CS378 Assignment 2: Feedforward Neural Network / Sequence Modeling

Due date: Tuesday February 14th at 11:59pm CST

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

Goals This assignment consists of two parts: (1) You will build a simple feedforward neural network model, re-doing the sentiment classification task (as in Assignment 1). (2) You will play with a sequence tagger – the input is a sequence of tokens, and the output is also a sequence of tokens. This diverges from Assignment 1 where the input is a sequence but the output is a single category label. We will build a simple generative model – HMM model, and learn about structured inference techniques, including dynamic programming and beam search, and how to implement these efficiently.

Dataset and Code

The released code will have two parts, named as part 1 and part 2. The data is not freely available typically (it's licensed by the Linguistic Data Consortium), so the code and data bundle is only available on Canvas. Please don't distribute the data. First expand the tgz file. You will submit codes separately for each part on the GradeScope. For the part 1, you will need to install PyTorch, which is described below.

Installing PyTorch You will need PyTorch for this project. **If you are using CS lab machines, PyTorch should already be installed and you can skip this step.** To get it working on your own machine, you should follow the instruction. 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: Deep Averaging Network (50 pts)

In this part, you'll implement a deep averaging network as discussed in lecture and in Iyyer et al. (2015). If our input $s = (w_1, \dots, w_n)$, 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. 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` 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**; this is because the GloVe embeddings (Pennington et al., 2014) 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) learning objective, which we will learn later in the course. We have processed the vectors with respect to the sentiment data, meaning that they do not contain embeddings for words that don't occur in the train, dev, or test data. Basically, this is a filtered version purely for a runtime and memory optimization.

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. 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 `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 learning rate. $\log(0)$ 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 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.
- Google/Stack Overflow and the PyTorch documentation are your friends. Although you should not seek out prepackaged solutions to the assignment itself, you should use the resources out there to learn the tools.

Q1 (50 pts)

(a) 30pts (25 pts autograder, 5pts writeup) 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 model 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. **Briefly describe what you did and report your results in the writeup.**

(b) 15pts (15 pts writeup) 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. **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).

(c) 5pts (writeup) 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? (You do not need to submit these changes to your code).

Requirements You should get least **77% accuracy** on the development set in less than **10 minutes** of train time on a CS lab machine (and you should be able to get good performance in 3-5 minutes). Your final implementation that you submit can be either batched or unbatched, but **you should implement both**.

If the autograder crashes but your code works locally, there is a good chance you are taking too much time. You may need to reduce the runtime below 10 minutes to get the autograder to complete, as the autograder VMs are underpowered and have some variance in performance. Try reducing the number of epochs so your code runs in 3-4 minutes and resubmitting to at least confirm that it works.

Part 2: Sequence Model (50pts)

Data The dataset for this project is the Penn Treebank (Marcus et al., 1993). The tagged sentences are one word/tag pair per line with blank lines between sentences, and the trees are in the standard PTB bracket format. You are given readers for each, so you should not have to interact with these representations directly.

Terminology Symbols in the grammar like S, NP, etc. are called *nonterminals*. Part-of-speech tags (NNP, VBZ, etc.) are a special type of nonterminal called *preterminals*. Actual words (the, eat, cake, etc.) are called *terminals*. A tree therefore consists of a number of nonterminal productions of arity 1 or greater, n preterminals, and n terminals (leaves), where n is the number of words in the sentence.

We will learn more about these later in the course, when we get to "Tree" part. For now, we will only study 'part-of-speech' tags.

Getting started To confirm everything is working properly, run:

```
python pos_tagger.py
```

This loads the data, instantiates a `BadTaggingModel` which assigns each word **its** most frequent tag in training data, and evaluates it on the development set. This model achieves 91% accuracy, since it can correctly tag all unambiguous words such as function words, so it's actually not a bad baseline!

Framework code The framework code you are given consists of several files. We will describe these in the following sections.

