Assignment 2 – Window-based Tagging

Yoav Goldberg

In this assignment you will use a framework that supports auto-diff (computation-graph, backprop) in order to compute gradients, and implement a sequence tagger with window-based features. You will also experiment with word pre-trained embeddings, and with different word representations, on two different tagging tasks.

Part 1 (25 points) – A simple window-based tagger

You will implement a sequence tagger, where the input is a sequence of items (in our case, a sentence of natural-language words), and an output is a label for each of the item. For example, the tagging could be mapping each word to its part-of-speech:

Input:	The	black	fox	jumped	over	the	lazy	rabbit
Output:	DT	JJ	NN	VBD	IN	DT	JJ	NN

Or tagging words with the named-entity types:

Input:	John	Doe	met	with	Jane	on	Tuesday	in	Jerusalem
Output:	PER	PER	O	O	PER	O	TIME	O	LOC

The first task is called POS-tagging,[1] and the second is called named-entity-recognition (NER), and our labels are PERson, LOCation, ORGanization, TIME, and Outside (not a named entity).[2]

The tagger will be greedy/local and window-based. For a sequence of words $w_1, ..., w_n$, the tag of word w_i will be based on the words in a window of two words to each side of the word being tagged: $w_{i-2}, w_{i-1}, w_i, w_{i+1}, w_{i+2}$. 'Greedy/local' here means that the tag assignment for word i will *not depend* on the tags of other words.[3]

More concretely, you will implement the following model: When tagging the word w_i, each word in the window will be assigned a 50 dimensional embedding vector, using an embedding matrix E. These embedding vectors will be

[1] You can find the list of tags and their meaning here: `https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html`.

[2] In practice, we often don't that with these labels, but with BIO-tags, as discussed in the Intro to NLP course. But for this assignment lets assume that this is the tagset.

[3] This choice is not ideal when tagging sequences, but this is what we will implement.

concatenated to each other, and fed into an MLP with one hidden layer and a tanh activation function. The output of the MLP will be passed through a softmax transformation, resulting in a probability distribution. The network will be trained with a cross-entropy loss. The vocabulary of E will be based on the words in the training set (you are not allowed to add to E words that appear only in the dev set).

Implement the network, and train it on the NER and on the POS data for several iterations. Measure your accuracy on the DEV data at each iteration. Experiment with several network configurations, learning rates, etc, until you reach an accuracy you are happy with.

Note: When evaluating the accuracy of the NER tagger, the accuracy could seem very high. This is because most of the words are assigned the 'O' tag. A tagger that tags every word as 'O', will already have an accuracy of 83% (but will be useless). So, when evaluating your result on the NER data, use a different metric: ignore all cases where the correct tag is 'O' and the tagger also said 'O', and then compute the accuracy (percentage of words that are assigned the correct tag).

Considerations: Think of how to handle the following: what do you do with a word that appears in the train set and not in the dev set? which word embeddings will you assign to it? What vectors will you use for the words surrounding the first word in the sequence? And for the last word?

What to submit:

- A pdf file called `part1.pdf` in which you describe the parameters of your best model for each of the tasks (NER and POS), as well your solutions to the "considerations" above. Include also two graphs showing the accuracy of the dev set for each of the task as a function of the number of iterations, and two graphs showing the loss on the dev set for each of the tasks as a function of the number of iterations.
- Your code, in a file called `tagger1.py` (additional files are possible)
- a (plain text) README file, explaining how to run your code, and which are the relevant files.
- Predictions of your best model on the blind test sets, in files called `test1.pos` and `test1.ner`.

The `pdf` file should include your name and ID number.

Part 2 (10 points) – External word embeddings

In part 1, the word embedding vectors were initialized randomly. We will now experiment with pre-trained word embeddings. Download the pre-trained embeddings. The format is two files, one of them containing the vectors themselves

(each line in the file is a vector) and the other files contains the corresponding vocabulary words (each line is a word). Note that the vocabulary file is lower-cased, it does not include any upper-case words.

You can read the vectors file into a numpy array using:

```
import numpy as np
vecs = np.loadtxt("vectors_file_name")
```

To get a feel of the embeddings, write a function `most_similar(word, k)` that gets a word and returns the k most similar words to it according to the cosine:

$$\texttt{sim}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\sqrt{\mathbf{u} \cdot \mathbf{u}}\sqrt{\mathbf{v} \cdot \mathbf{v}}}$$

where $\mathbf{u}$ and $\mathbf{v}$ are d-dimensional vectors, and $\cdot$ is a dot-product.

Compute the 5 most similar words to the following words, together with their distances: `dog`, `england`, `john`, `explode`, `office`.

What to submit:

- A pdf file called `part2.pdf` with the words, the top-5 most similar words, and the distances to the top similar words.
- Your code, in a file called `top_k.py` (additional files are possible)

Part 3 (20 points) – Adding the external word embeddings to the tagger

Change your tagger code to use the pre-trained word embeddings. (You will probably need to think of the following: what to do with words that appear in the training file but not the embeddings file? what to do with the embedding vocabulary being lower-case?)

