CS378 Assignment 2: Feedforward Neural Networks

Due date: Thursday, February 21 at 5:00pm CST

Academic Honesty: Reminder that assignments should be completed independently by each student. See the syllabus for more detailed discussion of academic honesty. Limit any discussion of assignments with other students to clarification of the requirements or definitions of the problems, or to understanding the existing code or general course material. Never discuss issues directly relevant to problem solutions. Finally, you may not publish solutions to these assignments or consult solutions that exist in the wild.

Goals The main goal of this assignment is for you to get experience training neural networks over text. You'll play around with feedforward neural networks in PyTorch and see the impact of different sets of word vectors on a sentiment classification problem.

Dataset and Code

Please use Python 3.5+ and PyTorch 1.0 for this project.

Data The dataset you'll be using here is the same as in Assignment 1.

Installing PyTorch You will need PyTorch 1.0 to run the code. **If you are using CS lab machines, PyTorch 1.0 should already be installed and you can skip this step.** To get it working on your own machine, you should follow the instructions at `https://pytorch.org/get-started/locally/`. The assignment is small-scale enough to complete using CPU only, 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 with the following command:and install PyTorch with the following commands:

```
conda create -n my-cs378-virtenv python=3
```

where `my-cs378-virtenv` can be any name you choose. Then, if you're running Linux, install PyTorch with:

```
conda install -n my-cs378-virtenv -c pytorch pytorch-cpu torchvision-cpu
```

If you're on Mac, use:

```
conda install -n my-cs378-virtenv -c pytorch pytorch torchvision
```

Getting started Download the code and data. Expand the tgz file and change into the directory. To confirm everything is working properly, run:

```
python neural_sentiment_classifier.py --model TRIVIAL --no_run_on_test
```

This loads the data, instantiates a `TrivialSentimentClassifier` that always returns 1 (positive), and evaluates it on the training and dev sets. Compared to Assignment 1, this runs an extra word embedding loading step.

Framework code The framework code you are given consists of several files. `neural_sentiment_classifier.py` is the main class, which is very similar to the main class from Assignment 1. **Do not modify this file for your final submission**, though it's okay to add command line arguments during development or do whatever you need. It uses `argparse` to read in several command line arguments. You should generally not need to modify the paths. `--model` and `--feats` control the model specification. This file also contains evaluation code. The main method loads in the data, initializes the feature extractor, trains the model, and evaluates it on train, dev, and blind test, and writes the blind test results to a file.

Data reading in `sentiment_data.py` and the utilities in `utils.py` are exactly as in Assignment 1. `sentiment_data.py` now additionally contains a `WordEmbeddings` class and code for reading it from a file. This class wraps a matrix of word vectors and an Indexer in order to index new words.

Your modifications will take place in a few files, as described in Part 2. First, you will explore `optimization.py` more in Part 1.

Part 1: PyTorch Warmup and Optimization (20 points)

In this part, you'll get some initial familiarity with using PyTorch to optimize functions. We define two functions in `optimization.py`. The first is a quadratic with n variables:

$$y = \sum_{i=1}^{n} (x_i - 1)^2$$

The second is the Rosenbrock function, a classic test for optimization algorithms:

$$y = \sum_{i=1}^{n-1} \left(100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right)$$

Both attain their minima at $x = (1, 1, \ldots, 1)$, all ones.

The main function in `optimization.py` will optimize one of these functions (depending on the command line argument) using PyTorch. For example, to run Adam on the quadratic function with learning rate of 1.0, run:

```
python optimization.py --func QUAD --method ADAM --lr 1.0
```

Q1 (10 points) Compare SGD and Adam on the quadratic function. Try different step sizes. For each optimization technique, describe the range of behaviors you see depending on step size. (Hint: try varying step sizes by orders of magnitude, like 1, 0.1, 0.001, etc.)

Q2 (10 points) Compare SGD and Adam on the Rosenbrock function. Try different step sizes. For each optimization technique, describe the range of behaviors you see depending on step size. (Hint: try varying step sizes by orders of magnitude, like 1, 0.1, 0.001, etc.)

