Deep Learning for NLP

Assignment 3: Language Identification using Recurrent Networks

Sharid Loáiciga
(based on content from M. Silfverberg and H. Celikkanat)

1 Introduction

In this assignment, you will be working on a language identification task with several recurrent architectures, as compared to the basic feed-forward architecture used before. The aim is to guess the language of a given word, among 6 different languages from the Uralic family.

2 Language identification using RNNs

We will train a recurrent layer, which consumes a sequence of characters as an input, and outputs a state representation of this sequence, the context embedding. This is usually the output of the network at the last time step. On top of this recurrent layer, we will implement a feed-forward component, which takes this state representation as input, and outputs the associated class (language) of this sequence as one of the 6 possible language classes. This is known as a many-to-one architecture, consuming a sequence of inputs, and giving out a single output eventually. Our model is composed of three sub-parts:

2.1 The character embedding layer

The first layer takes in the characters in our sequence as integers, and encodes them into their corresponding high-dimensional embeddings.

2.2 The recurrent layer

This layer takes the embedding for each input character of the sequence, and processes them internally one by one, while updating its internal state according to each incoming character. This internal processing will be handled by the relevant PyTorch module. Eventually, it is going to output its current state and the complete history of its past states. As the recurrent layer, we are going to use three different options: 1. an `LSTM`, 2. a `GRU`, and 3. a `RNN`.

You can find all these different layers readily implemented within the `torch.nn` module. Note that we will be using the more "abstract" classes e.g., `LSTM`, and not the more "lower-level" classes called `XCell` e.g., `LSTMCell`. The difference between the two is this: The cell corresponds to one step of a long-short term memory computation, it takes in one input, updates its step once, and produces one output. This is useful, when you want to customize the inputs and outputs of the layer. On the other hand, the `LSTM` class runs over an entire sequence from beginning to end, and gives us the outputs at each time step and the final hidden state. Using the `LSTM` will enable us to let PyTorch handle the entire time steps, and we will instead just feed it the input, and eventually collect the corresponding output.

2.3 The classifier (linear layer + log_softmax)

On top of the recurrent layer, we are going to implement a fully-connected feed forward layer, which takes in the output of the recurrent layer as its input, and returns 6 different outputs corresponding to the 6 different classes. We will finally pass these outputs through a `log_softmax` function to receive the log probabilities. This part is similar to the classifiers we have been implementing so far.

3 Getting started

First download and unpack the archive `assignment3.zip`. This results in a directory `assignment3` with the following structure:

```
- data/
--- uralic.dev
--- uralic.mini.dev
--- uralic.mini.test
--- uralic.mini.train
--- uralic.test
--- uralic.train
- src/
--- assignment3.py
--- data.py
--- paths.py
```

The directory `data` contains the Uralic language training, development and test sets. You can use the "mini" versions when developing your model, to progress faster. The skeleton code for the language identifier, as well as code for data handling is located in `src`.

You should start by copying `src/assignment3.py` to a new name `src/assignment3_LASTNAME.py`. For example, Sharid will copy the file to `assignment3_LOAICIGA.py`. This is the file that you will edit and finally upload to Moodle after you are done with the assignment.

4 Uralic language data

We will be working on datasets which originally stem from the Universal Dependencies[1] treebanks for six Uralic Languages Estonian (est), Finnish (fin), Komi-Zyrian (kpv), Meadow Mari (mrj), Erzya (myv) and Udmurt (udm). Universal Dependencies is a project which prepares syntactically parsed text corpora for an ever increasing selection of languages. The syntactic annotations follow the same standard for all languages. Hence, the "Universal" in the title.

For the purpose of this assignment, we've only retained word forms and language ID from the text corpora. Here are a few lines from the training set `data/uralic.train`:

```
сюлонь          myv
вольпасьын      kpv
туялысьлы       kpv
курскоень       myv
кӧнтусь         kpv
todistuksensa   fin
пойпанго        myv
```

There are two different scripts in the data. Estonian and Finnish are written in Latin script and the other languages are written in Cyrillic script. It is, therefore, relatively easy to distinguish between words which are Estonian or Finnish and words which are Komi-Zyrian, Meadow Mari, Erzya or Udmurt. However, it is not as easy to determine the specific language.

