DD2418 Language Engineering
Assignment 4

Johan Boye, Dmytro Kalpakchi

2021-05-07

All programs should be written in Python 3, unless specified otherwise in the problem instructions. Don't use any external libraries (that are not part of the Python 3 distribution) unless otherwise specified.

Mandatory part

1. (Gated Recurrent Units) Gated Recurrent Unit (GRU) networks, introduced in (Cho et al., 2014), is a type of recurrent neural networks (RNNs) used for capturing long-term dependencies in natural language. Mathematically, a GRU can be described by the equations found here. PyTorch has a ready-made implementation of GRU residing in `torch.nn.GRU` class, but for this problem we want to implement our own class GRU2 residing in `GRU.py` **using PyTorch standard mathematical operations**. The method `forward` of GRU2 takes a tensor with a batch of sequences as input, and outputs a tuple containing:

   - `outputs` - a tensor containing the output features h_t for each t in each sequence (the same as in PyTorch native `gru` class);
   - `h_fw` - a tensor containing the last hidden state of the forward cell for each sequence, i.e. when t = length of the sequence;
   - `h_bw` - a tensor containing the last hidden state of the backward cell for each sequence, i.e. when t = 0 (because the backward cell processes a sequence backwards). This is optionally returned if bidirectional flag of GRU2 class is on.

   Each element of the sequence should be processed using a corresponding `GRUCellV2` instance, i.e. `self.fw` for a forward cell and `self.bw` for a backward cell. Note that an efficient way is to process the t-th element of every sequence in a batch simultaneously, by exploiting tensor operations in PyTorch.

   When implementing the `forward` method of the `GRUCellV2` class, consider making an optimization of the original equations. Notice that computing reset, update and new gates of GRU requires multiplying the respective weight matrices by the current sequence element x_t and by the previous hidden state h_{t-1}. Instead of performing 3 matrix multiplications you can perform only one by stacking weight matrices on top of each other and multiplying by x_t or h_{t-1} directly. In fact, the weight matrices are already initialized stacked for you in the `__init__` method and you can use `torch.chunk` function to split this matrix back into 3 different matrices after performing the necessary matrix multiplications (for reset, update and new gates **in this order**, which is necessary for the tests below).

   After finishing the implementation, run

   ```
   python GRU.py
   ```

   If both questions in the output get "Yes" as an answer, this means that the hidden states of the forward and backward cells of your GRU implementation are identical to the ones of PyTorch implementation after running a forward pass.

2. (Named Entity Recognition with GRUs) In this problem, we will explore the use of recurrent neural networks (RNNs), specifically bidirectional RNNs (BIRNNs), for doing named entity recognition (NER). You will be using the GRUs that you implemented in the previous problem. After training your BIRNN on the training set, you'll use that model to classify words from a test set as either 'name' or 'not name'.

Just as in assignment 3.3, we will use word vectors as our features, but we will also complement the word vectors with character vectors, in order to use both word-level and character-level information.

Have a look in the training file `ner_training.csv`. Every line consists of a sentence number (empty if it's the same sentence), word and a label. If the label is 'O', then the word is not a name; if it something else, then the word is a name of some kind. Currently we will consider all of these as just 'names'.

Your task is to implement method `forward` of the class `NERClassifier` in `NER.py`.

To solve the NER problem, we're going to use the neural network architecture presented in Figure 1, which is a simplified version of the work by Lample et al. (2016). Let us explain how it works by considering an example sentence "My older brother Jim lives in Sweden".

- Every word is represented by a concatenation of the corresponding pre-trained word vector and a trainable character-level word vector.

- For pre-trained word vectors we suggest using 50-dimensional GloVe vectors available for download here (use a file named *glove.6B.50d.txt*). Every sentence will then be represented by a 3D tensor with the following shape

  (batch size, max sentence length, 50)

  Note that all sentences in the batch are already padded for you to the length of the longest sentence in the batch.

- Character-level word vectors address the problem that many named entities will lack pre-trained word vectors. Each such vector is a concatenation of last hidden states of forward and backward cells of a character-level BIRNN. For instance, consider the word "Jim", the word would be processed letter by letter from left to right by the forward cell of BIRNN by performing the following steps.