Part 2: Deep Averaging Network (55 points)

In this part, you'll implement a deep averaging network as discussed in lecture and in Iyyer et al. (2015). If our input $s = (w_1, \ldots, w_n)$, then we use a feedforward neural network for prediction with input $\frac{1}{n} \sum_{i=1}^{n} e(w_i)$, where e is a function that maps a word w to its real-valued vector embedding.

You are given two sources of pretrained embeddings you can use: `data/glove.6B.50d-relativized.txt` and `data/glove.6B.300d-relativized.txt`, the loading of which is controlled by the `--word_vecs_path`. These are trained using GloVe (Pennington et al., 2014). These vectors have been *relativized* to your data, meaning that they do not contain embeddings for words that don't occur in the train, dev, or test data. This is purely a runtime and memory optimization.

`models.py` is the primary file you'll be modifying for this part. `train_deep_averaging_network` is your main entry point: this function should return an instance of `NeuralSentimentClassifier`, which you should implement as well. Usage of this class is similar to the classifiers in Assignment 1.

PyTorch example `ffnn_example.py` implements the network discussed in lecture for the synthetic XOR task. It shows a minimal example of the PyTorch network definition, training, and evaluation loop. Feel free to refer to this code extensively and to copy-paste parts of it into your solution as needed. However, also feel free to modify it to change and improve any aspects that seem relevant.

Most of this code is self-documenting. **The most unintuitive piece is calling `zero_grad` before calling backward!** Backward computation uses in-place storage and this must be zeroed out before every gradient computation. Not doing so effectively leads to an extra momentum term in the optimizer and can cause mysteriously bad (but not horrific) performance. It's also important to initialize your weights appropriately.

Implementation Following the example, the rough steps you should take are:

1. Define a subclass of `nn.Module` that does your prediction. This should return a log-probability distribution over class labels.
2. Compute your classification loss based on the prediction. In lecture, we saw using the negative log probability of the correct label as the loss.
3. Call `network.zero_grad()` (zeroes out in-place gradient vectors), `loss.backward` (runs the backward pass to compute gradients), and `optimizer.step` to update your parameters.

Implementation and Debugging Tips Come back to this section as you tackle the assignment!

- You should log training loss over your models' epochs; this will give you an idea of how the learning process is proceeding.
- 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.
- If you see NaNs in your code, it's very likely due to a large step size.
- 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.
- There are two major approaches to handling sentence input data. The first is to pre-average the embeddings for each sentence (outside of PyTorch). The second is to treat the input as a sequence of word indices. In this case, you likely want to use `torch.nn.Embedding` for this purpose. You'll need to do the second if you want to fine-tune your word embeddings.

- Google/Stack Overflow and the PyTorch documentation[1] are your friends. Although you should not seek out prepackaged solutions to the assignment itself, you should avail yourself of the resources out there to learn the tools.

Q3 (30 points) Implement the deep averaging network. You should get least 75% accuracy on the development set in less than 15 minutes of train time on a CS lab machine (and you should be able to get good performance in 3-5 minutes). Your implementation should consist of averaging vectors and using a feedforward network, but otherwise you do not need to stick close to what's discussed in Iyyer et al. (2015). Things you can experiment with include varying the number of layers, the layer sizes, which source of embeddings you use (50d or 300d), your optimizer, the nonlinearity, whether you add dropout layers, and your initialization. **Once you're done tuning your model, hardcode the parameters so the command to be run is correct. Briefly describe what you did and report your results in the writeup.**

Q4 (5 points) Try varying the hidden dimension size(s) in your model. Report results for a few different values. What trends that you see? (Try as small as 1 and as large as you have patience for.)

Q5 (5 points) Try varying the learning rate for your chosen optimizer. Report results for a few different values. What trends do you see? (Hint: small changes typically don't make a difference; you should try changing it by a factor of 10 at a time until you feel like you've explored the space.)

