CS 287
Assignment 2: Tagging from Scratch

Due: Monday, Feb 22nd, 11:59 pm

In class we have been discussing an influential neural network model for part-of-speech tagging, based on the paper *NLP (Almost) from Scratch* (Collobert et al., 2011). The goal of this homework assignment is to replicate the central model from this work. At the end of this assignment, you will have created your own state-of-the-art part-of-speech tagger, and built a framework that can be used easily for many other NLP tasks.

Warning: this assignment will be a bit challenging in a way that is different from a standard CS assignment. There is not much code to write here, our implementation of this assignment contains $<$ 150 lines of Lua code; however you will have to run experiments that require several hours of training time . Therefore it is crucially important that all the details (network sizes, preprocessing, features, training) are correct. We will point you to the sections of the paper that contain important information, but it is up to you to test and implement each aspect. Before you start we advise that you get very comfortable with the notes from class, the paper itself, and especially Torch and the `nn` library.

As you complete this assignment, we ask that you submit your results on the test data to the Kaggle competition website at `https://inclass.kaggle.com/c/cs287-hw2` and that you compile your experiences in a write-up based on the template at `https://github.com/cs287/hw_template`.

1 Data and Preprocessing

1.1 Data

The main data for this task is in the `data/` directory. Here is the first training sentence with part-of-speech tags. (This is a classic sentence that any NLP researcher knows by heart. See this obituary on the amazing Language Log, `http://languagelog.ldc.upenn.edu/nll/?p=3594`.).

```
> head -n 20 data/train.tags.text
1 1 Pierre NNP
2 2 Vinken NNP
3 3 , ,
4 4 61 CD
5 5 years NNS
6 6 old JJ
```

```
7 7 , ,
8 8 will MD
9 9 join VB
10 10 the DT
11 11 board NN
12 12 as IN
13 13 a DT
14 14 nonexecutive JJ
15 15 director NN
16 16 Nov. NNP
17 17 29 CD
18 18 . .

19 1 Mr. NNP
```

Each line of the file contains one tokenized word. The columns represent:

1. the global id of the word
2. the sentence id of the word
3. the tokenized word form
4. the part-of-speech tag

Sentences are separated by blank rows. Sentence boundaries are a classic source of errors, so you will have to handle these carefully. In addition we also include a tag dictionary file.

```
> head  data/tags.dict
NNP     1
,       2
CD      3
NNS     4
JJ      5
MD      6
VB      7
DT      8
NN      9
IN      10
```

This is a mapping from each part-of-speech tag in the Penn Treebank to a unique ID. It is important that you use this mapping since it is used for the Kaggle competition data. For those interested, the specification of the mapping from names to descriptions is given at `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`.

1.2 Preprocessing

For this assignment you will write your own preprocessing code in `preprocessing.py`. This code should transform the data representations given above into a format for multi-class classification. In particular you will,

- Clean the input data following Section 3.4
- Pre-construct the features that are fed into the network
- Pre-construct the windows around each word (as described in class)
- Write out a vector of the pretrained embeddings for each word (see below)

The important paragraph from Section 3.4 describes the necessary preprocessing,

> All our networks were fed with two raw text features: lower case words, and a capital letter feature. We chose to consider lower case words to limit the number of words in the dictionary. However, to keep some upper case information lost by this transformation, we added a caps feature which tells if each word was in low caps, was all caps, had first letter capital, or had one capital. Additionally, all occurrences of sequences of numbers within a word are replaced with the string NUMBER, so for example both the words PS1 and PS2 would map to the single word psNUMBER. We used a dictionary containing the 100,000 most common words in WSJ (case insensitive). Words outside this dictionary were replaced by a single special RARE word.

Also note the remark on border words,

> Remark 1 (Border Effects) The feature window (3) is not well defined for words near the beginning or the end of a sentence. To circumvent this problem, we augment the sentence with a special PADDING word replicated dwin/2 times at the beginning and the end. This is akin to the use of start and stop symbols in sequence models.

Since there are only two types of features (word and capitalization), it is likely easiest to simply output two tensors. When done you should output something like the following matrices in an HDF5 group. We ended with something like this, although yours may vary.

> train_input_word_windows, train_input_cap_windows, train_output valid_input_word_windows, valid_input_cap_windows, valid_output, test_input_word_windows, test_input_words, nclasses, nwords, word_embeddings

2 Code Setup

Write your main code in `HW2.lua`. For this assignment part, you can (and should) use the `nn` library in addition to the standard Torch library. You should not need to use any extra libraries.

2.1 Prediction and Evaluation

Be sure you are able to read and output the text data and that you understand the format. Also be sure you are able output your classification results on the test data as a text file. Check that you can upload these to the Kaggle. The Kaggle submission takes in the global id of the word (first column) and the class (1-based). For example

```
ID,Class
1,20
2,24
3,12
```

2.2 Hyperparameters

Several of the models described have explicit hyperparameters that you will need to tune. It is your responsibility to cleanly separate these out from the models themselves and expose as command-line options. This makes it much easier to run experiments and to utilize experimental scripts.

