CS378 Assignment 2: Feedforward Neural Networks

Due date: Thursday, February 20 at 11:59pm 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 directly discuss details of the 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 the sentiment classification problem from Assignment 1.

Code Setup

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

Installing PyTorch You will need PyTorch 1.0 for this project. **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 via anaconda is typically easiest, especially if you are on OS X, where the system python has some weird package versions. Installing in a virtual environment is recommended but not essential. 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
```

Part 1: Optimization (25 points)

In this part, you'll get some familiarity with function optimization. We define two functions in `optimization.py`. The first is a quadratic with two variables:

$$y = (x_1 - 1)^2 + 8(x_2 - 1)^2$$

The second function is a neural network, described later. The main function in `optimization.py` will optimize one of these functions (depending on the command line argument).

Q1 (10 points) This part deals with optimizing the quadratic function. Run:

```
python optimization.py --func QUAD --lr 0.01
```

to run with a learning rate of 0.01.

a) Implement the gradient of the provided quadratic function in `quadratic_grad`. `sgd_test_quadratic` will then call this function inside an SGD loop and show a visualization of the learning process. **Note: you should not use PyTorch for this part!**

b) With a step size of 0.01, how many steps are needed to get within a distance of 0.1 of the optimal solution? You can round your answer to the nearest 10 steps; you don't need an exact answer.

c) What is the "tipping point" of the step size parameter, where step sizes larger than that cause SGD to diverge rather than find the optimum? You can measure this empirically; you don't need an analytical solution. You can round your answer to the nearest 0.01.

d) When initializing at the origin, what is the best step size to use? Measure this in terms of number of iterations to get with 0.1 of the optimum. You can round your answer to the nearest 0.01. **Include a plot of optimization with this best step size using the provided contour plot machinery.**

Q2 (15 points) We have a basic neural network with one hidden layer implemented with explicit gradient computation "from scratch" in numpy. `sgd_test_nn` optimizes this neural network; you can run it with

```
python optimization.py --func NN --lr 0.01
```

a) If you train the network in its current state, it will not be able to successfully classify the training data. In `sgd_test_nn`, there are **two** errors in how the network is configured and implemented. Fix these so the network gets 100% accuracy on the training data. Note: these errors are not in the forward or backward functions.

b) Modify the code to use ReLU ($x \rightarrow \max(x, 0)$) as the nonlinearity instead of $\tanh$. Your network should train successfully, **and** you should be able to confirm that your gradient matches the "empirical gradient" computed by taking the difference in loss after perturbing the input parameter by a small amount. There is a function included to check this for you.

Part 2: Deep Averaging Network (50 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.

Getting started Download the code and data; the data is the same as in Assignment 1. 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. **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. You should generally not need to modify the paths. The `--model` and `--feats` arguments control the model specification. 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.

`models.py` is the file you'll be modifying for this part, and `train_deep_averaging_network` is your entry point, similar to Assignment 1. Data reading in `sentiment_data.py` and the utilities in `utils.py` are similar to Assignment 1. However, `read_sentiment_examples` **now lowercases the dataset**; the GloVe embeddings do not distinguish case and only contain embeddings for lowercase words.

`sentiment_data.py` also 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. The Indexer contains two special tokens: `PAD` (index 0) and `UNK` (index 1). `UNK` can stand in words that aren't in the vocabulary, and `PAD` is useful for implementing batching later. Both are mapped to the zero vector by default.

Data 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.

PyTorch example `ffnn_example.py`[1] 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. 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.

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. Your module should take a list of word indices as input and embed them using a `nn.Embedding` layer initialized appropriately.

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. You can do this directly, or you can use a built-in loss function like `torch.NLLLoss` or `torch.CrossEntropyLoss`. Pay close attention to what these losses expect as inputs (probabilities, log probabilities, or raw scores).

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.

[1] Available under "Readings" on the course website.

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

- You should print 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 likely due to a large step size. $\log(0)$ is the main way these arise.
- 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` or `repeat` 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.
- 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. 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.
- Google/Stack Overflow and the PyTorch documentation[2] 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 **77% 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 hidden layer sizes, which source of embeddings you use (50d or 300d), your optimizer (Adam is a good choice), the nonlinearity, whether you add dropout layers (after embeddings? after the hidden layer?), 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 (10 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 different sentences have different lengths; to fit these into an input matrix, you will need to "pad" the inputs to be the same length. If you use the index 0 (which corresponds to the `PAD` token in the indexer), you can set `padding_idx=0` in the embedding layer. For the length, you can either dynamically choose the length of the longest sentence in the batch, use a large enough constant, or use a constant that isn't quite large enough but truncates some sentences (will be faster).

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

Q5 (10 points) Try removing the GloVe initialization from the model; just initialize the embedding layer with random vectors, which should then be updated during learning. How does this compare to using GloVe?

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 below 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.

Q7 (5 points) Consider the following sentences:

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

Suppose that we have word embeddings of dimension $d = 2$, the word embedding vectors w for *dog* and *cat* are both $(0, 1)$, and the word embedding vectors w for *a* and *the* are $(1, 0)$. Treat these as fixed.

a) If we treat the above data as our training set, what is the set of probabilities $P(y|\textit{the})$ that maximizes the data likelihood? Note: this question is asking you about what the optimal probabilities $\log P(y|x)$ are *independent* of any particular setting of the skip-gram vectors.

b) Can these probabilities be achieved with a particular setting of the vector for *the*? Give the most optimal vector and a description of why it is optimal.

Q8 (10 points) Now 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. (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.)

Q9 (10 points) Suppose our word embedding space is V dimensions where V is the number of tokens. Each word vector is a one-hot vector: a vector of all zeroes with a single 1 at the corresponding index in the vector space. So word 1 has the word vector $(1, 0, 0, \ldots, 0)$.

a) What is the optimal context vector for a word that occurs with word 1 twice, word 2 once, and word 3 once?

b) Consider a count-based representation of the word in context. That is, if the word occurs with word 1 twice, word 2 once, and word 3 once, the vector is $(2, 1, 1, 0, \ldots, 0)$. How does this representation compare to the representation from part (a)?

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. `optimization.py` and `models.py`. **Do not modify or upload any other source files.**

Make sure that the following commands work before you submit:

```
python optimization.py --func NN
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
```

The last two 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)*.