  - For each character pick a corresponding character embedding from a randomly initialized embedding matrix. Each word now becomes a 2D tensor, where each row corresponds to the character embedding for every letter of the word.

  - All neural networks in PyTorch operate on tensors, which must have a fixed size. Evidently, words have different lengths, so they should be padded to the length of the longest word in the entire corpus. As the padding symbol, we have chosen a special character `<UNK>`, which is the 0-th character in the `CHARS` list. Given that each word is a 2D tensor (as we saw in the previous step), the whole sentence is now a 3D tensor and a batch of sentences will be a 4D tensor.

  - Now you will need to input the padded 4D tensor to the forward and backward GRU cells of your character-level BIRNN. However, RNNs in PyTorch (and in our custom implementation) operate on max 3D tensors, so you will need to reshape the 4D tensor

	not name	name
My	0.7	0.3
older	0.87	0.15
brother	0.92	0.08
Jim	0.34	0.66
lives	0.65	0.35
in	0.88	0.12
Sweden	0.18	0.82
<pad>	0.98	0.01
<pad>	0.98	0.01
<pad>	0.98	0.01

Fully connected softmax prediction layer

Word-level BiRNN

concatenate

Character-level BiRNN

Word embedding projection layer

Character embedding projection layer

J   i   m

Repeat for every word
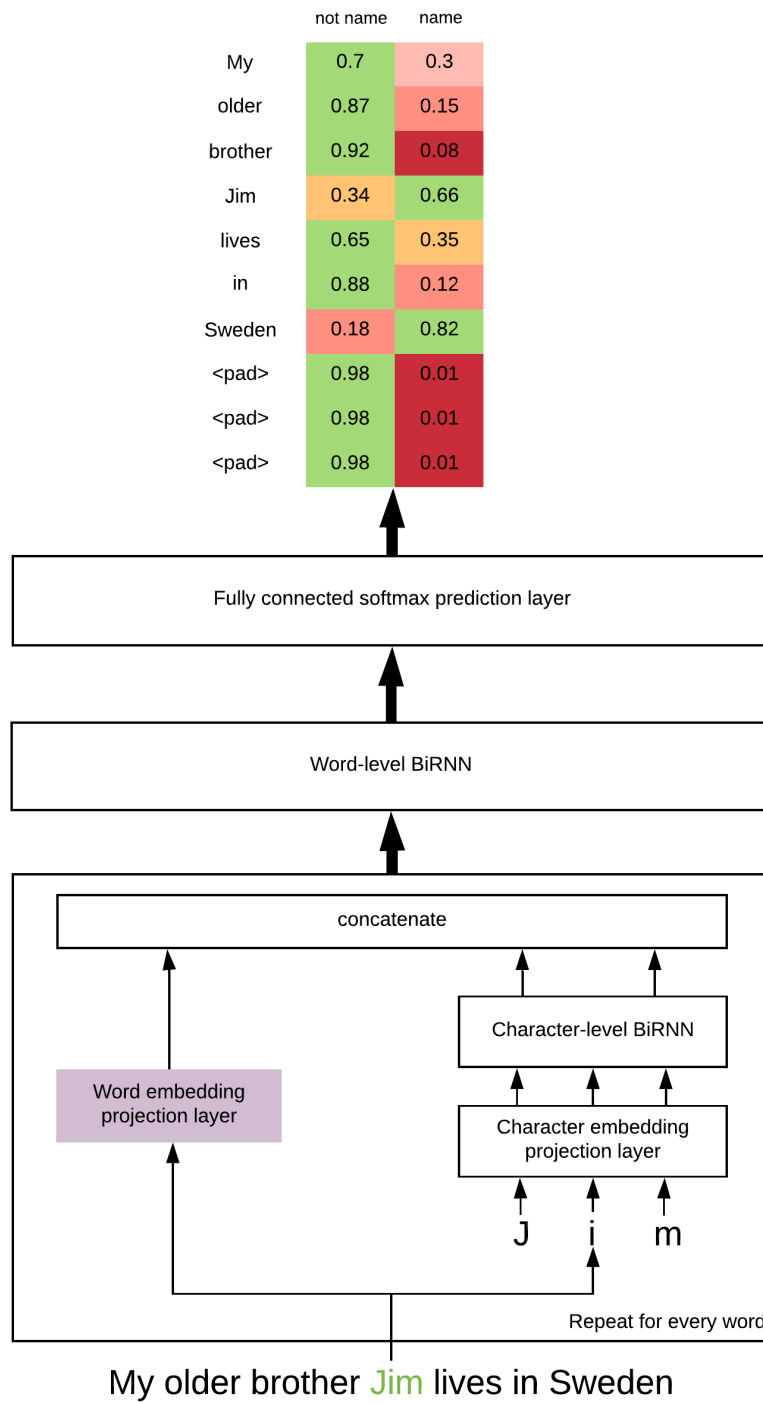
My older brother Jim lives in Sweden