Train a NER tagger and a POS-tagger. Did accuracy improve over the tagger without the pre-trained embeddings? by how much?

What to submit:

- A pdf file called `part3.pdf` in which you describe the parameters of your best model for each of the tasks (NER and POS), as well as a description of the logic of using the pre-trained vectors (how do you handle lower casing, what do you do with words that are not in the pre-trained vocabulary). Include also two graphs showing the accuracy of the dev set for each of the task as a function of the number of iterations, and two graphs showing the loss on the dev set for each of the tasks as a function of the number of iterations.

- Your code, in a file called `tagger2.py` (additional files are possible). It is also possible to add the code to the tagger in part 1, and having the embedding/no-embedding behavior controlled with a commandline switch. In fact, this is more elegant. If you chose to do so, you do not need to submit the file `tagger2.py`, but please make it clear in the README that this is what you were doing.
- a (plain text) README file, explaining how to run your code, and which are the relevant files. This can be the same README as in part 1, but indicate clearly the relevant parts.
- Predictions of your best model on the blind test sets, in files called `test3.pos` and `test3.ner`.

Part 4 (20 points) – Adding sub-word units

Your taggers do not have access to the form of the words. Knowing something about the letters of the word can help determine its tag even if it was seen only a few times, or even not seen at all in the training file.

One way to incorporate sub-word information (which you are required to implement) is to assign an embedding vector to each prefix of 3 letters and each suffix of 3 letters (among the prefixes and suffixes that were observed in the corpus). Then, you can represent each word as a **sum** of the word vector, the prefix vector and the suffix vector.

Implement the above method, and test how it behaves with and without the use of pre-trained embeddings for the two tagging tasks.

If you think of other ways of incorporating sub-word units, you are most welcome to try them as well. Describe your efforts (and results) in the pdf file.

What to submit:

- A pdf file called `part4.pdf` in which you describe the parameters of your best model for each of the tasks (NER and POS). If you had to make any choices when including the sub-word units, describe them also. For each of the conditions (subword-units, pre-trained) and (subword-units, no pre-trained), include two graphs showing the accuracy of the dev set for each of the task as a function of the number of iterations, and two graphs showing the loss on the dev set for each of the tasks as a function of the number of iterations. **Write also** a brief analysis of the results. For example – which of the pre-trained embeddings and the subword units is more useful, and why? are their contributions complementary? are the trends consistent across the different tagging tasks? why?
- Your code, in a file called `tagger3.py` (additional files are possible). It is also possible to add the code to the tagger in part 2, and having the subwords/no-subwords behavior controlled with a commandline switch. In fact, this is more elegant. If you chose to do so, you do not need to

submit the file `tagger3.py`, but please make it clear in the README that this is what you were doing.

- a (plain text) README file, explaining how to run your code, and which are the relevant files. This can be the same README as in part 2, but indicate clearly the relevant parts.

- Predictions of your best model on the blind test sets, in files called `test4.pos` and `test4.ner`.

Part 5 (25 points) – Convolution-based sub-word units

We will now replace the sub-word embeddings from above with a method based on CNNs, with the architecture described by Ma and Hovy 2016.[4]

The architecture is the following: each word is represented as a max-pooled vector over the vectors resulting from a convolution over the word's characters (see Figure 1 in the Ma and Hovy paper). Combine the pre-trained word-vectors and the character-based CNN vectors via concatenation.

Start with the initialization, number of filters, and window-size as appears in the Ma and Hovy paper. How does this method compare to what you implemented in Part 4? What happens when you try fewer filters? When you try more filters? What happens when you try different window sizes?

Try andd analyze the filters learned in the NER experiment and in the POS experiment. Can you identify meaningful learned filters? Are the filters learned for POS and for NER different, or are they similar?

Describe your efforts and results in the pdf file. For the analysis part, in addition to answering the questions, describe also your method for identifying the meaningful filters.

What to submit:

- A pdf file called `part5.pdf` in which you describe your efforts and answer the questions above.

- Your code, in a file called `tagger4.py` (additional files are possible). It is also possible to add the code to the tagger in part 2 or 3, and having the subwords/cnn-subwords/no-subwords behavior controlled with a commandline switch. In fact, this is more elegant. If you chose to do so, you do not need to submit the file `tagger4.py`, but please make it clear in the README that this is what you were doing.

- a (plain text) README file, explaining how to run your code, and which are the relevant files. This can be the same README as in part 4, but indicate clearly the relevant parts.

[4] `https://arxiv.org/pdf/1603.01354.pdf`

- Predictions of your best model on the blind test sets, in files called `test5.pos` and `test5.ner`.

The content in the `pdf` files is important! Invest in writing clear and complete documents.