Q6 (15 points) Implement batching in your neural network. To do this, you should modify your `nn.Module` subclass to take a batch of examples at a time instead of a single example. You should compute the loss over the entire batch. Otherwise your code can function as before. You can either leave test time as unbatched or batch it, it's up to you. **Try at least one batch size greater than one. Briefly describe what you did, any change in the results, and what speedup you see from running with that batch size.**

Note that if your module is set up to take word indices as input, different sentences of different lengths can cause problems. You'll need to *pad* your inputs so the sentences fit in a square matrix. The typical way to do this is to add a placeholder index that maps to the zero vector. If your module takes pre-averaged word embeddings as input, you won't need to do this.

Part 3: Understanding Word Embeddings (25 points)

Consider the skip-gram model, defined by

$$P(y|x) = \frac{\exp(w_x \cdot c_y)}{\sum_{y'} \exp(w_x \cdot c_{y'})}$$

where x is the "main word", y is the "context word" being predicted, and w, c are d-dimensional vectors corresponding to words and contexts, respectively. Note that each word has independent vectors for each of these, so each word really has two embeddings.

Assume a window size of 1. The skip-gram model considers the neighbors of a word to be words on either side. So with these assumptions, the first sentence above gives the training examples ($x = \text{the}, y = \text{dog}$) and ($x = \text{dog}, y = \text{the}$). The skip-gram objective, log likelihood of this training data, is $\sum_{(x,y)} \log P(y|x)$, where the sum is over all training examples.

[1] `https://pytorch.org/docs/stable/index.html`

Q7 (10 points) Consider the following sentences:

```
the dog
the cat
a dog
```

Suppose the dimensionality of the word embedding space $d = 4$. Write down the following:

(a) The training examples derived from these sentences

(b) A set of vectors that *nearly* optimizes the skip-gram objective. That is, give d-dimensional word and context vectors for *the*, *a*, *dog*, and *cat*. (We say *nearly* because this objective can only be optimized in the limit with vectors of infinite norm. So you can round up to 1 any probability of 0.99 or more.)

(c) The probability of the training set under this set of vectors (again, assuming that $0.99 \approx 1$)

You are allowed to use code to check your answer to this problem, but you should aim to solve it with mathematical reasoning. Do not submit any code for this part.

Q8 (10 points) Consider the following sentences:

```
the dog
the cat
a dog
a cat
```

Now suppose the dimensionality of the word embedding space $d = 2$. Write down the following:

(a) The training examples derived from these sentences

(b) A set of vectors that *nearly* optimizes the skip-gram objective (as in the previous part)

(c) The probability of the training set under this set of vectors

Q9 (5 points) Pick a collection of 4-6 nouns from the sentiment dataset and investigate their corresponding word embeddings. What are the cosine similarities between these? What do these tell you about the geometry of the vector space? Do these make sense? **Briefly describe your answers in a few sentences.** Cosine similarity of vectors $\mathbf{a}$ and b is defined as

$$\cos(\mathbf{a}, \mathbf{b}) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}$$

where $\|\mathbf{a}\|$ is the 2-norm of the vector ($\sqrt{\sum_{i=1}^{n} a_i^2}$). You do not need to submit any code you used for this part.

Deliverables and Submission

Your submission will be evaluated on several axes:

1. Writeup: correctness of answers, clarity of explanations, etc.
2. Execution: your code should train and evaluate within the time limits without crashing
3. Accuracy on the development set of your deep averaging network model
4. Accuracy on the blind test set (you should evaluate your best model and include this output)

Submission You should submit the following files to Canvas **as a flat file upload (no zip or tgz)**:

1. A PDF or text file of your answers to the questions
2. Blind test set output in a file named `test-blind.output.txt`. The code produces this by default, but make sure you include the right version! Include the output from your best deep averaging network model.
3. `models.py`. **Do not modify or upload any other source files.**

Make sure that the following commands work before you submit:

```
python neural_sentiment_classifier.py --word_vecs_path data/glove.6B.50d-relativized.txt
python neural_sentiment_classifier.py --word_vecs_path data/glove.6B.300d-relativized.txt
```

These commands should all print dev results and write blind test output to the file by default.

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)*.

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)*.