Text Classification Project

Amol Sharma

6th February 2019

This project explored 2-class sentiment analysis. The Stanford Sentiment Treebank [4] was used for the task of determining whether movie reviews were positive or negative. This task was accomplished with simple RNNs, then with LSTMs for comparison, and then with pre-trained word embeddings in the LSTM.

1 Text Classification with RNNs

First, the tutorial from *AllenNLP* [1] was referenced to make a neural text-classifier model. The simple model receives sentences as tokens and 2-class labels for each of those sentences from a dataset reader. It then uses a TextFieldEmbedder to get an embedding of dimension 300 (from given pretrained embeddings) for each of those tokens. It then encodes the sentence and attempts to classify it using a feed forward neural network. Once classified, we check to see against the given label from the data and update our accuracy. Over time, we try to maximize this accuracy.

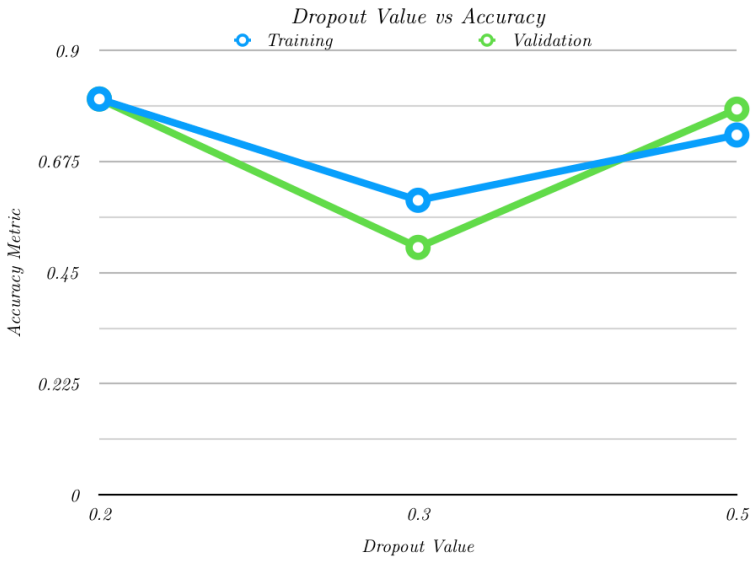
To decide from the several hyper-parameters, I first tested the parameters from the tutorial as a baseline. I noticed that the accuracy over epochs seemed to be increasing at first, and then decreasing. I thought that this may have been due to taking large steps on the gradient, and added gradient normalization in the config, which drastically improved overall accuracy. Since I had already changed a hyper-parameter, I opted to modify all of them to find the best configuration instead of testing one.

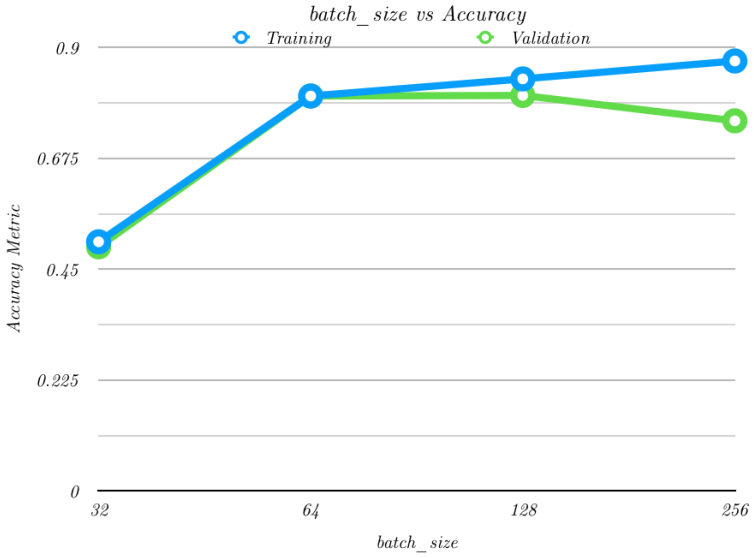
I first modified the nonlinearity of the first layer of the feed forward network. Among 'relu', 'leakyRelu', 'tanh', 'softplus' and 'sigmoid', tanh turned out to be the most accurate at around 0.8 accuracy. In those experiments, the best epoch was always under 15 (usually 6 - 10), so I decided that number of training epochs and patience was reasonable although this may depend on optimizer. For the optimizer, 'dense_sparse_adam' seemed to have the best accuracy, although 'adamax' was pretty close. Adamax had a higher training set accuracy but performed worse on the validation data. I prioritized held out data and used dense_sparse_adam. For both dropout and batch_size, there was a lot of variation and I decided to tune those two with further experiments.

