Assignment 2: Feedforward Neural Networks, Word Embeddings, and Generalization

Academic Honesty Please see the course syllabus for information about collaboration in this course. While you may discuss the assignment with other students, **all work you submit must be your own!**

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 a recent version of PyTorch for this project.

The list of installed packages in the autograder is: `numpy`, `nltk`, `spacy`, `torch`, `scipy`, `matplotlib`, `torchvision`.

Installing PyTorch You will need PyTorch for this project. 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.

Part 1: Optimization (25 points)

In this part, you'll get some familiarity with function optimization.

Q1 (25 points) First we start with `optimization.py`, which defines a quadratic with two variables:

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

This file contains a manual implementation of SGD for this function. Run:

```
python optimization.py --lr 1
```

to optimize the quadratic with a learning rate of 1. However, the code will crash, since the gradient hasn't been implemented.

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) When initializing at the origin, what is the best step size to use? Set your step size so that it gets to a distance of within 0.1 of the optimum within as few iterations as possible. Several answers are possible. Hardcode this value into your code.

Exploration (optional) 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?

Part 2: Deep Averaging Network (75 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. As before, you cannot modify this file for your final submission, though it's okay to add command line arguments or make changes during development. You should generally not need to modify the paths. The `--model` argument controls 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.

You'll want to use `get_initialized_embedding_layer` to get a `torch.nn.Embedding` layer that can be used in your network. This layer is trainable if you set `frozen` to False (which will be slower), but is initialized with the pre-trained embeddings.

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.

Note that the 300-dimensional vectors are used by default. The 50-dimensional vectors will enable your code to run much faster, particularly if you're not using frozen embeddings, so you may find them useful for debugging.

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

[1] Available from Exercise 8b on edX.

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 `NLLLoss` or `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.

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. The `WordEmbeddings` class can produce this layer for you.

- 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.

Q2 (50 points)

a) Implement the deep averaging network. Your implementation should consist of averaging vectors and using a feedforward network, but otherwise you do not need to exactly reimplement 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),

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

the nonlinearity, whether you add dropout layers (after embeddings? after the hidden layer?), and your initialization.

b) 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 also try out batching at test time by changing the `predict_all` method.

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).

Requirements You should get least **77% accuracy** on the development set without the autograder timing out; you should aim for your model to train in less than **10 minutes** or so (and you should be able to get good performance in 3-5 minutes on a recent laptop).

If the autograder crashes but your code works locally, there is a good chance you are taking too much time (the autograder VMs are quite weak). Try reducing the number of epochs so your code runs in 3-4 minutes and resubmitting to at least confirm that it works.

Q3 (25 points)

In the final part of the assignment, you will handle data that contains misspellings. The file `dev-typo.txt` contains a version of the dev set with random typos introduced. Importantly, these typos are never in the first 3 characters, and never in short words. Bag-of-words methods (like Assignment 1) and word embedding-based methods are quite vulnerable to typos, as a single letter change will cause a word to "miss" in the indexer and get discarded.

The model you use for this part **will still only train on the typo-free train set from Q2**. The idea here is to design a model that *generalizes to data from a different distribution*. This requires some additional effort.

You have two options for how to pursue this, or you're free to explore your own solutions:

Option 1: Spelling correction Perhaps the most obvious solution is to try to correct the spelling of the words before passing them into the indexer.

The steps you should take for this are as follows:

1. Implement or find a library to compute *edit distance* between two strings (the number of character-level changes needed to turn one string into the other). This is supported by `nltk`.
2. Modify your inference code to do spelling correction. For a word that doesn't appear in the indexer, you should try to find a word in the indexer that has low edit distance.
3. Optimize your spelling correction approach. Naïve implementations that make $O(|V|)$ edit distance computations per word in the development set will be far too slow for the autograder. See if you can think of clever ways to speed this up and reduce the number of calls there.

Option 2: Prefix embeddings Because the typos are never in the first 3 characters of the word, you can potentially just discard the later information in the word and only use the prefix. It turns out this works surprisingly well!

The steps you should take for this are as follows:

1. Define an indexer and a set of embeddings over characters. You can implement this as a class called `PrefixEmbeddings` that closely resembles the `WordEmbeddings` class.
2. Initialize these embeddings somehow. Hint: one good way to initialize a prefix is by using an average of the word embeddings for words that start with that prefix. (This step is technically optional, but it improves the performance a lot compared to random initialization.)
3. Make sure the embeddings are fine-tunable (not frozen)
4. Change your deep averaging network to use PrefixEmbeddings instead of WordEmbeddings, and be sure to retrieve the appropriate vectors. However, most of the code should remain the same!

Option 3: Whatever you want! Feel free to brainstorm and use another solution. For example, you could implement a character-level LSTM to produce additional word embeddings.

Implementation and code modifications You can implement this by changing the code you've written in `train_deep_averaging_network` to behave differently when `use_typo_setting` is set to True. You can either train a totally different model (option 2) or change the inference in your current model (option 1) by appropriately threading the argument through your code. Run the modified version with:

```
python neural_sentiment_classifier.py --use_typo_setting
```

For full credit, your method should get at least 74% accuracy on the altered dev set.

Deliverables and Submission

You will upload your code to Gradescope.

Code Submission You should submit both `optimization.py` and `models.py`, which will be evaluated by our autograder on several axes:

1. Execution: your code should train and evaluate within the time limits without crashing
2. Accuracy on the development set of your deep averaging network model using 300-dimensional embeddings
3. Accuracy on the development set with typos, using your typo-aware method
4. Accuracy on the blind test set: this is not explicitly reported by the autograder but we may consider it, particularly if it differs greatly from the dev performance (by more than a few percent)

Note that we will only evaluate your code with 300-dimensional embeddings.

Make sure that the following command works before you submit (for Parts 1 and 2, respectively):

```
python optimization.py

python neural_sentiment_classifier.py

python neural_sentiment_classifier.py --use_typo_setting
```

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)*.