Deep Learning in NLP

Practical Session P8

Introduction

Our goal is to implement a simple LSTM-based POS tagger, which we will apply to universal dependencies (UD). The plan is to focus on the POS tagging task during ≈ 3 sessions. The long-term goal is to extend our program to predict dependencies.[1]

The implementation of the POS tagger will be split into a series of exercises, which we will be doing (mostly) during the practical sessions.

Starting from now, we will use higher-level PyTorch API. In particular:

- We will use the Module class from PyTorch rather than our home-baked version. See Appendix A for a summary of differences between the two.
- We will use PyTorch datasets and data loaders.
- We will not implement LSTM ourselves, we will use the version available at `https://pytorch.org/docs/master/nn.html#torch.nn.LSTM`.

Preparation

Download the code from the course webpage, unpack it, and open in VSCode (or your other favorite editor). The archive `.zip` file also contains the dataset[2] we will be working with. We will focus on English for now, but the model we implement should be generic enough to work with the majority of UD languages.

Git

If you want to use `git` to keep track of changes in the `hhu-dl-materials` repository, see the Appendix B at the end of this document.

The code for the POS tagging task is in the `universal-pos-deps` directory of the repository. Note that the github repository only contains code, you will have to download the datasets separately.

[1]The plan is to implement something along the lines of `https://arxiv.org/pdf/1611.01734.pdf`.

[2]See `https://universaldependencies.org/conll18/data.html`

Model

The model can be summarized as follows. For a given input sentence:[3]

- **Embedding**: Replace each word in the input sentence with the corresponding vector representation.
- **LSTM**: Use LSTM to obtain the hidden, contextualized input word embeddings.
- **Scoring**: Apply a linear layer to score the hidden vectors and transform them to POS tags.

Today we focus on the embedding part. Once we succeed in implementing the full model, we will consider more advanced embedding strategies (`CharCNN`, `CharLSTM`), pre-trained embeddings (e.g., `FastText`), etc.

Exercise 1

The Embedding module provided by PyTorch only works with integer keys. The goal of this exercise is to encapsulate the PyTorch's Embedding module in our own embedding module which maps generic keys (strings, tuples, etc.) to embedding vectors.

Note that we already implemented such a custom Embedding module before, but now the task is to implement it in terms of the PyTorch's Embedding module.

Place the solution to this exercises in `neural/embedding.py`, so we can use it in Ex. 2.

Exercise 2

Based on `neural/embedding.py`, implement a word embedding module which maps words to embedding vectors. Place the implemention in the `word_embedding.py` file.

Why should we have two separate emedding-related modules?

- `neural/embedding.py` is abstract, it can be used as a sub-module for different tasks (e.g., language prediction for person names).
- `word_embedding.py` is specifically designed for the POS tagging task at hand. We will place there more advanced word embedding modules later.

The word embedding module should:

- Consider words as atomic units. For instance, *cat* and *cats* should receive two unrelated embedding vectors.
- Optionally, conflate words with difference letter case, so that (for example) capitalized *Cat* and lower-case *cat* receive the same embedding vector.

[3]We assume pre-determined tokenization.

Finally, create an instance of the implemented word embedding module based on the given UD dataset. Verify in IPython that you are able to embed words (also out-of-domain words).

Exercise 3

Calculate the set of POS tags (the *tagset*) in the dataset. You can do that in IPython, but you will also need the tagset to create the POS tagging module.

Investigate the tagset and make sure it corresponds to the UD POS tagset (`https://universaldependencies.org/u/pos/`). If there are discrepancies, propose and implement a way to deal with them.

A. PyTorch Modules

In contrast with our own version of the `Module` class:

- You must use `super(ParentClass).__init__()` at the beginning of the initalization method.
- No need to register sub-modules (just assign them to attributes in the initialization method).
- In case you want to use a raw tensor as a module's parameter, wrap it in the Parameter object. Then, you have to additionally register it.

The PyTorch Module is slightly less flexible than the class we implemented, but it is also more powerful. For instance, it keeps track of the state of the module: training or evaluation. You can access this information via the `training` attribute in the main module and any of the sub-modules.

B. git

Instead of downloading the code manually, you can use `git` to keep track of changes in the `hhu-dl-materials` repository.

On linux, you will first have to clone the repository to a local directory:

```
git clone https://github.com/kawu/hhu-dl-materials
cd hhu-dl-materials
```

Then, each time you want to sync with the github repository:

```
git pull
```

The command above has to be run in the local `hhu-dl-materials` directory.