Pros & Cons Sentiment Analysis Neural Network

COMP 3503 – Tyler McKay – 150187m

Problem Description

Using ~2000 examples scrapped from the website Epinions.com; can a machine learning model accurately classify the sentiment behind a short snippet of text? Can it distinguish between positive and negative sentiment? This model will train on the 2000 labeled examples and see if it can accurately predict against ~44,000 different unlabeled examples.

Neural networks have very quickly become extremely powerful and complex in the last 5-8 years. Sentiment analysis should be trivial for even a fairly simple model. 90%+ accuracy on the training data is to be expected.

Background

Python is currently the machine learning programming language of choice for most data scientists. There are a handful of common place python packages as well, such as numpy, pandas, tensorflow, matplotlib, etc. The Jupyter notebook format is also a popular means of mixing rich text and python code into a single file. These will be the main technologies used in this project.

There are many existing examples of sentiment analysis using neural networks, and this project will not stray far from the mean. A handful of examples can be found in the references section.

Method and Approach

The first step in any data science project is to consolidate and clean the data that will be used during training. The 2000 training examples are formatted in XML; however, they do not contain valid XML, thus some changes need to be made. Each line in the file is a single instance, so it is possible to traverse the file line by line and make alterations when necessary. Instead of using XML it will be converted into the CSV format. On each line, the text data is wrapped by an XML tag in a format like: [<Pro>this is the text data</Pro>]. Using the Python regex library and the built-in string manipulation tools, the XML tags are removed, and the text is converted into valid CSV format. The changes look like: [”this is the text data”,1]. You may notice that the text is now wrapped in double quotes, this is because the text data occasionally contains commas [and other problematic characters] which ruin the validity of a CSV file. Encasing the text in quotes allows the file to treat the text as a single element, instead of splitting it up by an extraneous character. These changes were done to both the training data and the prediction data.

After the data has been cleaned, it is time to split it into train, validation, and prediction sets. Using a 65% validation split, it results in about ~1300 training examples, ~700 validation examples, and ~44,000 prediction examples.

Before feeding the training data into the model, it needs to be transformed in a way such that the neural network can interpret it. Unsurprisingly, neural networks can't read plain text. This is where the Tokenizer from the TensorFlow library becomes useful. Supplying the tokenizer with a list of text data, it will create an indexed vocabulary of words based upon the contents of the text. It can then take that text data again, and then convert each individual instance into a list of indices corresponding to the index in the word vocabulary. This is what the neural network will be train upon.

Now to the architecture of the neural network. The model is a simple **sequential** network, with only a handful of layers. The most important here is the **embedding** layer. It takes the vocabulary-indexed text data and converts each 'word/index' into an n-dimensional vector of real numbers, which correspond to the relation each word has to every other word in the vocabulary. This layer itself has weights and 'learns' on its own, attempting to create an adequate 'embedding' of the vocabulary. The outputted list of n-dimensional real vectors created by the embedding layer gets fed into a combination **bidirectional** and **long short-term memory** (LSTM) layer. Starting with the LSTM layer; it is a layer that is designed around learning order dependance of a sequence. That is, how does the given set of words all relate to one another, in the order that they are supplied. This is extremely beneficial, as in written text, the order in which something is said matters quite significantly. Two-fold, the bidirectional wrapper layer allows the network to feed the sequence of vectors in, in both directions. This is helpful because the LSTM layer can gleam helpful information when given the context of front-to-back and back-to-front. Then follows two 'general' **dense** layers. Then a single-output dense layer which outputs a prediction in the range [0,1] indicating either 'Pro' or 'Con'.

Experimental Design

The training and evaluation of the network followed a very simple methodology of tweaking a single element at a time and observing how that affected the results. The main metric to focus on for a binary classification problem like this is accuracy, however it may also be beneficial to monitor the f1 score [which is a combination of precision and recall].

Tweaking the number of nodes in each of the layers, modifying the learning rate, adding/remove a layer or two, adjusting the vocabulary size, playing with the number of training epochs; all of these different techniques were attempted at some point during development.

Results

Considering the number of different experiment conditions attempted, the only real effect they had on the final results were by a few percentages. Results ranged from ~94-99% accuracy on the training/validation sets. This is not extremely significant with only ~2000 training examples.

Upon analyzing the 'prediction text' against the actual predictions generated by the model, it is clear to see that the model struggles in somewhat ambiguous situations. Any example which contained 'positive' words that were used in more of a 'negative' context confused the model. This can be attributed to the relatively small amount of training examples and the extremely large amount of prediction examples. There is just not enough context for the model to use when predicting ~44,000 unseen examples.

After each training session the model would output 10 random 'predictions texts' and it's prediction for that example. On average there would be 1-2 examples which were clearly incorrect. Based on these observations, it would be safe to assume that the model can accurately predict at ~80-90% accuracy in real world use cases.

Discussion

The logical progression of this model would be to supply it with additional data to train upon. Instead of just ~2000 examples, present it the entire ~45,000 examples of the train and prediction data together. Neural networks vastly improve in generalization the more unique data they can observe.

With the additional data being supplied to the model, it would also be beneficial to expand upon the architecture of the network. Increasing the number of nodes, adding additional layers, etc. This would increase the training times but would also allow the network to have more complex connections, and perhaps allow it to better comprehend the difference between a 'positive' and 'negative' sentiment.

Conclusion

Neural networks have come a long way in the last 5-8 years. This project is a great example of how accessible this type of analysis has really become. In less than 250 lines of python code. It is possible to create a very reasonably accurate model that can predict the emotional sentiment behind a short snippet of text. This kind of analysis would beyond comprehension only 20 years ago. However, a great deal is still unknown.

It is possible to create a model such as the one found in this project, but it is still mostly a mystery to **why** the model comes to the conclusions that is does. On a small network of only a handful of nodes and connections, it would be possible for a human to brute force an understanding of why the model behaves the way it does. But most models in-use today are well beyond that scope.

The extremely impressive natural language processing model GPT-3 created by OpenAI has close ~175,000,000,000 billion individual neurons and was trained on over 500,000,000,000 billion tokens. It is **impossible** for an individual to understand the reasoning behind any single 'choice' that GPT-3 makes. There are just too many contributing factors.

Aside

There are not images, graphs, or examples present in this report. This is intentional. The corresponding Jupyter notebook containing the source code from this project has been augmented and annotated in a way such that you can understand and follow the concepts discussed in this project easily. I recommend opening this report and the notebook in conjunction and follow along simultaneously. The entirety of this written report is contained within the notebook for easy learning.

References

tf.keras.layers.Embedding
tf.keras.preprocessing.text.Tokenizer
Bidirectional LSTMs with TensorFlow 2.0 and Keras
Sentiment Analysis Using Convolutional Neural Network
How to Use Word Embedding Layers for Deep Learning with Keras
A Gentle Introduction to Long Short-Term Memory Networks by the Experts
Deep Convolutional Neural Network for Sentiment Analysis (Text Classification)
Word Embeddings Versus Bag-of-Words: The Curious Case of Recommender Systems