SIMON D. LEVY

Computer Science Department, Washington and Lee University

CSCI 315 Assignment #6

Due on Github 11:59PM Friday 14 April

Goals

1. Understand state-of-the art recurrent networks by building a Long Short-Term Memory (LSTM) network for part-of-speech (POS) tagging.
2. Learn how neural nets represent word meanings by building a network for dense word embeddings.
3. Get a feel for the latest development in Deep Learning — the attention/transformer networks behind ChatGPT — by deploying and modifying up a simple PyTorch example.

Part 1: LSTM

Read through this tutorial and copy/paste the code into a Python script **lstm.py**. Because of the small problem size, it should be possible to run the whole script (including training) in a few seconds on your laptop.

As usual, we're going to learn more about this model by making some improvements.

First, based on the tutorial's description of the final output (tag scores), add some code to report the output to the more human-readable form described in the large comment (DET NOUN VERB DET NOUN). A nice output would report each word along with the POS label learned by the network, followed by the correct POS label (just as we did all the way back with XOR learning). For example:

```
The:   DET (DET)
dog:   NN ( NN)
ate:   V ( V)
```

```
the:   DET (DET)
apple: NN ( NN)
```

Once you've got that working, factor the code into a test function **`test(words, targets)`**, where **`targets`** are the desired POS tags, that will run this test for either of the two training examples (*The dog ate the apple* or *Everybody read that book*.) At this point you can probably comment-out the `print()` statements on the code before the training/testing part, to avoid distraction. Then add the usual code inside your training loop to report the number of epochs at a reasonable interval.

Now that we've got a nice little training/testing script set up, let's add some more parts of speech (POS) to our problem. Adjectives seem like a natural place to start: How about: *The big dog ate the red apple* and *Everybody read that awesome book* ? Feel free to come up with your own vocabulary (bonus points for making me laugh!)

Of course, we haven't really done an honest training/testing evaluation of our model, because we're training and testing it on the same data set. To see how the model works on data it hasn't seen, try passing a couple of new sentences to your test function, by making new sentences from the existing vocabulary.

Applying some critical thinking to our results so far, you can probably see that they aren't all that impressive: all we've got it is a model that classifies data (words) that it's already seen! Indeed, you could probably do this with a simple logistic-regression or classic perceptron model using a single layer of weights and no recurrent (feedback) connections. So what's the big deal about LSTM and other recurrent networks?

Well, as we discussed in lecture, the cool thing about these networks (going back to Elman's 1990 Simple Recurrent Network) is that they can *predict* the identity or category of the next item, based on the items seen so far; e.g., you know that the next word after *The dog ate the* … must be a noun or adjective. What's more, you can use this approach to solve "Plato's Problem" of deducing information about words you have never heard: when you hear *The dog ate the knish*, you can immediately infer that *knish* is a noun (and probably something edible!) So, to finish up Part 1 of the assignment, figure out how to add a new word to the vocabulary, run the training on the same sentences as before (i.e., sentences not containing that new word), and see what your test function does on a sentence containing the new word.

Part 2: Word Embeddings

As in the previous part, read the PyTorch tutorial on Word Embeddings, copy/pasting the code into a script **embedding.py**. And again, comment-out the annoying print statements, except for the final one reporting the embedded vector value (tensor) for one of the words. Next, replace that final print with a loop that prints each vocabulary word followed by its vector, without the distracting `tensor .. grad_fn` wrapping. After "a little help from my friends" on StackOverflow, I was able to use numpy formatting to get an attractive printout like this, sorted in alphabetical order:

```
'This   [+2.295 +0.676 +1.714 -1.794 -1.521 +0.918 -0.549 -0.347 +0.473
-0.429]
And     [+0.487 -0.309 -3.014 -1.247 +1.349 +0.269 -1.128 -0.601 +1.837
-1.071]
…
youth's [+2.414 +1.021 -0.44  -1.734 -1.026 +0.521 -0.453 -0.126 -0.588
+2.119]
```

Note that because of the simple way that the text was split (via the default blankspace delimiter), we're getting bogus punctuation included with some words (like the quotation mark in `'This`) — which as Shakespeare might say, doth vex me somewise! Pythonistas have lots of tricks for getting rid of punctuation, but for the current project I think it's simply easier to either (a) not worry about the problem, or (b) edit the sonnet fragment a bit to eliminate punctuation and upper-case letters. I chose the either (b), which gave me a final vocabulary size of 86 lower-case words.

Now that we've got a nice little word-embedding program with sensible output, let's see whether we can understand the embeddings (vectors) that it's giving us. As we saw in the lecture on the Simple Recurrent Net (slides 13-16), a clever way of doing this is to build a distance matrix, then run Hierarchical Cluster Analysis on the distance matrix to build a dendrogram (tree diagram) to visualize the semantic structure encoded in the embeddings.

Fortunately, there are now powerful tools that can do both of these steps for you automatically. This page has a schweet example. One I got this example running, I printed out the tiny dataset used for the clustering and saw that it was simple a list of 2D vectors (represented as tuples). After puzzling over how to go from that kind of 2D data to our ten-

dimensional embedding vectors, I figured I'd just put the vectors into a big list and use them as the data. Sure enough, it worked! A little more googling revealed how to use my Shakespeare vocabulary as the labels for the dendrogram, and then how to rotate the dendrogram so that the labels appeared on the left rather than at the bottom, for greater readability.