[1] `https://universaldependencies.org/`

5 Starter Code

Let's look at the main program in `assignment3.py`. It ...

1. accepts an argument from user to select the recurrent layer (LSTM, GRU, or vanilla RNN). If no argument is given, LSTM is selected by default,
2. reads in the Uralic datasets using `read_datasets()`,
3. initializes the selected model, the corresponding optimizer, and the loss function,
4. trains the model for 100 epochs on the training data and then uses the learned model to classify the test data.

The function `read_datasets` returns 3 objects:

- `data` – a dictionary containing the training, development and test data.
- `character_map` – which maps characters like 'a' into ID numbers which can be used to access a PyTorch embedding.
- `languages` – which is a list of all the language codes in the datasets.

The dictionary `data` has three keys:

- `data['training']` is itself a dictionary which contains the training data for each language. For example, `data['training']['fin']` is the Finnish training data.
- `data['dev']` is the joint development data for all languages.
- `data['test']` is the joint test data for all languages.

A `sample` from the dataset might looks like this:

```
{
    'WORD': 'rantideta',
    'TOKENIZED WORD': ['r', 'a', 'n', 't', 'i', 'd', 'e', 't', 'a'],
    'LANGUAGE': 'est',
    'TENSOR': tensor([72, 37,  3, 33,  0, 70, 43, 20,  0,  3, 72])
}
```

Here `sample['WORD']` is the original word form, `sample['TOKENIZED WORD']` the tokenized word form, `sample['LANGUAGE']` the gold standard Uralic language and `sample['TENSOR']` is a `torch.tensor` which includes the ID numbers corresponding to the characters in the word, for example, 'a' corresponds to 37. Note that this sequence begins and ends with a word boundary marker, corresponding to the ID 72.

5.1 `LSTMModel`, `GRUModel`, `RNNModel`

Your first task is to implement the three recurrent models. They are going to be called `LSTMModel`, `GRUModel`, and `RNNModel` respectively. As mentioned above, they will all include one embedding layer, one recurrent layer, and one feed-forward layer, followed by a `log_softmax` nonlinearity.

The inputs in `forward` should be a PyTorch tensor of shape `[sequence_len` $\times$ `batch_size]`. Here the `sequence_len` refers to the number of elements in the sequence that the recurrent layer is supposed to process through. In our case, this corresponds to the number of characters in the word, also including the word boundary and possible padding characters.

Note that the `batch_size` is the second dimension! This is the typical way in PyTorch to handle batching when processing sequences in recurrent layers.

The embedding layer should receive `batch_size`-many sequences (in our scenario each sequence is a word). The shape of its input tensor will be `[sequence_len × batch_size]`. It should convert all the characters in the sequences to their corresponding embeddings, and return a tensor of shape `[sequence_len × batch_size × embedding_dim]`.

The recurrent layer (whether it is an LSTM, GRU, or RNN) is going to take the outputs of the embedding layer, and processes all the sequences internally. It will return two outputs, e.g.:

```
outputs, (hn, cn) = self.lstm(inputs)
```

The `outputs` variable holds all the states that the LSTM passed through while processing the input sequence. It is going to be of shape `[sequence_len × batch_size × hidden_dim]`. This means, for each sequence in the batch, it will include one state of size `hidden_dim` for each letter in the sequence. Hence it will have `sequence_len`-many of these states. The second return variable includes both the the final hidden state `hn` of the recurrent layer after it consumes the whole input (and eventually the cell state `cn`, if existing). Therefore, it will be equal to the final element in `outputs`. Its shape will be of shape `[1 × batch_size × hidden_dim]`.

This redundant design is a conscious choice in PyTorch. The second output could possibly be omitted all together, since it can be readily obtained by taking the last element of the first output, but the final state is used commonly enough by itself that the designers preferred outputting it separately to allow cleaner codes.

