CS388 Mini 2: Neural Networks for Sentiment Analysis

Due date: Tuesday, February 23 at 11:59pm CT

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 and your writeup must be your own! **Please 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 feedforward "deep averaging" 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 PyTorch 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).

You'll be using the movie review dataset of Socher et al. (2013). This is a dataset of movie review snippets taken from Rotten Tomatoes. The labeled data actually consists of full parse trees with each constituent phrase of a sentence labeled with sentiment (including the whole sentence). The labels are "fine-grained" sentiment labels ranging from 0 to 4: highly negative, negative, neutral, positive, and highly positive.

We are tackling a simplified version of this task which frequently appears in the literature: positive/negative binary sentiment classification of sentences, with neutral sentences discarded from the dataset. The data files given to you contain of newline-separated sentiment examples, consisting of a label (0 or 1) followed by a tab, followed by the sentence, which has been tokenized but not lowercased. The data has been split into a train, development (dev), and blind test set. On the blind test set, you do not see the labels and only the sentences are given to you. The framework code reads these in for you.

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 previous assignments, you will additionally need to install Pytorch and its dependencies. You should follow the instructions at `https://pytorch.org/get-started/locally/`. All assignments for this course are small-scale enough to complete using CPUs, so don't worry about installing CUDA and getting GPU support working unless you want to. Installing through anaconda often works well.

To get started, try running:

```
python ffnn_example.py
```

This trains and evaluates a neural network on a set of 4 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 Pytorch program: defining the computation graph modules, preprocessing the data and setting up the data pipeline, 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 trained with Adam. The actual training iterates through the training points one at a time, computes the loss on each one, computes the gradient, and applies the gradient in the optimizer.

We provide a harness in `sentiment.py` for reading in the datasets and running your system. Helper functions in `sentiment_data.py` 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 in the sentiment dataset and tokenize it

`SentimentExample` is a simple wrapper around a 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 better, but the smaller vectors can be faster for debugging initial stages of your development.

`models.py` is the file you'll be modifying, which provides you an entry point for your two models.

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 `ffnn_example.py` as a template for how to design your code. See also the Pytorch Tips section for some advice on how to implement certain operations.

For full credit on this part, your model should get at least 77% accuracy on the development set. Our reference implementation's performance varies depending on the initialization, but it can get around 78% accuracy with 300-dimensional embeddings in less than 5 minutes of training.

Part 2: RNN or CNN

In this part, you should implement a more sophisticated 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. Each of these has additional hyperparameters to tune beyond those in part 1. For RNNs: the choice of cell type, 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 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. We haven't

discussed CNNs much in this course, so only choose this option if you are already familiar with CNNs or feel comfortable picking up these concepts from elsewhere.

For full credit this part, your model should get at least 81% accuracy on the development set. Strong implementations can exceed 83%.

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 briefly describe the exploration you did both here and in part 1. What decisions mattered? What helped and what didn't? Did you do anything special?

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 control this by calling initializers from `torch.nn.init` on your modules' weight matrices.
- Consider using batch training. You'll need to make sure your computation graph can handle multiple examples fed in at once.

Optional: ELMo / BERT If you want, you may explore using ELMo or BERT to improve performance in this part. **However, this is strictly optional and will not be counted towards your assignment grade.** If you use ELMo in a "frozen" manner, you can cache the extracted ELMo vectors; this is computationally feasible even without GPU resources. However, fine-tuning ELMo or BERT will probably require long running times or using GPU resources, which we cannot make available to all students.

PyTorch Tips

Google/Stack Overflow and the PyTorch documentation are your friends.

Basic tensor manipulation For creating tensors, `torch.tensor` and `torch.from_numpy` are pretty useful. For manipulating tensors, `permute` lets you rearrange axes, `squeeze` can eliminate dimensions of length 1, `expand` can duplicate a tensor across a dimension, etc. You probably won't need to use all of these in this project, but they're there if you need them. Pytorch supports most basic arithmetic operations done elementwise on tensors.

Word vectors To handle sentence input data, you typically want to treat the input as a sequence of word indices. You can use `torch.nn.Embedding` to convert these into their corresponding word embeddings; you can initialize this layer with data from the WordEmbedding class using `from_pretrained`. By default, this will cause the embeddings to be updated during learning, but this can be stopped by setting `requires_grad_(False)` on the layer.

Dealing with sentence data Sentences are tricky inputs to deal with because they are different lengths. 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. For LSTMs with batches of size greater than one, you will probably want to look at `torch.nn.utils.rnn.pack_padded_sequence`. This converts a padded sentence representation into a packed format that can be consumed by the default RNN implementations in Pytorch.

LSTMs You probably want to use `torch.nn.LSTM` or `torch.nn.GRU`.

CNNs If you want to use CNNs, you might investigate `torch.nn.Conv2d`.

Submission and Grading

You should submit on Canvas **as three file uploads**:

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 1 page. Your report should list your collaborators, **briefly** restate the core problem and what you are doing, describe relevant details of your implementation, and present a table of results. No abstract is needed for such a short report.

Slip Days Slip days may be used on this assignment. See the syllabus for details about the slip day policy.

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)*.

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)*.