DD2418 Language Engineering Assignment 4

Johan Boye, Dmytro Kalpakchi

2020-05-06

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.

In any problem, the TA might check your solution using a secret input, which you haven't seen before. Therefore, try to make your solutions as general as possible; don't hard-code your solutions to work with the provided test cases only.

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` will take 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.

   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. Well done! Go grab yourself coffee or tea with some cinnamon buns - you've successfully finished the mandatory part of the last assignment of this course! ☺

Optional part

1. (Named Entity Recognition with GRUs) In this problem, we will explore the use of recurrent neural networks (RNNs), specifically bidirectional RNNs (BiRNNs) with GRUs (that you implemented in the mandatory part), for doing named entity recognition. 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'.

   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'.

   In assignment 2, we trained a binary logistic regression model on the same dataset with a set of manually engineered features. By contrast, in this assignment we'll avoid manual feature engineering, and will use word vectors as our features.

   Your task is to implement method `forward` of the class `NERClassifier` in `NER.py`, as well as the training procedure using mini-batch gradient descent in the main section of `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 one presented in (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've used 50-dimensional GloVe vectors available for download here (use a file named *glove.6B.50d.txt*).
   - 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 letter pick a corresponding character embedding from a randomly initialized embedding matrix. Each word now becomes a 2D matrix, 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 in every sentence have different lengths, so they should be padded to the maximal length of a word. You can pad them with any special character of your choice. We have chosen a special character `<UNK>`, which is a 0-th character in the `CHARS` list. Given that each word is a 2D matrix (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.

  - Input your padded 4D tensor to the forward and backward GRU cells of your character-level BiRNN by first reshaping it into a 3D-tensor via collapsing the first two dimensions (reshaping is necessary because RNNs in PyTorch operate on max 3D tensors). Save the shape of the first two dimensions - you'll use it in the next step. **Use your own implementation of bidirectional GRU, i.e. a `GRU2` class implemented in the mandatory part.**
  - Remember that every word represented on a character level was a 2D matrix (in a now reshaped 3D tensor) and we want just one vector for each word. To achieve that concatenate the last hidden states of the forward and backward cells of your character-level BiRNN. This will result into a 2D matrix that should be reshaped the tensor back to 3D tensor (using the saved shape and unfolding the first two dimensions). Now you have your character-level word vectors!
- Concatenate GloVe vectors with character-level word vectors.
- 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).

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*.

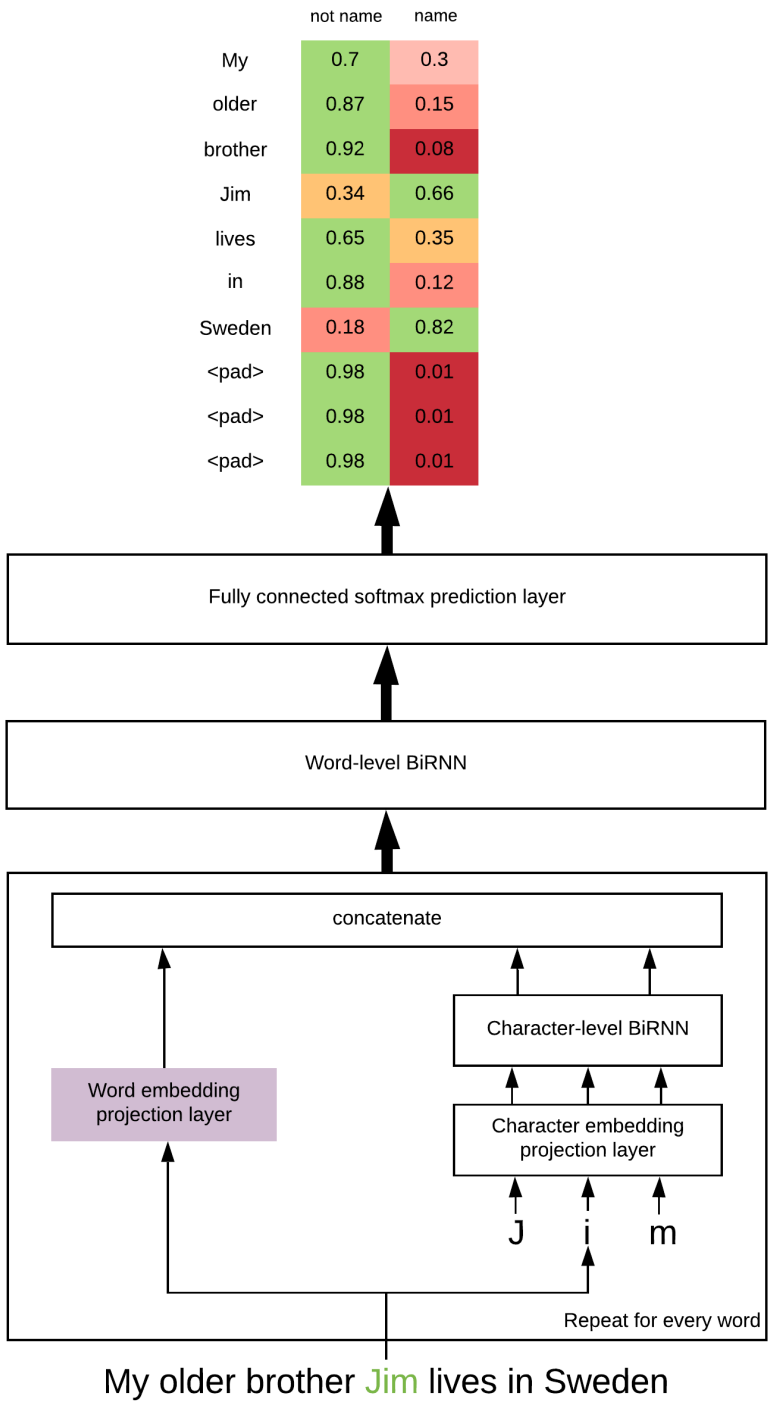

Figure 1: The architecture for a neural network that we're going to use for classifying named entities