't done.

Loss Function:

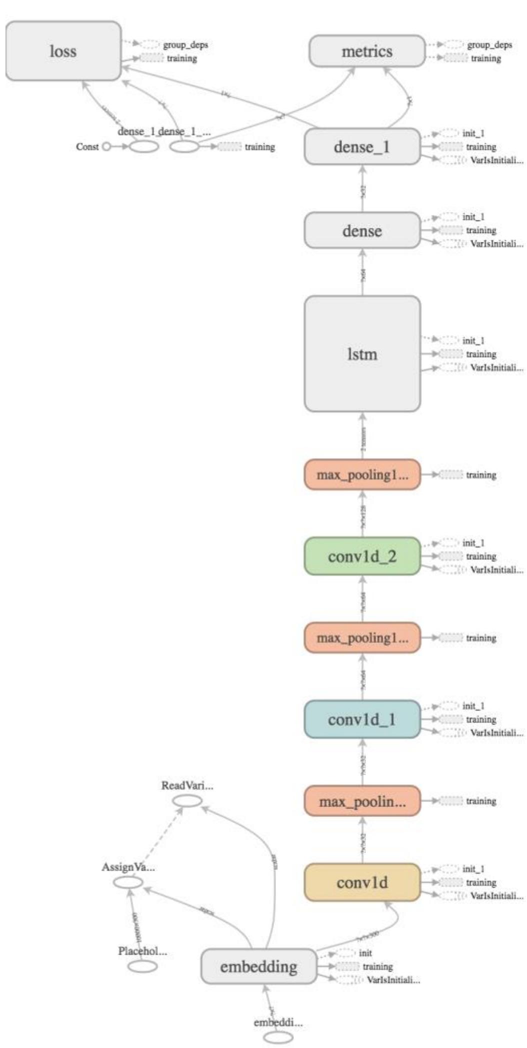
Final Loss Function that we used is: **binary_crossentropy**

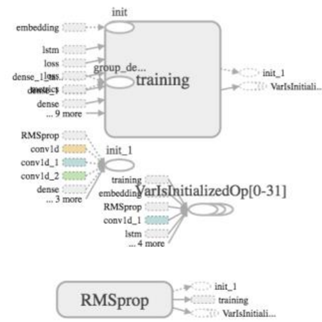
Optimizer:

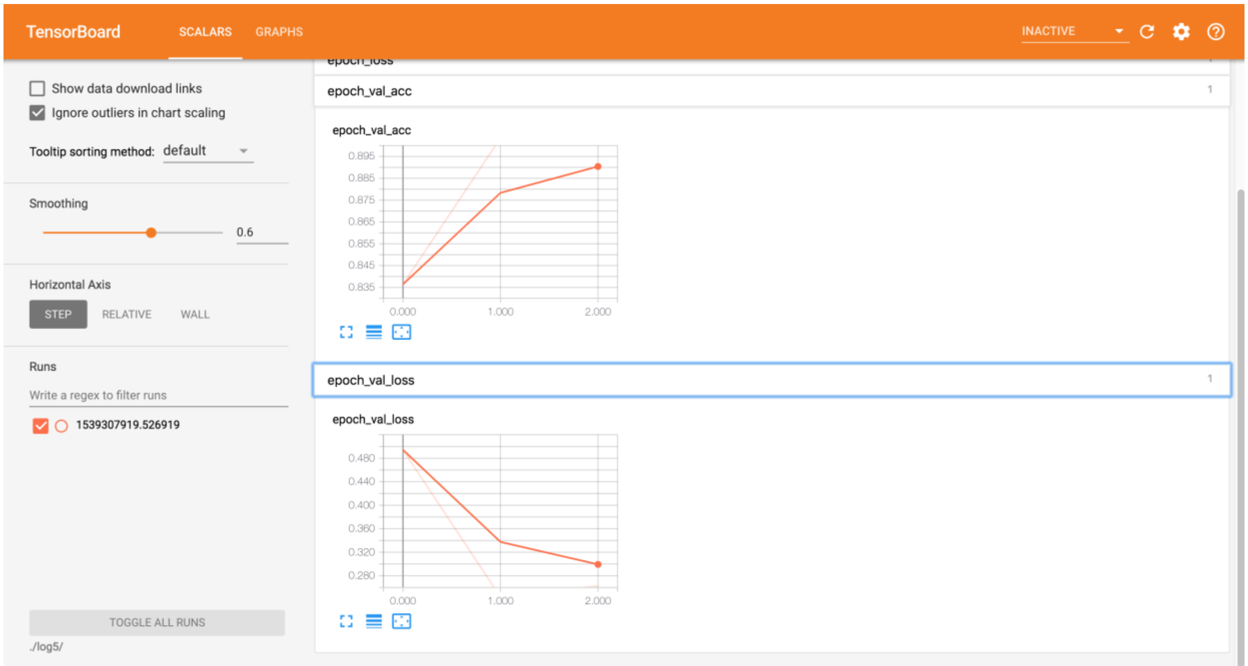
Final Optimizer we used is: **rmsprop**

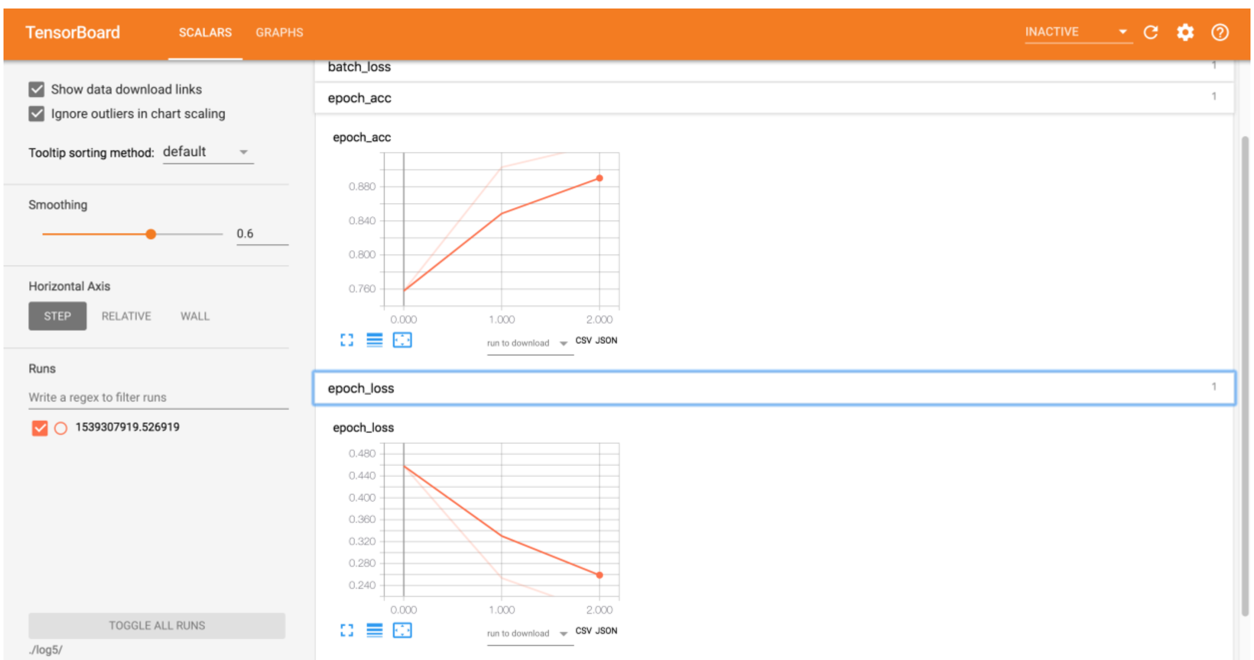
Tensorboard Screenshots:

These are from the final architecture: 3 epochs



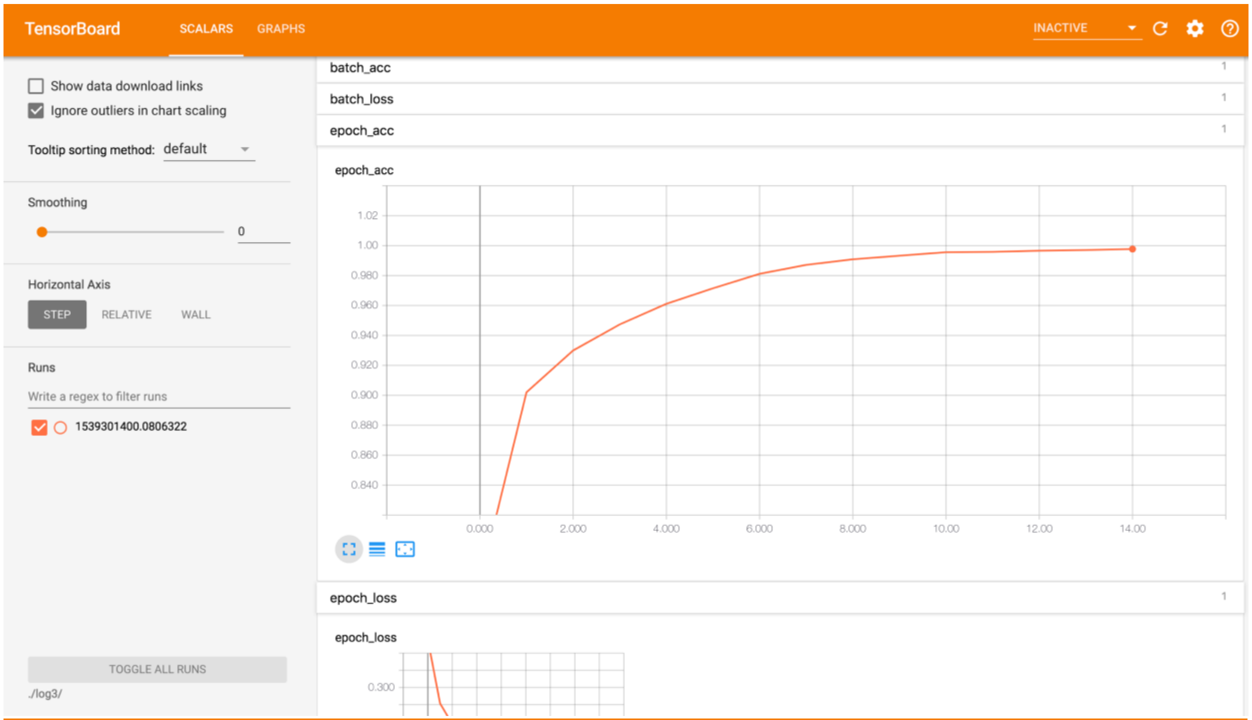


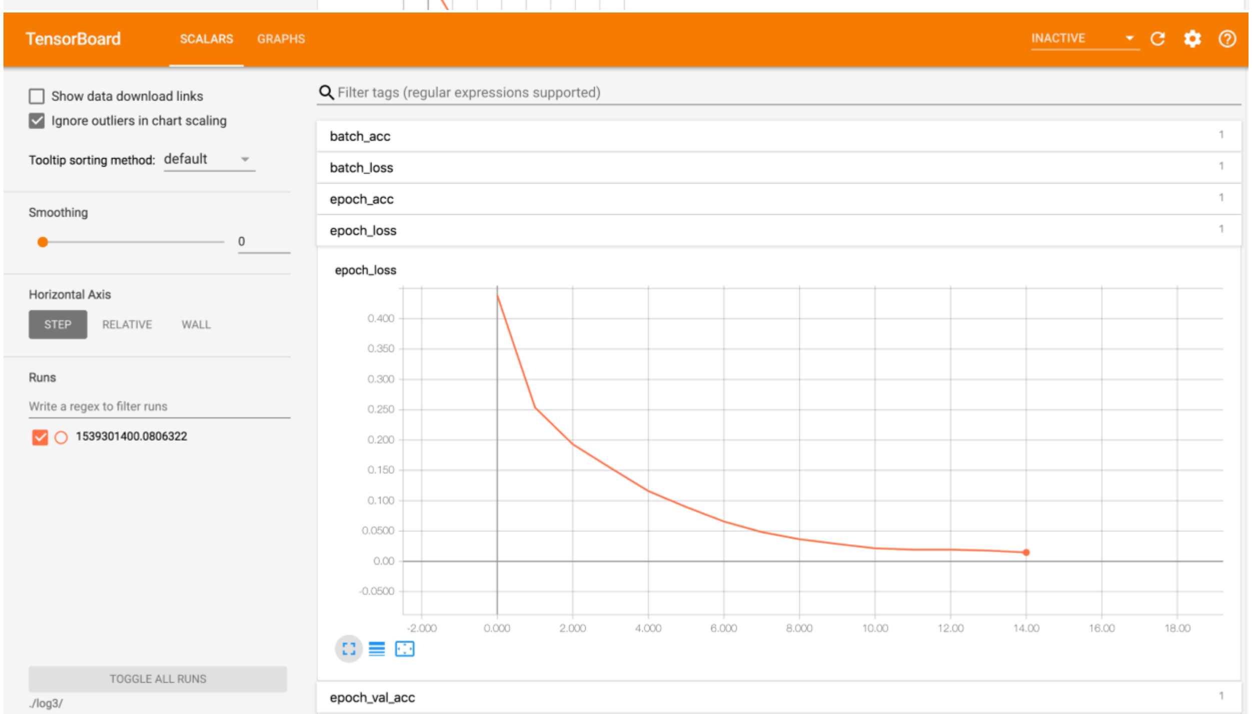