The final linear layer is going to take the final state of the recurrent layer, for which you can use either the last element of the first output (eg. `outputs[-1]`), or the second output directly. This input should be reshaped into a `[batch_size × hidden_dim]` format, which is how batches are formatted for being used by feed-forward networks. Therefore, the final layer is going to accept an input of shape `[batch_size × hidden_dim]`, and output the class scores in the shape `[batch_size × n_classes]`. You should then feed this output to a `log_softmax` function in the `forward()` function to obtain the class log probabilities.

Hint The official PyTorch documentation uses a function `init_hidden()` to initialize a hidden state, and then feeds this initial hidden state to the recurrent layer (eg. LSTM) as its second argument. This is useful when you want to learn initial states as well, which you can use to steer the network to a certain direction. For example, you may want the network to translate a sentence in one style (e.g., formal) given a certain initial state, and in another style (e.g. i,nformal) given another initial state. The initial state can thus be used as a "cue" to the network. However, our inputs will be in completely random order, and we will not know in advance which language they are from, so we will not use such a scheme. Therefore, you don't need to initialize the hidden state. Instead, here you can use only the embedding layers output as your single input, as shown above. When the second argument is omitted, the network automatically creates for every input zero vectors for the initial states.

6 Training on a single example, batch size = 1

Now, you will implement the model training. Start by training on a single example (i.e., a batch size of 1) at a time to make sure that your basic models are working properly. For this part, use the mini-versions of the datasets (`uralic.mini.train`, `uralic.mini.dev` and `uralic.mini.test`) to make sure the training is not too slow.

You will implement an auxiliary function `get_minibatch()` which will take the word examples in the current minibatch as a list called `minibatchwords`, and return the corresponding input to the network `mb_x`, as well as the expected output `mb_y`. Feel free to use additional / different arguments for `get_minibatch()`. The `minibatchwords` variable will be a list, including `batch_size` elements from the training set.

The output `mb_x` of `get_minibatch()` consists of this data being processed into a tensor of shape `[sequence_len × batch_size]` as expected by the embedding layer. Since in this part your batch size is going to be 1, your `minibatchwords` is going to consist of a single data point. You need to extract the `TENSOR` representation for this data point, and reshape it to a tensor of `[sequence_len × 1]` to be used as input into the embedding layer.

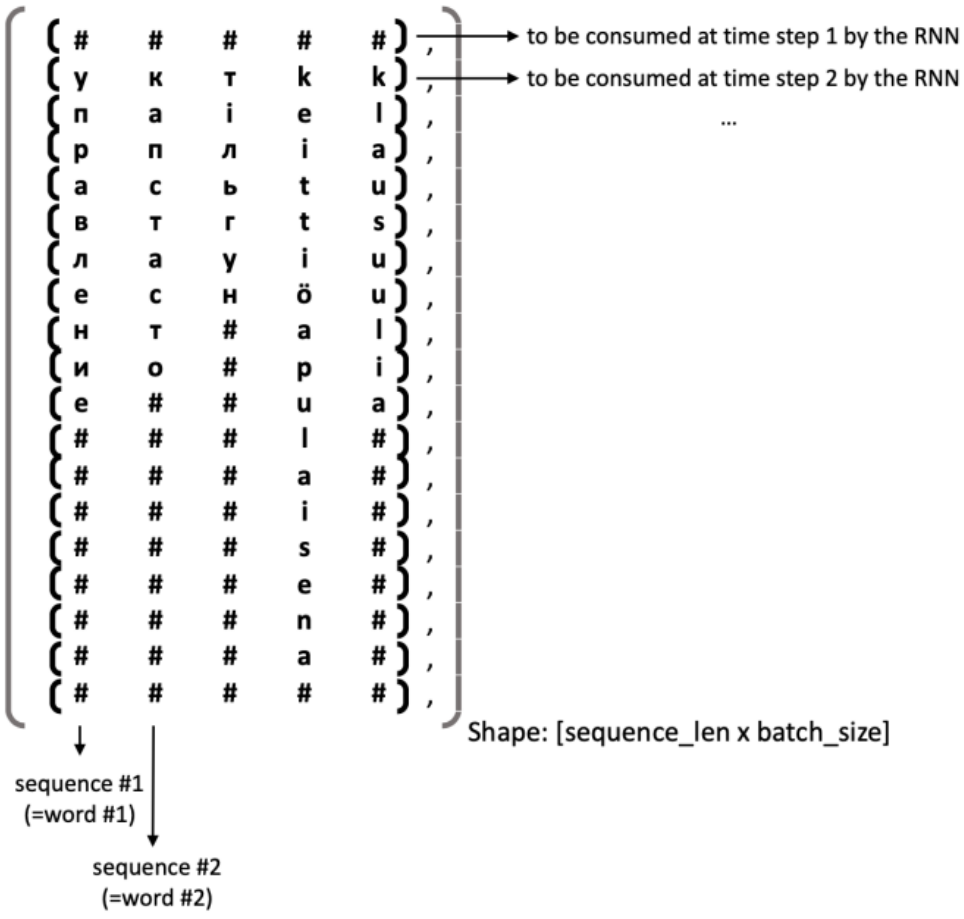
The output `mb_y` of `get_minibatch()` will include the corresponding class ID (out of 6 possibilities). In general, it would be of the shape `[batch_size]`. While your `batch_size` is 1, this tensor will consist of a single gold label for the single example in the batch. You can use the function `label_to_idx()` in the skeleton code to convert language labels ('fin', 'udm', etc.) to corresponding class IDs.

