CS88 Mini 2: Neural Networks for Sentiment Analysis

Due date: Tuesday, October 8 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 must be your own! Your extension should also be distinct from those of your classmates and your writeup should be your own as well. **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).

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) (called "MR" in that paper) and in Iyyer et al. (2015) (called "RT").

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. This assignment spec will assume you are using Pytorch. **Project 2's framework code will be in Pytorch**, so it's recommended that you familiarize yourself with Pytorch. However, you are free to use Tensorflow if you are more comfortable with it.

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 in a virtual environment is usually best; we recommend using anaconda, especially if you are on OS X, where the system python has some weird package versions. Once you have anaconda installed, you can create a virtual environment and install pytorch with the following commands:

```
conda create -n my-cs388-virtenv python=3
conda install -n my-cs388-virtenv -c pytorch pytorch torchvision
```

where `my-cs388-virtenv` can be any name you choose. (You can also install tensorflow, tensorboard, or any other necessary packages in this virtual environment as well.)

To get started, try running:

```
python feedforward_example_pytorch.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 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 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_pytorch.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 74% accuracy on the development set. Our reference implementation gets around 76% accuracy here using the 300-dimensional vectors and two hidden layers trained for 20 epochs.

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 76% accuracy on the development set. Strong implementations can exceed 80%. Our bidirectional LSTM gets around 78% 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 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 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 `torch.nn.Embedding` for this purpose.

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)*.