Figure 1: The architecture for a neural network that we're going to use for classifying named entities

into a 3D tensor by collapsing the first two dimensions. The shape of the new tensor should be

(batch size × max sentence length, max word length, char embedding size)

Make sure to save the exact values for batch size and max sentence length - you'll use it in the next step. Now this 3D tensor can be fed to both forward and backward GRU cells. **Use your own implementation of bidirectional** GRU**, i.e. a GRU2 class implemented in the previous problem.**

- Every word was now fed through the forward and backward cell character by character, so we have 2 last hidden states (one for forward and one for backward cell) per word. Each of the hidden state tensors should have the following shape

(batch size × max sentence length, char hidden size)

Note that the last hidden state of the forward cell corresponds to the last character, whereas the last hidden state of the backward cell to the first character. So effectively you have just read your word character-by-character in both directions. However, we want only one character-level vector per word, so we simply, concatenate the last hidden states of the forward and backward cells of your character-level BIRNN. This will result into one vector for each word, i.e., a tensor of the following shape

(batch size × max sentence length, 2 × char hidden size)

- Recall that a sentence on the word level is represented as a 3D tensor of the shape

(batch size, max sentence length, 50)

In order to be able to concatenate character-level and word-level tensor, we need the former as a 3D tensor as well, with the first two dimensions being the same as for the latter. This can be easily achieved by reshaping the character-level tensor from the last step into the one with the shape

(batch size, max sentence length, 2 × char hidden size)

. Now you have your character-level word vectors ready to go!

- Concatenate GloVe vectors with character-level word vectors into a tensor of a shape

(batch size, max sentence length, 50 + 2 × char hidden size)

- Input a batch of sentences (it's already padded on the sentence-level for you), which is a 3D tensor to the word-level BIRNN. **Use your own implementation of bidirectional** GRU**, i.e. a GRU2 class implemented in the mandatory part.**

- This time use the `outputs` tensor of your BIRNN and input it to the `self.final_pred` linear layer and return this tensor. **NOTE: softmax will be applied automatically by torch.nn.CrossEntropyLoss, so we shouldn't do it in our `forward` method.**

To test your implementation run

```
python NER.py
```

You should be able to reach the accuracy of about 97% after 5 epochs (it should take max 1h to train).

4

Optional part

1. (Ablation studies) The architecture of the network above is quite complex and it's not immediately obvious that a simpler architecture would not suffice. A simpler architecture is more beneficial in practice, since both training time and inference time gets reduced. To investigate the matter, researchers do the so called *ablation studies* by switching off a certain component of the architecture and observing the performance difference (if any). Your task in this problem is to perform the following ablations:

    1. Switch off character-level embeddings by putting character embedding size (`-c` argument) to 0. Convert the word-level RNN to unidirectional (`-wud` switch).

    2. The same as previous, but make the word-level RNN bidirectional.

    3. Switch on character-level embeddings, but make the character-level RNN unidirectional (`-cud` switch). Try this with both bidirectional and unidirectional word-level RNNs.

    4. The same as the previous one, but make the character-level RNN bidirectional.

    Save the confusion matrices of all your ablation studies and be ready to reason about the results.

References

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Lample, G., Ballesteros, M., Subramanian, S., Kawakami, K., & Dyer, C. (2016). Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360*.