As for the previous classification problems, please again use the `NLLLoss` to optimize your model. Also use a reasonably optimizer, e.g., `Adam`. You can choose whether or not to use a regularizer.

7 Mini-batch training

Batching in recurrent networks is different from batching in feed-forward networks. This is because there's an additional dimension, we need to think about the index over the elements of the sequence.

The logic of batching in sequence-processing models is that we arrange the first element of the batch so that it holds the first elements of every sequence in the batch. The second element of the batch is going to hold the second element of every sequence in the batch, and so on. We can visualize it as:



There is one important practicality in this type of batching. To be able to represent `batch_size`-many inputs in a single tensor, we have to make sure that all of them are of the same length. However, words in our data have varying lengths. Remember that this is not a problem if our recurrent layer is processing 1 sequence at a time (`batch_size = 1`). However, if we want to use a `batch_size` greater than 1 (which we generally do), then we need to make sure all the sequences in the batch have the same length with each other, so that they can be tucked into the same tensor. We can achieve this by padding extra characters at the end of each word, until they are all the same length with the longest word in the batch. Note that different batches can have different sequence-lengths, they don't have to match with each other. The sequence-length of each batch is simply determined by the longest word it contains.

However, notice that there is important inefficiency here: since the length of the words in our batch are widely different, we have to pad each one to the longest word in it. Since our dataset can include very long words, we will likely need to pad almost all words to a great degree, like in the above figure. This is both time-wise inefficient, and also introducing a slight bias into the data. A good trick to get out of this conundrum is to sort our dataset according to the word length after shuffling it at every epoch. The sorting makes sure that similar-length words are next to each other.

(The python `sort()` function is stable in the sense that it preserves the input order in between items that are equal in terms of the sorting key. This will allow the dataset to still retain some of the order from the previous shuffling, so your dataset order will still be randomized between the epochs.)

Also note that it's not exactly ideal to use a "sorted" train set, because this violates the ideal randomness of the training examples. Our system will acquire a small bias because we always start presenting from the shorter examples and move towards the longer ones. Strictly speaking, this presentation order should also have been randomized to achieve a more unbiased system. However, this effect is negligible for this system, so you can ignore this effect. But, keep in mind that the presentation order of examples can introduce a bias if it is not strictly-random, and always consider if this results in your system exploiting this loophole and developing itself accordingly.

Now you can also start using mini-batches in the evaluation code, which will be much more efficient than processing sequences one by one.

8 Submission

Upload your `assignment3_LASTNAME.py` file on Moodle.