2.3 Logging and Reporting

As part of the write-up, you will need to report on the training and predictive accuracy of your models. To make this possible, your code should report on various metrics of the model both at training and test time. We will leave it up to you on which metrics to log, but we recommend reporting training speed, training set NLL, training set predictive accuracy, and validation predictive accuracy. It is your responsibility to convince us that the model is correctly training.

3 Models

For this assignment you will implement the three models. We will warm up with naive Bayes and multiclass logistic regression, then we will move on to the full neural network model from Collobert et al. (2011), both with and without pre-trained embeddings.

3.1 Naive Bayes

To start, modify your code from HW1 to implement a windowed version of naive Bayes first. This will give you an efficient method that is easy to debug.

Note since the windowed setup has a fixed length, we can use a more appropriate factorization than the last problem. Informally,

$$p(\boldsymbol{x}, \boldsymbol{y} = \delta(c)) \approx p(\boldsymbol{y} = \delta(c)) \prod_{i=-\lfloor dwin/2 \rfloor}^{\lfloor dwin/2 \rfloor} p(\text{word at i in } \boldsymbol{x} | \boldsymbol{y} = \delta(c)) p(\text{capitalization at i in } \boldsymbol{x} | \boldsymbol{y} = \delta(c))$$

That is we can treat each relative window position as a separate multinomial distribution (assuming the word and its capitalization are independent), instead of having a single multinomial over all features.

3.2 Multiclass Logistic Regression

As a second baseline, you will reimplement multiclass logistic regression for this problem. However, for this assignment instead of using your previous code, you will implement it using the `nn` library with mini-batch SGD.

Note the following correspondences,

Math	Torch
sparse $\boldsymbol{xW}$	nn.LookupTable
log softmax	nn.LogSoftMax
$L_{cross-entropy}()$	nn.ClassNLL
dense $\boldsymbol{xW} + \boldsymbol{b}$	nn.Linear

The documentation (`https://github.com/torch/nn`) is quite good for all of these models (although it is possible the notation may be slightly different than class).

The training and evaluation code for the model will take a different (but similar) form from HW1. Here are a couple tips for implementing this code.

- Train using minibatches (we used size 32). This will significantly speed up training.
- Keep track of the average loss as you train.
- Run on the development set each iteration to see the progress of the model.
- You should not have to run for more than 20 epochs on this data. If you are seeing large changes beyond that, you may have an issue.
- We recommend that you first test this code on multiclass logistic regression before moving on to the neural network model.

3.3 Neural Network Model

Finally implement the word-level neural network described in Section 3 of the paper (Figure 1), and discussed in class. The model should not require many more units than for multiclass logistic regression. Basically you should be able to reuse all training and test code, and only change the model itself.

We refer you to the paper itself for the details about the model architecture and layer sizes. The authors are very detailed about these sizes and the experiments that they ran. You should not have to worry about aspects like model initialization since Torch sets reasonable defaults.

One tricky aspect is handling separately-sized embeddings for words and capitalization features. Start with just word-embeddings. To include capitalization we used `nn.ParallelTable`, `nn.JoinTable`, and `nn.View` (although there are other ways to do this.)

3.4 Neural Network Model with Pretrained Vectors

Perhaps the most influential aspect is Section 4 which discusses semi-supervised training of the model. This section incorporates a large corpus of unlabeled data to "pretrain" the word embeddings that are used in the model. This pretraining gives good starting representations, which are particularly useful for rare words in the training vocabulary.

Unfortunately we do not have the computational power or the time to replicate these experiments. The authors note, ”Since training times for such large scale systems are counted in weeks, it is not feasible to try many combinations of hyperparameters” (Although as we will see in the next assignment, that this training time has been brought down significantly.)

Luckily, pre-trained word embeddings are now conveniently available. We will use the vectors learned in the work of Pennington et al. (2014). These are given in the data directory as `data/glove.6B.50d.txt.gz`. The format is one word per line with d_{in} columns,

```
zcat data/glove.6B.50d.txt.gz | head -n 1
the 0.418 0.24968 -0.41242 0.1217 0.34527 ...
```

First, you should align the word features with the vectors in this file in your preprocessing. As part of preprocessing you should output a matrix of embeddings, where each row i contains the embedding for word index i.

Then before running training you should replace the parameters in your lookup table with the word embeddings before training your model. Read the code in `nn.LookupTable` to get a sense of how these parameters are stored and what you should replace.

3.5 Additional Experiments

Once these models are constructed, you should also report on additional experiments on these data sets. We will leave this aspect open-ended, but suggestion include:

- Including additional features (for instance see Section 6 (Collobert et al., 2011))
- Experiment with different optimization techniques. For instance see the `optim` package.
- Experimenting with different architectures. Do more layers help?
- Sometimes it is more convenient to have fixed embeddings (as opposed to changing them during training). Experiment with ”fixing” the embedding layer. (Hint: make sure the embedding layer does not receive gradients.)

4 Report and Submission

For your write-up, follow the report template at `https://github.com/cs287/hw_template`. Be sure to include a link to your code, Kaggle ID, and reports on your results.

In addition to submitting your Kaggle results, we also expect you to report on your experimental process. This should include data tables, graphs and discussion of any issues that you may run into.

References

Collobert, R., Weston, J., Bottou, L., aKrlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 12:2493–2537.

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543.