CS395T Project 3: Neural Networks for Sentiment Analysis

Due date: Thursday, November 2 at 9:30am

Collaboration As stated in the syllabus, you are free to discuss the homework assignments with other students and work towards solutions together. However, all of the code you write must be your own! Your extension should also be distinct from those of your classmates and your writeup should be your own as well. **You should list your collaborators at the top of your written submission.**

Goal: In this project, you will implement two different neural networks for sentiment analysis: a feed-forward neural network in the style of Iyyer et al. (2015) and either an RNN or CNN-based approach of your choosing. The goal of this project is to give you experience implementing standard neural network architectures in Tensorflow for an NLP task.

Background

Sentiment analysis comprises several related tasks: binary classification of sentences as either positive or negative (Pang et al., 2002), ordinal classification using a star system (Pang and Lee, 2005) or a range from strongly negative to strongly positive (Socher et al., 2013), and others. Traditionally, Naive Bayes or SVM bag-of-words models worked relatively well (Pang et al., 2002; Wang and Manning, 2012). However, these have recently been supplanted by neural network approaches including convolutional networks (Kim, 2014) and feedforward networks Iyyer et al. (2015).

The dataset that we use in this project consists of sentences from Rotten Tomatoes classified as either positive or negative sentiment, as introduced in Pang and Lee (2005). While standard evaluation uses cross-validation to evaluate on the whole dataset (Kim, 2014), we have simply broken it into 80/10/10 train/dev/test splits for you to use. Results you get may not be exactly comparable to those in other papers but they should be close. This dataset was also evaluated on in Kim (2014) (as the MR dataset) and in Iyyer et al. (2015) (as the RT dataset).

Your Task

In this project, you'll be implementing two neural networks: a feedforward neural network based on averaged word vectors over the input sentence and another neural network architecture of your choosing.

Getting Started

In addition to the setup for Projects 1 and 2, you will additionally need to install tensorflow and its dependencies. You should follow the instructions at `https://www.tensorflow.org/install/`. If you have OS X El Capitan, you should take their advice and use virtualenv—the default system python interpretation has an incompatible version of sixt that will likely give you problems otherwise.

To get started, try running:

```
python feedforward_example.py
```

This trains and evaluates a neural network on a set of 6 training examples of the XOR function. This file is heavily commented: you should refer to it as you build your own networks if you're stuck and not sure how to do something. It walks through the three main important components of a Tensorflow program: defining the computation graph, defining the training algorithm, and actually running training and evaluation on the data. In this case, the computation graph is a simple feedforward neural network with one hidden layer. Training uses Adam and with a small learning rate decay every 10 gradient steps. The actual training iterates through the training points one at a time and calls `sess.run` for each one, evaluating the train operator to apply gradient updates and accumulating the loss.

We provide a harness in `sentiment.py` for reading in the datasets and running your system. `sentiment_data.py` helps load and store the dataset. This process has been broken into several steps:

1. Load in the sentiment dataset, tokenize it, and figure out its vocabulary [done offline in advance]
2. Load the word vectors and write them back out using only the vocabulary of the current dataset. This process is known as relativization [done offline in advance]
3. Load the relativized word vectors
4. Load the sentiment dataset, tokenize it, index words, and convert any word with no known word vector to UNK, which is assigned the zero vector.

`SentimentExample` is a simple wrapper around an indexed sentence and a binary label (0/1, 1 is positive sentiment). `WordEmbeddings` is a wrapper around word embeddings, represented as a numpy matrix and a word indexer, where the ith row of the matrix is the embedding of the ith word in the indexer. We have provided two sets of vectors: 50-dimensonal and 300-dimensional GloVe vectors (Pennington et al., 2014). Larger vectors typically work a bit better (up to a point), but can be significantly slower depending on what kind of network you're training.

Part 1: Feedforward Neural Networks

In this part of the project, you should implement a feedforward neural network similar to the architecture from Iyyer et al. (2015). This model works by averaging together word embeddings from the sentence, then using that as a fixed-length input to a feedforward neural network with one or more hidden layers. You are given substantial leeway as to how many layers you use and the hyperparameter choices (optimizer, nonlinearity, dropout, training regimen, whether you fine-tune embeddings, etc.). You may wish to use `feedforward_example.py` as a template for how to design your code. See also the Tensorflow Tips section for some advice on how to implement certain operations.

There are no hard performance requirements for this part—it's primarily meant to be a stepping stone to part 2. A tuned implementation should be able to replicate the results from Iyyer et al. (2015), but a quick-and-dirty implementation should get at least 70% accuracy using the 300-dimensional vectors as input. Our reference implementation gets around 76% accuracy here using the 300-dimensional vectors and two hidden layers.

