CS695-002 Assignment 1: Welsh and Irish POS Tagging

Antonis Anastasopoulos

Due: EOD 9/18
Value: 10 credits (available: 12)

Deliverables

Submit a PDF with your report on the assignment, along with all the code you wrote, the models that you trained, and the outputs of your models on the dev and test that are needed for obtaining the results. Answer the questions under the sections that have credits assigned to them as best as you can. In your report, include the terminal commands that we would need to run, using your code, to obtain the same outputs and results. **I will look into and run your code. If you use code snippets from StackOverflow or from other people's code that you find online, appropriately mention it in the report with a citation and/or in the code through a comment.**

Submission Information: Zip your code, outputs, and your report PDF in a single file under your name (e.g. mine would be `antonisanastasopoulos.zip`, and email it to the instructor at `antonis[at]gmu[dot]edu`. Deadline is end-of-day 9/18. You can use your late days, but I highly recommend you save them for the later assignments.

Part-of-Speech Tagging

Part-of-Speech (POS) Tagging is the task of annotating every token in a sentence with its grammatical category. For example, a tagging of English sentence "the cat sat on the mat" would be `DET NOUN VERB PREP DET NOUN`, where `DET` stands for 'determiner' and `PREP` stands for 'preposition'.

Although the proper categorization for some languages is still debatable, in this assignment we will work within the Universal Dependencies (`https://universaldependencies.org/`) formalism, which uses the universal part-of-speech tags defined by Petrov et al. [1].

Data

We will work with the Irish (`https://universaldependencies.org/treebanks/ga_idt/index.html`) and Welsh treebanks (`https://universaldependencies.org/treebanks/cy_ccg/index.html`) from the Universal Dependencies project. The `data` directory of the assignment package includes training, development, and test files, following the following format:

- Each line corresponds to a sentence
- Each line provides the tokens and their corresponding POS tags in the following format: `word|tag`

The Irish train file includes 2019 sentences, the development and test sets have 451 and 454 respectively. The Welsh training file has 614 sentences, while the test file has 953 sentences (there's no Welsh development set).

Code

The assignment package provides the following scripts:

- A utilities `utils.py` script with basic data reading functions that are used by the other scripts.
- A basic LSTM model `pytorch_tagging.py` implemented in PyTorch (based on this tutorial: `https://pytorch.org/tutorials/beginner/nlp/sequence_models_tutorial.html`).
- A simple script to compute accuracy given two files in the above format. Run as: `python compute_accuracy.py [your_output] [gold_file]`

Prerequisites You should be able to easily install PyTorch locally by following the instructions here: `https://pytorch.org/get-started/locally/`.

1. Baseline Results [4 credits]

The example script has toy data hard-coded into it. You can modify it to read the provided data in the desired format by doing the following:

```
TRAINING_FILE = "data/irish.train"
training_data = utils.convert_data_for_training(utils.read_data(TRAINING_FILE))
```

1.a [1 credit] Model Read/Write The baseline script does not implement storing and loading the trained models. Implement this functionality, so that you don't have to train your models from scratch every time you run your script (and so that I can load your models and immediately obtain outputs when running your script).

1.b [1 credit] Unknown Words The baseline script has a very severe limitation: it can only handle input words that appear in the vocabulary (the dictionary `word_to_ix` in the script). However, there's no guarantee that the test set will only include these words! In fact, you can be sure that there are words in the test set that do not appear in the training set. One way to deal with this is to use an `UNK` token that we will use to represent rare words. The `utils.py` includes the skeleton for a `substitute_with_unk()` function, that you can use to modify the corpus. After training with the rare words substituted with `UNK`, we can easily use our model on the test set (as long as we also substitute the unknown words – the words that are not in our training vocabulary – with `UNK`). Implement this functionality, and train and evaluate a model using UNK for words that appear only once in the training set. What accuracy does your model achieve on the Irish development set after training for 20 epochs? What happens if you use two copies of the training set, one with UNKs and one without, as your training set? This should increase your vocabulary coverage while also learning to properly handle unknown words.

1.c [1 credit] Early Stopping The baseline script does not utilize the Irish development set. As a result, we face the unfortunate potential of *overfitting* on the training set, which can lead to much worse performance on the evaluation set. One way to combat overfitting is through *early stopping*. The idea is simple: while training, we will periodically[1] check the performance of our current model on the development set. If the accuracy is better than anything we have achieved before, then we keep training. If a few epochs pass and we still are not improving (e.g. after 8 epochs of no improvement. This is called "patience" in most toolkits) then we stop training and we use the model that achieved the best dev accuracy as our final model. Implement early stopping. After how many epochs did your model achieve its best performance?

Note: A good practice is to keep track of both the training loss and the development set loss. The training loss should always decrease. The development set loss will eventually plateau and then start increasing again, as you start overfitting. Visualizing the training process by plotting the losses is a great way to keep track of what's going on.

1.d [1 credit] Batching Our dataset is small enough that we can train one instance at a time. However, for larger datasets it is always better to batch sentences together, so that we process them in parallel taking advantage of large matrix multiplications on GPUs. You will most likely need to use batching for your projects, so this is important! PyTorch supports automatic batching through the DataLoader modules (`https://pytorch.org/docs/stable/data.html`), or you can follow the example from this medium post: `https://towardsdatascience.com/taming-lstms-variable-sized-mini-batches-and-why-pytorch-is-good-fo`

[1] e.g. after every epoch or after every 100 update steps or something like that.