After all that work, I found my dendrogram results somewhat disappointing: with so many words, it was impossible to read the whole plot clearly, and when I tried zoom in on it, I found it difficult to discern the kinds of word-class patterns that Elman got. As is often the case in science (esp. data science), your results can be sensitive to not only the algorithm you use, but also the data! In other words, if I could go back to the simpler, artificially-generated sentences used by Elman, I might see some kind of pattern in my own embedding results.

As usual, googling a bit for RANDOM SENTENCE GENERATOR PYTHON, I found a simple solution on StackOverflow. Even better, I realized that instead of using random numbers, I could simply enumerate every possible sentence of the form ADJ NOUN VERB ADVERB (e.g., *adorable rabbit runs occasionally*, by a quadruply-nested cascade of `for` loops. This solution had the advantage of having a very small vocabulary size (20 words) and a much much larger data set (5^4 four-word sentences) than the original sonnet fragment. So, for the final part of the assignment, I added a little code to my embedding.py to generate these simple sentences and use them as the training set. By decreasing the context size and number of embedding dimensions, I was able to get reasonable (not perfect!) dendrogram results after 100 epochs. Try that, see what you get, and include your dendrogram picture in your writeup.

Part 3: Attention / Transformers

I was unable to do this compute-intensive part this on my computer at home, so I suspect you will have to do it on the lab machines. As before, I've installed all the necessary Python packages beforehand, so you can ignore the instructions telling you to install a particular package. Please let me know however if something appears to be missing on the machine you're using!

Here is the code to copy/paste/modify into your initial **transfomer.py** script. This time the PyTorch folks did a nice job formatting the training reports — including time info! Unfortunately they did not include a more detailed test case (like our confusion matrices

from the previous assignments); hence, as before, we have an opportunity to explore further.

First, as before, let's do the easy thing and see how much value we get from the GPU, by finding the `device = ...` code, commenting it out to force CPU, and then running a trial with and without CUDA. As before, if you run the code with `time /usr/bin/python3` instead of just `/usr/bin/python3`, you can get a nice overall time summary at the end, to include in your writeup.

Next, let's see what this model is actually learning! I found the tutorial description pretty minimal, so as usual I started printing things out and exiting before the training started. By printing out the size (`len`) of various data variables in the code, I quickly got a confirmation that this is indeed an model of the English language. (Hint: take a look at this statistic). Then the sizes of the training and testing sets then made sense too. Make sure to note these three sizes and report them in your writeup, with a brief explanation. Also comment in your writeup on a new "one weird trick" you can see in the training report!

Now that we've got a good sense of what kind of data this model is using and how long it takes to train, let's do the usual thing and break it up into training and testing scripts. Unfortunately the code saves the pickled model **best_model_params.pt** in some weird temporary directory that I couldn't locate, so the first modification I made was to force it to save that file in the current directory, with a helpful message about saving the file, as we did in the previous assignments. Once you've got the model saved, comment briefly in your writeup on the number of apparent parameters (floating-point weights) it appears to contain, assuming the standard four-byte floating-point encoding.

So now it's time to split up our code the usual way: **transformer.py** (class definition), **transformer_train.py** (train model and save it), and **trainsformer_test.py** (load model and test it). Because of the way that the original transformer script mixes up global variables and function parameters, this step can take a while to get right, but at the end you'll have a standalone test script that you can use to try out your *P*re-*T*rained transformer: the PT part of ChatGPT!

As it stands, the `evaluate()`function used by the training and testing scripts doesn't report anything interesting; it just returns the loss value. So to get a better idea of what the network is actually doing, I copy/pasted the `evaluate()`function into my test script to create a new function `report()`, which I then modified to report the actual input and

target words. As mentioned in the (confusing to me!) tutorial instructions, the job of the `get_batch()` function used in `evaluate()`is to make a target sequence out of the input sequence by shifting the input sequence by one position — the same trick as in Elman's original 1990 sequence-learning model.

To verify this claim for the actual vocabulary in the data, it took a little bit of experimental printing, to see that the data (input) and targets were both of size 35×10, but that the targets had been reshaped to 1×350. Once I figured that out, I was able to reverse-engineer the vocabulary object to extract the words corresponding to each word index, and then write some code to report the data words followed by the target words. *Hint*: as usual, `type()` will tell you the type of a variable, after which you can look up its methods in the online documentation. For the first iteration of the `report()` loop I got this output (abbreviated here for simplicity), showing that the inputs and targets had the expected relationship:

```
= next either imagery and her = . was hitting
robert day blunt and n death boston seneca proscribed the
<unk> it ( clear @-@ as celtics asked . slow ...

robert day blunt and n death boston seneca proscribed the
<unk> it ( clear @-@ as celtics asked . slow
= joined <unk> , <unk> it = <unk> these @-@ ...
```

Looking at this data, I still couldn't make any sense of the individual lines: WTF is `= next either imagery and her = was hitting` ... supposed to mean?! As a final effort at understanding this complicated model, I managed to find the URL for the WikiText-2 dataset zipfile, hidden in the PyTorch source code. Downloading and unzipping this dataset and looking through the testing part, I solved the final mystery! In your writeup, briefly comment on what you find when you do this; i.e., how is the code representing the actual text?

What to submit to Github

As usual, your PDF writeup will be the main part, plus the Python scripts to preserve your work.

Extra-Credit Opportunities

The tutorials for the first two parts have an exercise at the bottom. Although I did not attempt these myself, you should feel free to try one or both for some extra credit.

For me, the remaining mystery of the attention / transformer exercise was the `output` produced in our `evaluate()` and `report()` functions. This output appears to be logits (net inputs to the softmax layer), so an interesting task would be to see whether we can convert them to word indices and see what the network is actually outputting. I'm continuing to work on that!