Tyler Teufel - Assignment 3 Report

Part 1

Output Metrics from the two models, and the optimized LSTM

	Model	Test Loss	Test Accuracy	Precision	Recall	F1 Score
0	RNN	0.316913	0.215	0.53	0.215447	0.306358
1	LSTM	0.315839	0.215	0.53	0.215447	0.306358
2	LSTM (Title + Description)	0.314672	0.215	0.53	0.215447	0.306358

The outputs shown in the above table tells the story of how small, if at all present, the differences betweeen the two models truly are. If looked at closely, you could indeed conclude that RNN does ever so slightly outperform LSTM in terms of test loss in both cases, with LSTM becoming an increasingly worse perfomer with the addition of the Title into consideration with the description.

The main process of this part was to break each movie description down within the dataset to the token words, removing all stop words through the cleaning of the data. I immediately dropped all columns not under consideration, and cleaned the description data, then converting the text tokens to a sequence of ints. After crreating a dictionary of these words, I padded the data and then proceeded to load in the GloVe data, and preparing the embedding matrix. The first step was to implement multilabeling, since movies could be related to more than one genre. From there I split the data as instructed, and then proceeded to prepare the embedding matrix based upon the 400,000 word vectors unpacked from glove.6B.100d.txt file.

From here, I built the models, first creating the RNN using the embedding matrix weights, and the sigmoid activation function. After training that model, I moved on to training the LSTM model. After completing both of those, I went back and combined the title and descriptions of the data entries and retrained the LSTM model, capturing all of the metric data. After plotting and collecting the data, I stored it all in model_evaluation.csv, and found that, overall, as mentioned earlier, the results were not significantly different.

Difference in Test Loss

The difference in test lost mentioned above could also lead us to believe that with an increase in epochs, and possibly an altered learning rate, that the test loss gap could continue to potentially grow, leading to similar results amongst the other metrics, since there current differences are so minute that you truly cannot tell from the leading decimal places.

Part 2

In regards to this implementation, the preprocessing steps proved to be pretty straightforward, with my strategy consisting of parsing the directory of the dataset download, and resizing each image to 128x128 before storing them and their labels in np arrays to be utilized for training. I put all of this logic into the load_images function, which would take in one parameter consisting of the path to that folder directory. The way I went about traversing all of the individual directories, was by making the desired input parameter the

parent directory for either the training data or the test / validation data, and then indexing them based upon the existing child directories. This allowed for a singular call to the function when getting setup.

The next step was to normalize the image pixel values using one-hot encoding, througth scaling the image pixel values to a smaller range, typically between 0 and 1. Since Neural networks tend to work best with smaller input value range, I divided by 255.0 to get them in the range [0,1]. The purpose of the one-hot encoding, was to represent the class labels as binary vectors, with a 1 at an index indicating a likelihood of the image being a member of that class.

For training the models, I first setup both neural networks in similar fashions, with the 6 convolution layer model simply having each layer doubled. I doubled the batch size each layer deeper into the neural network, and ran a few different iterations with different optimizer functions, learning rates, and batch sizes for the actual fitting. Regardless, in all test instances I ran, as displayed in the 'cnn_results.csv' file, the 3 convolution layer model performed better across the board.

The coolest part of this entire project was getting to output image by image test cases. Getting to see the images actually be predicted accurately in most cases was fairly mind blowing. The way I went about this was simply randomly selecting two images from the range of indices I identified earlier, and pulling their labels as well. From here I would simply call the predict function on each model and run compare the predicted value to the labeled value for each image.

Overall, it was very interesting to see how they performed over the course of the different iteratons.