Above: Dropout for 1st lyr of FFN, batch_size = 64. Below: Batch sizes, dropout = 0.2

Dropout Value	Training Accuracy	Validation Accuracy
0.2	0.8015895953757225	0.801605504587156
0.3	0.5965317919075145	0.5011467889908257
0.5	0.7287572254335261	0.7809633027522935

batch_size	Training Accuracy	Validation Accuracy
32	0.505635838150289	0.4954128440366973
64	0.8015895953757225	0.801605504587156
128	0.8362716763005781	0.8027522935779816
256	0.8726878612716763	0.7511467889908257





The following are graphs of the tables given above.

From these graphs, we see that a dropout rate of 0.2 results in higher accuracy. Batch_sizes of 64 and 128 perform very similarly, although it seems the graph is indicating overfitting to the training data for higher numbers. I am not sure why that would be the case and need to investigate further. For now, I decided to stick to a batch_size of 64. From this best performing model, the test set was evaluated. This resulted in an accuracy of 0.7402526084568918 and a loss of 0.62267172439345.

As a sanity check, I tested a few arbitrary sentences as well. For "that movie was bad", there was a 0.9204174 probability that it was negative, and "best movie ever" was positive with probability 0.9158368.

2 Experimentation: Recurrent Units

To convert the model from using a simple rnn to an lstm when encoding, I simply changed the value for Seq2VecEncoder in the config file. Following a similar experimental procedure as before, I first trained the model as-is for a baseline. I observed that the accuracy for the training set was 99% while the accuracy for the validation set had risen marginally to 81%. This indicated that I may need to increase the dropout rate for the feed forward network.

For a fair comparison, I modified all the same hyperparemeters, choosing values that maximimzed accuracy. This way, I can compare the rnn and lstm at their best performance/accuracy. I chose 'leaky_relu' for the non-linearity, did not change epochs and patience, and ended up choosing the same optimizer. I then ran the same experiments for dropout and batch_size as part 1.

Changing dropout value in the FFN, batch_size = 128

Dropout Value	Training Accuracy	Validation Accuracy
0.2	0.9709537572254335	0.8096330275229358
0.4	0.9589595375722544	0.8211009174311926
0.5	0.9700867052023121	0.786697247706422

Changing batch_size in iterator, dropout = 0.4

batch_size	Training Accuracy	Validation Accuracy
64	0.991907514450867	0.801605504587156
128	0.9589595375722544	0.8211009174311926
256	0.9955202312138728	0.7993119266055045

Based on these results, I chose a dropout of 0.4 and batch_size of 128 along with the other hyperparameters for the LSTM version of the model. The final accuracy on the test set was 0.7803404722679846 (0.04 increase) with a loss of 0.9185794219374657 (0.3 increase). I repeated the prediction test on the same sentences, and found that this model was 98-99% confident about labels for both of the sentences (7% increase from rnn).

Longer sequences

To test whether the LSTM version of the model had better accuracy than the RNN for longer sequences, I trimmed the test data to only include lines that were more than 250 characters long, leaving 1073 lines. I then evaluated the performance of the models using the trimmed data.

Test Accuracy and Loss for reviews > 250 chars

Model	Test Accuracy	Test Loss
RNN	0.7231638418079096	0.6779340952634811
LSTM	0.7491525423728813	1.007767528295517

Here, we see that test accuracy for longer sequences does increase somewhat, although loss increases as well. However, I think that the LSTM model may simply be more accurate in general than only over longer sequences. To verify, I got the same metrics for smaller sentences.

