DD2417 Language Engineering Assignment 4

Johan Boye, Dmytro Kalpakchi

2023-04-28

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 the `torch.nn.GRU` class, but for this problem we want to implement our own class `GRU2` residing in `NER/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/NER.py`.

   - First, download pre-trained Glove word embeddings by running the program `get_glove.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, use the 50-dimensional GloVe vectors you have downloaded. 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

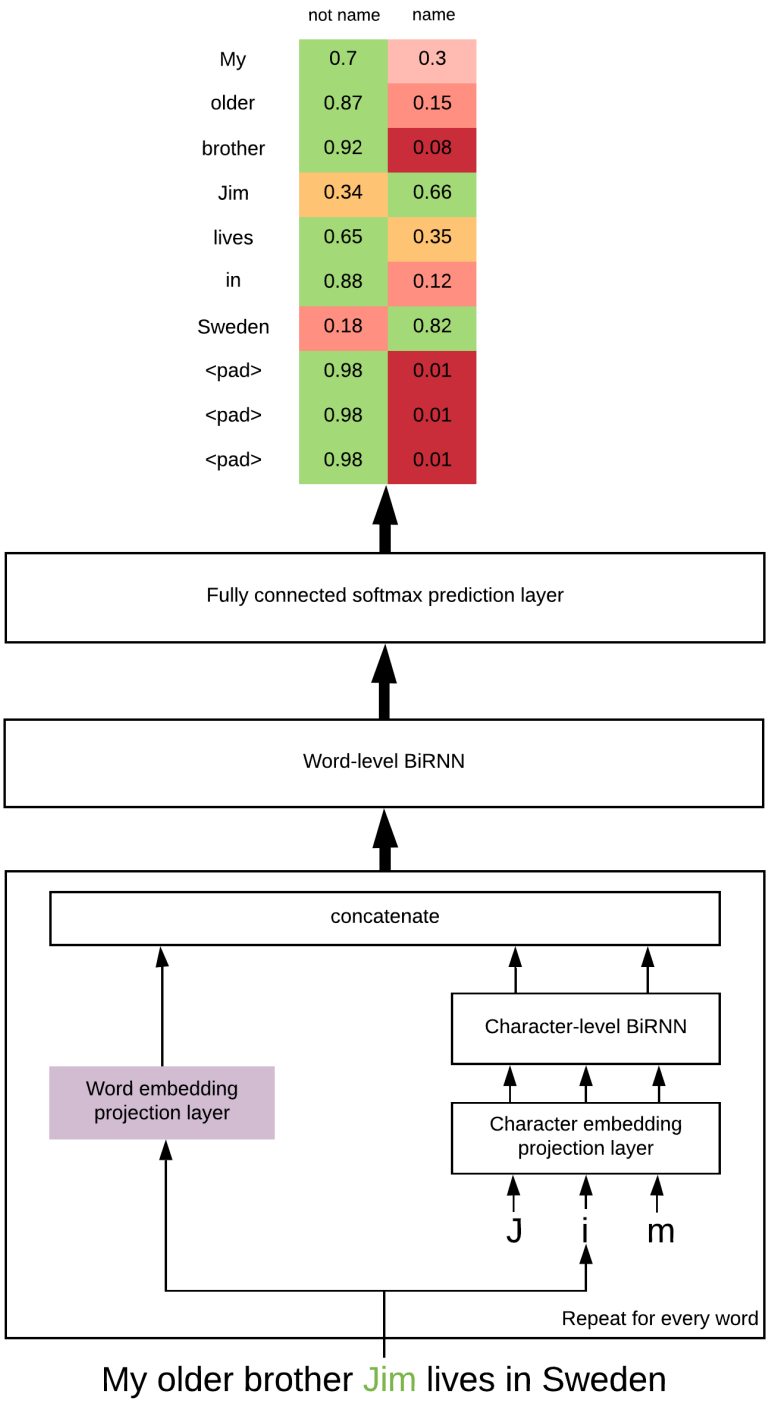

Figure 1: The architecture for a neural network that we're going to use for classifying named entities

implementation) operate on max 3D tensors, so you will need to reshape the 4D tensor 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).

Optional part

3. (Translation using RNNs) In this assignment, we are going to explore the task of translating English sentences into Swedish, using two connected recurrent neural networks with GRU cells. These two networks are called the *encoder* and the *decoder* (see lecture 8c).

