Brandon Edmunds
CSE582
Dr. Yin
04/09/23

Homework 2

Introduction

Sentiment classification is done using a convolutional neural network and a long short-term memory neural network. Yelp restaurant reviews are used for the data, with the sentiments derived from the reviews. The resulting performance of the neural networks are compared and analyzed. The network implementations are also analyzed.

Preprocessing

The first step was reading the data. The data was read line by line to make sure that there was enough memory to read all the data. Only the first 100,000 entries of the Yelp reviews are considered. The data is transformed into JSON data and is then read into the program. The original data did not have a sentiment column, so a sentiment column was created using the stars column. Reviews with stars less than or equal to two are considered negative, reviews with stars equal to three are considered neutral, and reviews with more than three stars are considered positive. The 100,000 sample points are further reduced to 30,000 data points, using 10,000 data points from each sentiment category, allowing for uniformly distributed data. Next, the reviews are tokenized and any accents on the letters are removed. Next, the tokens are stemmed. The stemmed reviews are then used to create embeddings using word2vec. Next, reviews are all made the same length by adding pad tokens to the end of shorter reviews. Using word2vec, each of the stemmed tokens is converted into a number which is used to fetch a word's embedding. The three different sentiment types are similarly converted to numbers.

Convolutional Neural Network Implementation Analysis

The convolutional neural network (CNN) has multiple steps. In initializing the convolutional neural network, first, the word2vec model is used to initialize the word embeddings. The pad token is frozen so that it does not get updated while training, allowing for it to remain a pad token. The CNN used a number of filters, where 10 performed reasonably well given bounded epochs and runtime. In the provided example, four different convolutional layers were used where each convolutional layer has a different kernel size. The four kernel heights that were used were 1, 2, 3, and 5. The widths of the kernels are all set to the embedding size. The differing heights represent how many words to consider out of a review. In each of the convolutional layers, the padding was set to the height of the kernel minus one in the height direction, and a zero in the width direction, allowing the kernels to consider one word at a time, followed by two words at a time, up to the height of the kernel (assuming a sufficiently large review). In the forward pass, after getting the embeddings for each word, each of the four convolutional layers are given the embeddings, then each of the outputs of the four convolutional layers are given to an activation function. Each of the outputs of the activation functions are then shrunk using max pooling. In the provided example, the max pooling is done over all of the words in the review. All of the outputs of the max pooling layers are finally concatenated together, then the resulting tensor is flattened and given to the fully connected layer which outputs values for each of the three sentiments (note that the softmax layer is unnecessary since PyTorch's cross entropy loss is being used). The whole process is done for each training example in the batch.

Long Short-Term Memory Neural Network Implementation Analysis

Similarly, the long short-term memory neural network (LSTMNN) uses the same inputs as the CNN. Also, the LSTMNN defines the embeddings in the same way. In the forward pass of the LSTMNN, the embeddings for each word in each review in each batch are again found. The embeddings are then given to the LSTM layer. Different numbers of LSTM layers were tried, where increasing the number of LSTM layers increased the running time without much change in performance. The hidden states of the final LSTM layer are then flatted and given to the fully connected layer, which outputs values for each of the three sentiments.

Experimentation

When giving 676,043 parameters to CNN, CNN took approximately 3.5 seconds per epoch. When giving 774,495 parameters to LSTMNN, LSTMNN took roughly 6 seconds per epoch, indicating that LSTMNN is slower for a similar number of parameters. Also, in analyzing the number of parameters in a single convolutional and LSTM layer, a single convolutional layer was seen to have a number of parameters on the order of 10^2, and a single LSTM layer was seen to have a number of parameters on the order of 10^3, emphasizing the larger size of LSTM layers. The main hyperparameters considered include using an Adam optimizer, cross entropy loss, a learning rate of 0.01, a batch size of 32, an embedding size of 25, and a hidden size of 32 for LSTM. Different hyperparameters were experimented with, including different optimizers such as stochastic gradient descent, different numbers of filters for CNN, different filter sizes, and different numbers of convolutional and LSTM layers, in addition to varying hyperparameters such as learning rate. Also, different activation functions were tried for CNN (note that LSTM for PyTorch does not allow for changing internal activation functions without degrading GPU performance), including ReLU, leaky ReLU, ReLU6, tanh, and sigmoid, with ReLU6 having a slight edge. Another note is that the LSTMNN was more prone to overfitting than the CNN. Also, bidirectional LSTMs were considered, yielding additional overfitting while increasing training time. When adding dropout to the LSTM layer, the overfitting decreased. As the goal of the assignment is not to obtain maximal accuracy, the number of epochs was fixed to 10 to ensure a fast runtime. Finally, the CNN consistently got around 70% accuracy and the LSTMNN consistently got around 70% accuracy for a train-test split of 70%-30% with the varied hyperparameters.

Conclusion

Both the LSTMNN and CNN models performed similarly, though the LSTMNN model was more expensive. On the other hand, the LSTMNN model was more straightforward to implement, as specifications for components such as kernel sizes did not need to be considered. The LSTMNN model was also able to overfit more easily than the CNN model, indicating that the LSTMNN model may perform better for increasingly complex learning tasks.