Implement batching. Does your model train faster? Does the final performance change?

Hint: Without any additional modifications, and using an embedding and hidden dimension of 32 and training for 20 epochs with a batch size of 1, I got a test set accuracy of around 74% on Welsh and around 77% on Irish. What is the best accuracy you obtained?
To submit: code that implements the above four items, and any development set or test set outputs, under a directory named "part1".

2. Improving the Baseline [6 credits]

We will now improve upon this simple baseline:

2.a [1 credit] Incorporate context from both sides The current recurrent model only passes through the sequence in the left-to-right manner. This means that when making a prediction for position j, we have only taken into account the left context from previous positions $0 \dots j$. However, incorporating the right context from the positions $j+1 \dots$ can be very informative and aid the model in making a more informed prediction. This can be done by running another LSTM over the sequence in a right-to-left manner, and then combining the outputs of the two LSTMs. Implement this in your model, either by:

- making a copy of the input sequence, reversing it, passing it through a second LSTM, and then adding the outputs of the left-to-right and right-to-left LSTMs OR
- taking a look into the LSTM PyTorch module (`https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html`) which already implements it.

Note: the second option is very, very easy. I strongly suggest you try the first option first!

What performance do you get on the Irish development set by incorporating the right-hand-side context into your models?
To submit: code that implements the above, and any applicable dev/test set outputs, under a directory named "part2a".

2.b [2 credits] Optimizing training and hyper-parameters The baseline model is very simple, and small. What happens if you increase your model's size? Try out the following, and report and changes in the Irish development set performance:

1. Stack two or more BiLSTM layers instead of using a single layer.
2. Change the size of the embeddings or the hidden sizes of the LSTMs.

3. Dropout [2] is a way to regularize your neural network, in order to avoid overfitting. In low-resource settings, like the one we have here, it usually helps boost performance. Good dropout values are generally in the range between [0.1,0.4], depending on the setting.
4. Substitute the simple SGD optimizer (line 82 in the original script) with the Adam optimizer (`https://pytorch.org/docs/stable/optim.html?highlight=adam#torch.optim.Adam`)

Note: no need to perform extensive hyper-parameter search; 2-3 different settings should be enough.

What is the best performance you achieve? What is your takeaway of the best hyper-parameter setting? What about convergence time (i.e. does the model achieve its best performance faster)?
To submit: No need to submit code for this part. You can include your best Irish dev set outputs under a directory named "part2b".

2.c [2 credits] Incorporate Character Embeddings A good way to handle unknown (OOV) words, rather than just substituting them with "UNK", is to represent them using the characters that make them up. There's several ways you can implement this. Choose at least one of the following (or, try all of them to see what works best!):

- Define character embeddings (you'll need to define a character vocabulary) and treat the UNK words as a sequence of characters. Then you can define a character-level LSTM, which will read each word one character at a time, and the final representation of the word will be the last output.
- Define character embeddings (you'll need to define a character vocabulary) and treat the UNK words as a bag of characters. That is, the embedding for the word `ysgol` will be equal to the sum of the character embeddings $e_{\text{ysgol}} = e_y + e_s + e_g + e_o + e_l$. You can extend this to cover character n-grams, so that: $e_{\text{ysgol}} = e_y + e_s + e_g + e_o + e_l + (e_{ys} + e_{sg} + \ldots + e_{ol}) + (e_{ysg} + \ldots + e_{gol})$.
- Pre-segment all rare or OOV words at the character level, and now you'll have a sequence
- Note: You can implement the above solutions either just for UNK words, or for all words.

To submit: code that implements the above, and any applicable dev/test set outputs, under a directory named "part2c".

2.d [1 credit] Using pre-trained embeddings Facebook has publicly released `fastText` embeddings (which incorporate the character n-grams we discussed above) that have been trained on large amounts of data. Details here:

`https://fasttext.cc/docs/en/crawl-vectors.html`; the Irish and Welsh pre-trained embeddings can be downloaded in the same webpage. Use the fastText utilities to obtain vectors for all words, including the the out-of-vocabulary ones from the development and the test set (see instructions in the fasttext page on how to do that). Modify your code to read in these pre-trained embeddings, instead of using random initialization. Do you see any improvements in your test set performance?
To submit: code that implements the above, and any applicable dev/test set outputs, under a directory named "part2d".

3. Bonus: Multilingual Models [1-2 credits]

Irish and Welsh are related to each other. They both belong in the Celtic branch ("genus") of the Indo-European language family. What happens if you train a multilingual model that can perform POS tagging on both languages? Does the performance improve?
Disclaimer: I haven't tried this for these exact data I'm sharing with you, so I don't know if the performance will improve or not.

Hint 1: Note the imbalance in training data sizes.
Hint 2: You don't have to share all parts of the model between the two languages. For example, you could use different embedding matrices for different inputs. Or you could use different linear layers depending on the input language – that's the beauty of dynamic frameworks!
To submit: code that implements the above, and the Irish development set outputs and Irish and Welsh test set outputs, under a directory named "part3".

References

[1] Slav Petrov, Dipanjan Das, and Ryan McDonald. "A Universal Part-of-Speech Tagset". In: *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*. Istanbul, Turkey: European Language Resources Association (ELRA), May 2012, pp. 2089–2096. URL: `http://www.lrec-conf.org/proceedings/lrec2012/pdf/274_Paper.pdf`.

[2] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.