Part 2: RNN or CNN

In this part, you should implement a structured neural network for the same sentiment task as in part 1. Specifically, you can choose to either implement an RNN (LSTM/GRU/etc.) or a CNN (pooling or dilated). Each of these has additional hyperparameters to tune beyond those in part 1. For RNNs: the choice of cell type (this space is large!), whether you use a bidirectional model or not, whether you use more than one LSTM layer or not, whether you pool the outputs from every state or just use the output from the last state, and what kind of dropout you use. For pooling CNNs: the number of filters of each width, what kind of pooling you use, the number of CNN layers, the number of feedforward layers, and whether you use wide or narrow convolutions.

For this part, the network you build should get at least 75% accuracy on the development set. Strong implementations should get close to or exceed 80%. Our bidirectional LSTM got 78.5% accuracy on the development set training on a laptop for roughly 20 minutes. Note that you can make networks run for arbitrary amounts of time; you'll have to judge yourself how large they need to be to be effective.

You are allowed to consult existing implementations and documentation in the literature as you optimize these methods; however, **the code you write must be your own**! Your writeup should primarily emphasize the exploration you did both here and in part 1. What decisions mattered? What helped and what didn't? Did you do anything special? Feel free to pursue this in an open-ended way and document your exploration! This part constitutes the extension for this project, so provide an appropriate amount of comparison and analysis.

Implementation Tips

- You should be able to do the vast majority of your parameter tuning in small-scale experiments. Try to avoid running large experiments on the whole dataset in order to keep your development time fast.
- It's up to you whether you want to initialize word vectors randomly and learn them, treat the pretrained embeddings as fixed, or initialize with pretrained and then learn, all options which are discussed in Kim (2014)). Think about the speed, modeling, and learnability tradeoffs each of these represents.
- Parameter initialization is important! Make sure all hidden layers in feedforward neural networks are initialized to nonzero values. You can use `tf.contrib.layers.xavier_initializer` for the Glorot initializer.

Tensorflow Tips

Google and the Tensorflow docs[1] are your friends.

Basic tensor manipulation `tf.reshape` is useful for changing shapes of tensors while preserving the data, e.g. if you end up with a tensor of shape $[1, k]$ and you need to turn it into a vector of length $[k]$. Similarly, `tf.transpose` lets you permute axes of tensors. `tf.concat` lets you concatenate them.

Arithmetic Tensorflow supports most basic arithmetic operations done elementwise on tensors. For matrix multiplication, you probably want to use `tf.tensordot`; `tf.matmul` is pretty restrictive in the types of inputs it can deal with.

Word vectors A sentence is represented as an indexed sequence of word indices. We need some way of converting these to embeddings. You likely want to use `tf.embedding_lookup` for this purpose. Alternatively, you can use `tf.one_hot` to replace word indices (e.g., 74) with one-hot vectors (a vector the length of the vocabulary with a one in the 74th position) and compute embedded representations using matrix multiplication, but this may be slower.

[1] `https://www.tensorflow.org/api_docs/python/tf`

Dealing with sentence data Sentences are tricky inputs to deal with because they are different length. The easiest way to handle this is to find the max sentence length and pad all sentences to be at least this size. Make sure your padding doesn't change the computation you're doing! For convolutional networks, this can work pretty well. However, for LSTMs, you'll likely end up needing to extract the outputs from a `[batch_size, max_sentence_length]` tensor corresponding to the end of each sentence. `tf.gather` is useful for this purpose: it lets you read out particular indices of values from a tensor.

If you want to be even fancier, you can build several copies of the computation graph of different sizes, pad each batch of data only to the smallest greater computation graph size, and run using that graph.

LSTMs You probably want to use the `tf.rnn.static_rnn` for implementing RNNs (or you can use dynamic RNNs). Many different cell types are possible, including LSTM cells (`BasicLSTMCell`), GRUs, and more.

CNNs If you want to use CNNs, you might investigate `tf.nn.conv2d`.

Submission and Grading

You should submit three files on Canvas:

1. Your code (a .zip or .tgz file)
2. Your best model's output on the blind test set (a .txt file).
3. A report of around 2 pages, not including references, though you aren't expected to reference many papers. Your report should list your collaborators, **briefly** restate the core problem and what you are doing, describe relevant details of your implementation, present results from both your feedforward and structured model, and discuss what choices you found to be important and which mattered less. Your report should be written in the tone and style of an ACL/NIPS conference paper. Any LaTeX format with reasonably small (1" margins) is fine, including the ACL style files[2] or any other one- or two-column format with equivalent density.

Slip Days Per the syllabus, you have seven slip days to use during the semester, so you may choose to use part or all of your budget on this process to turn it in late as needed.

References

Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. 2015. Deep Unordered Composition Rivals Syntactic Methods for Text Classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*.

Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Bo Pang and Lillian Lee. 2005. Seeing Stars: Exploiting Class Relationships for Sentiment Categorization with Respect to Rating Scales. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL)*.

[2]Available at `http://acl2017.org/calls/papers/`

Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. 2002. Thumbs up? Sentiment Classification using Machine Learning Techniques. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Sida Wang and Christopher Manning. 2012. Baselines and Bigrams: Simple, Good Sentiment and Topic Classification. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL)*.