`pos_tagger.py` is the driver class. You should be familiar with the general structure by this point. The data pipeline involves calling `read_labeled_sents` from `treedata.py` to read POS tagged sentences out of the tagged data files (`train_sents.conll` and `dev_sents.conll`). `treedata.py` contains preprocessing and data reading code. You will be using `LabeledSentence`, which represents a sentence as a list of `TaggedToken` objects, each of which contains a string word and a string tag.

`models.py` contains two tagging models. `train_bad_tagging_model` trains an instance of `BadTaggingModel`, which assigns each word its most frequent tag in the training set (so "training" just entails counting word-tag pairs). `train_hmm_model` estimates parameters for the HMM and returns an instance of `HmmTaggingModel`.

Please read the comments in `HmmTaggingModel` to understand what is given to you as the output of the training procedure. Let $|U|$ denote the number of tags (tags being NN, JJ, DT, VB, ...), to avoid colliding with the matrix T for transitions. Take note of the shape of each matrix: `init_log_probs` is a $|U| - 1$-length vector, `transition_log_probs` is a $|U| - 1 \times |U|$-sized matrix, and `emission_log_probs` is a $|U| - 1 \times |V|$-sized matrix.

Q2 (30pts autograder) Implement the Viterbi algorithm in the `viterbi_decode` function in `inference.py`. **Report performance in both accuracy and runtime. Your model must get at least 94% accuracy and evaluate on the development set in at most 250 seconds on a CS lab machine.**

Note that your model should be a correct implementation of the Viterbi algorithm: for an arbitrary `HmmModel` and a given sentence, it should return the highest probability path in all cases. You should not hardcode in anything specific to this particular tagset, English, or this particular HMM.

Q3 (15pts autograder, 5pts writeup) Implement beam search for the sequence model in the `beam_decode` function. This is used instead of Viterbi if you pass in the `--use_beam` argument.

`utils.py` contains a `Beam` class if you wish to use it. `Beam` maintains a set of at most `size` elements in sorted order by scores. Note that this implementation uses lists and binary search, meaning that it will not be as efficient as it would be if it used data structures like heaps. However, for most applications in NLP, particularly neural network models, manipulating the beam is not the code bottleneck, rather computing beam elements and their scores is.

Report performance in both accuracy and runtime for three to five different values of the beam size, particularly beam size 1. Discuss how the beam size affect the performance of the model (both in terms of runtime and accuracy). For beam size 3, you should get at least 94% accuracy and evaluate on the development set in at most 50 seconds. You should be able to get this working nearly as well as Viterbi depending on the beam size, so if you're seeing a major performance drop, you have a bug. For this part, we will only grade the results on beam size 3.

Deliverables and Submission

You will submit both your code and writeup to Gradescope. These are submitted as **three separate uploads (two codes, one for part 1 and another for part 2, and one pdf)** to Gradescope.

Written Submission You should upload to Gradescope a PDF or text file of your answers to the questions. This can be handwritten and scanned/photographed if that works best for you. **You can submit the written assignment independently of the code.** If you are unable to get the code fully working, please write up what you did and answer as many questions as possible, even partially, so we can assign you appropriate partial credit.

Code Submission: Part 1 You should submit `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 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).

Code Submission: Part 2 Your code in `inference.py` will be evaluated by our autograder on its execution time and whether it meets the required accuracy values.

Make sure that the following command works before you submit:

```
python pos_tagger.py --model HMM

python pos_tagger.py --model HMM --use_beam --beam_size 1

python pos_tagger.py --model HMM --use_beam --beam_size 3
```

We will evaluate the accuracy of **viterbi** and the accuracy of **beam search**. Lastly, it will run a hidden test on a new HMM to check for correctness of **both Viterbi and beam search** implementation, using multiple different beam sizes including beam size 1. If you fail this third test, come up with a small example yourself and verify that your inference code is computing scores correctly and extracting the correct best path.

References

Mohit Iyyer, Varun Manjunatha, Jordan L. Boyd-Graber, and Hal Daumé. 2015. Deep unordered composition rivals syntactic methods for text classification. In *ACL*.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of english: The penn treebank. *Comput. Linguistics*, 19:313–330.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *EMNLP*.

Credit: The homework is developed by Greg Durrett.