Test Accuracy and Loss for reviews $\leq$ 250 chars

Model	Test Accuracy	Test Loss
RNN	0.7561497326203208	0.5634443481763204
LSTM	0.8096256684491978	0.6573029458522797

From this, it appears that the LSTM is more accurate in general and not only over longer sequences. This result may be caused by my choice of 250 characters to separate long and short sentences, and we may see a larger difference in accuracy depending on what a long sentence is defined to be. I also added a long sentence to the prediction task to see which model was more confident about its labelling. The LSTM model labelled the long sentence correctly with a probability/confidence of 96.19% whereas the RNN model labelled it *incorrectly* with a probability of 94.02%. This experiment much more clearly shows that the LSTM model is much better at labelling longer

sentences. I think that this may also have to do with the specific sentence itself: "hardly makes the kind of points.., nor does it....". This sentence may sound overall positive without the they keywords "hardly" and "nor does it". The RNN may have gotten confused by the relative lack of negative words and forgotten the 'hardly' at the start of the sentence, while the LSTM would have 'remembered' it.

3 Pretrained Word Embeddings

I chose to use the GloVe pretrained embeddings. According to their website, (nlp.stanford.edu/projects/glove/) is trained on the Common Crawl corpus which is just a collection of web crawl data. In contrast, the embeddings used in part 1 and 2 are trained on a subset of Google News' data. This means that GloVe may have seen a more diverse set of words in training. I chose the 300 dimension version of the GloVe embeddings in order to have a fairer comparison. I also found that word2Vec has around 3 million words, and chose the GloVe version with the closest vocabulary size (glove.840B.300d).

Following the same procedure as the previous parts, I settled on my hyperparameters. I decided to remove gradient normalization with these new embeddings, as they may no longer be needed for the new set of vectors. I used 'leaky_relu' for my activation function, 'dense_sparse_adam' for the optimizer, the same number of epochs/patience, reduced the dropout back to 0.2, and reduced the 'batch_size' back to 64. From this model, the accuracy results were as follows:

Training: 0.986

Validation: 0.826

Test: 0.841

Using these different embeddings, there wasn't a big difference between training and validation accuracies. However, test accuracy was much higher. The embeddings for part 2 gave us a test accuracy of 0.78 with a loss of 0.91. The new embeddings result in an accuracy of 0.84 with 0.38 loss. The GloVe embeddings performed much better for this task as accuracy increased and loss decreased at the same time.

Antonyms

Embeddings of antonyms may have a tendency to be similar because they will inevitably share features. Words like 'good' or 'bad' are both adjectives, both have few letters, will be seen in similar environments, and are likely to have a similar frequency of usage among other common features. This means that many of the dimensions (in our case 300), might be very close for embeddings between antonyms. If we increase the number of dimensions, I think it becomes less likely for the embeddings to be very close. However, reducing dimensions will not necessarily result in closer embeddings if the features are well chosen (by us or the network).

A very simple word that is informative to a sentiment classifier is 'good' with its antonym 'bad'. In word2Vec, the cosine distance of bad from good is 0.719005. It is the second closest embedding

after 'great'! The 10^{th} closest embedding is 'lousy' and the 12^{th} closest is 'terrible'. A very simple test sentence for these vectors would be 'That movie was good/bad' or 'A pretty good/bad show'. Since these embeddings are in word2Vec, I used the implementation in part 2 to test these sentences with the predictor. For 'that movie was bad', the predictor correctly classified it as being negative, with 99% probability. Its counterpart 'that movie was good', was *incorrectly* classified as positive with probability 66% (very low confidence). 'A pretty good/bad show' was correctly classified in both cases with 99% probability. Since swapping out w for w' changed the prediction in one case, the embeddings are close enough to be confused with each other, but far enough that other factors are still important.

References

[1] Matt Gardner et al. "AllenNLP: A Deep Semantic Natural Language Processing Platform". In: 2017. eprint: `arXiv:1803.07640`.

[2] Tomas Mikolov et al. "Efficient Estimation of Word Representations in Vector Space." In: *Workshop at ICLR*. 2013.

[3] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "GloVe: Global Vectors for Word Representation". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, pp. 1532–1543. URL: `http://www.aclweb.org/anthology/D14-1162`.

[4] Richard Socher et al. "Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher Manning, Andrew Ng, and Christo- pher Potts". In: *EMNLP*. 2013.