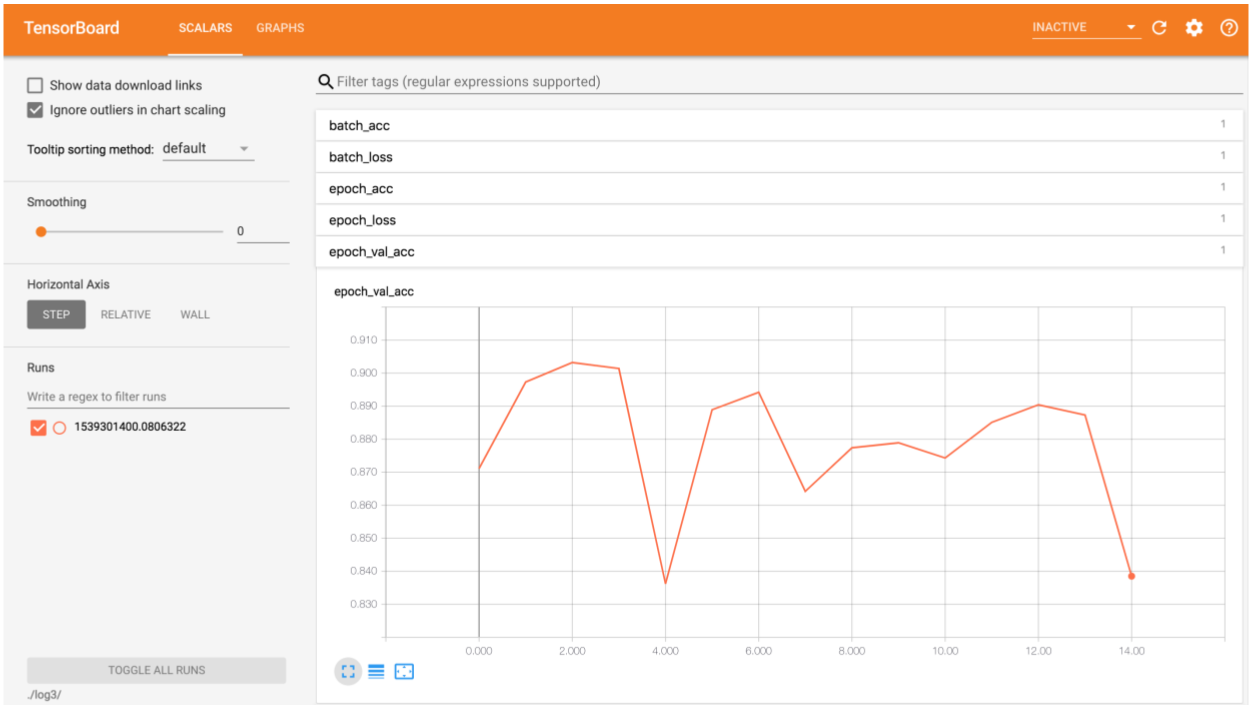


The reason why we chose 3 epochs as the sweet spot:

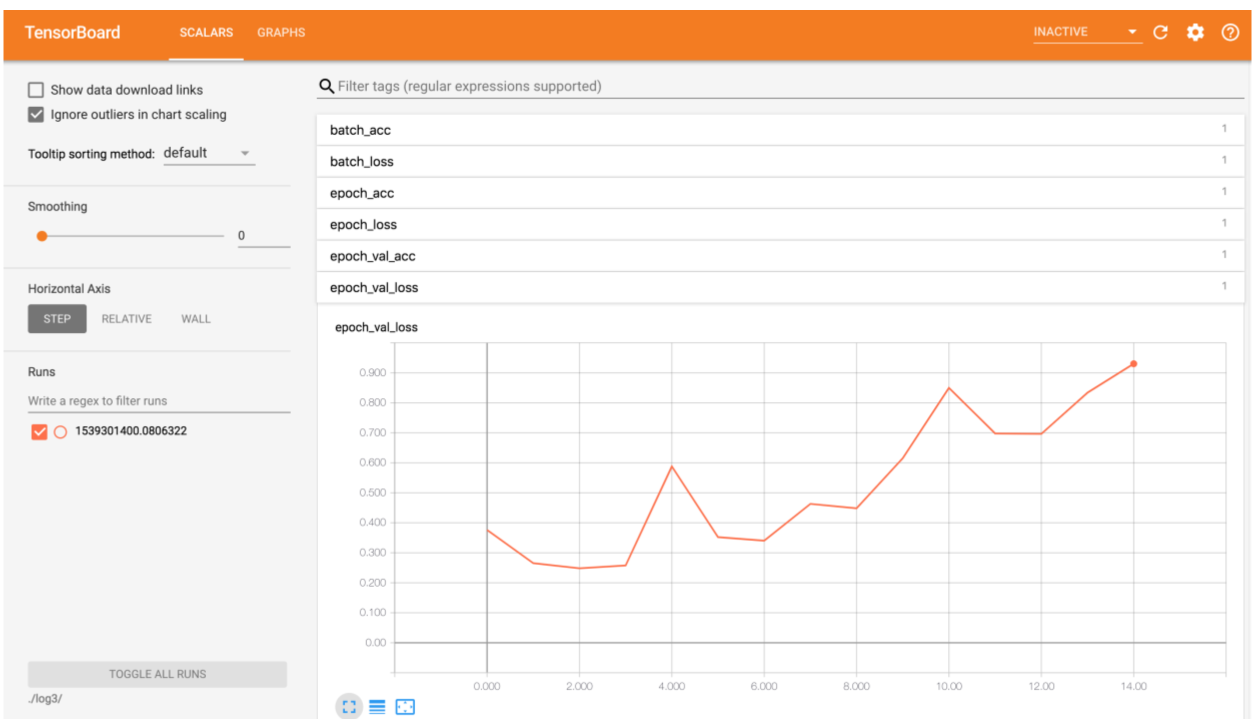
We ran this for 15 epochs to understand and get a good idea of from where it starts overfitting:


TensorBoard
SCALARS
GRAPHS
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
1539301400.0806322
TOGGLE ALL RUNS
./log3/
batch_acc
batch_loss
epoch_acc
epoch_acc
1.02
1.00
0.980
0.960
0.940
0.920
0.900
0.880
0.860
0.840
0.000
2.000
4.000
6.000
8.000
10.00
12.00
14.00
epoch_loss
epoch_loss
0.300


TensorBoard
SCALARS
GRAPHS
INACTIVE
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing
0
Horizontal Axis
STEP
RELATIVE
WALL
Runs
Write a regex to filter runs
1539301400.0806322
TOGGLE ALL RUNS
./log3/
Filter tags (regular expressions supported)
batch_acc
batch_loss
epoch_acc
epoch_loss
epoch_loss
0.400
0.350
0.300
0.250
0.200
0.150
0.100
0.0500
0.00
-0.0500
-2.000
0.000
2.000
4.000
6.000
8.000
10.00
12.00
14.00
16.00
18.00
epoch_val_acc



Val accuracy peaks at epoch 3



val loss is least at epoch 3 and is going hand in hand with training till there and then we can see overfitting starts.
Hence we decided to go till epoch 3.