   - The *encoder* produces a sequence of hidden states (as many as there are input tokens) from the English input. Each hidden state h_j is a representation of the English sentence up to (and including) the jth word. The encoder is a bidirectional RNN with GRU cells.
   - The *decoder*, which is an unidirectional RNN, computes the most likely Swedish output sentence, one word at a time. When computing the ith output word, the decoder uses a context consisting of the preceding output word, the preceding decoder hidden state s_{i-1}, and (if an attention mechanism is used), all the hidden states from the encoder.

   In principle, the encoder and decoder could have hidden states of different dimensionality, but for simplicity we are going to assume that all hidden states are n-dimensional. Words are associated with a unique ID, and each word ID is represented by e-dimensional word embeddings. There is one set of embeddings for English words, and another set for Swedish words. The English embeddings will be initialized using Glove embeddings, but can be modified during training of the network.

   (a) Your first task is to implement the encoder, which takes a batch of input words x_i in English and first transforms each of them into an embedding by projecting them using the the embedding matrix E. The sequences of embeddings E_{x_i} are then fed into the RNN to produce two outputs (default in PyTorch):

   1. A 3D-tensor containing hidden states for each subsequence in the sequence.
   2. A matrix containing the final hidden state, corresponding to the whole sequence (should be the same as the last element of the aforementioned 3D-tensor).

   (b) Your second task is to write the decoder without an attention mechanism. In this simplified scenario, the decoder does not use the information in all the hidden states from the encoder. Instead, when generating the next output word y_i, the decoder only uses a context consisting of the preceding output word y_{i-1} and the preceding decoder hidden state s_{i-1}. The only information the decoder receives from the encoder is the last hidden state, which now becomes the first hidden state s_0 in the decoder.

   The decoder thus generates a sequence of output words IDs $y_1, \ldots, y_{Ty}$. First the hidden state is updated:

   $$s_i = \mathsf{GRU}(E_{y_{i-1}}, s_{i-1})$$

   where $E_{y_{i-1}}$ is the embedding for word y_{i-1}. The ith word ID y_i is generated as:

   $$y_i = \arg\max Os_i$$

   where O is a matrix of size (*number of unique Swedish words* $\times$ *decoder hidden size*). The resulting vector thus has as many dimensions (=logits) as there are unique Swedish words.

   To start things off, the first decoder hidden state s_0 is set to the last hidden state of the encoder, and the first input symbol y_0 to the decoder is the special boundary symbol '#'.

Don't use any Python loops in your implementation. Do all computations using tensor operations.

After having completed the implementation, run the script `run_translator_no_att.bat` (or `.sh`). This script will also save your model in a directory with a name starting with "model_". You could load the saved model by running `load_model.sh` and providing the directory name for your model as the only argument. You can expect to get about 50% of the translations correct.

(c) Your third (and final) task is to add an attention mechanism to your decoder. We now compute the hidden state of the decoder as follows:

$$s_i = \mathsf{GRU}(E_{y_{i-1}}, c_i)$$

where c_i is the *context* in state i, which depends both on the preceding hidden state s_i and all the encoder hidden states. The context is defined as follows:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

where α_{ij} is a probability distribution expressing how much output word i should pay attention to input word j. It is defined as follows:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ij})}$$

$$e_{ij} = v^\top \tanh(W h_j + U s_{i-1})$$

where $v \in \Re^n$ is a parameter vector, and $W \in \Re^{(n \times 2n)}$ and $U \in \Re^{(n \times n)}$ are parameter matrices, all trainable.

Don't use any Python loops in your implementation. Do all computations using tensor operations.

After having completed the implementation, run the script `run_translator_att.bat` (or `.sh`). This script will also save your model in a directory with a name starting with "model_". You could load the saved model by running `load_model.sh` and providing the directory name for your model as the only argument. You can expect to get slightly better results than in problem (a).

After evaluation, the program allows you to enter an English sentence and see the Swedish translation, along with the attention matrix. See the next page for some successful translations we tried (and, no, not all sentences work as well!).

```
> it is seven o'clock.
klockan är sju . <END>
+---------------+---------+------+------+------+-------+
| Source/Result | klockan | är   | sju  | .    | <END> |
+---------------+---------+------+------+------+-------+
| it            | 0.34    | 0.00 | 0.00 | 0.00 | 0.03  |
| is            | 0.03    | 0.96 | 0.00 | 0.00 | 0.00  |
| seven         | 0.30    | 0.03 | 0.98 | 0.01 | 0.08  |
| o'clock       | 0.08    | 0.00 | 0.01 | 0.91 | 0.16  |
| .             | 0.24    | 0.00 | 0.00 | 0.08 | 0.73  |
+---------------+---------+------+------+------+-------+

> i should go to bed now.
jag borde gå och lägga mig nu . <END>
+---------------+------+-------+------+------+-------+------+------+------+-------+
| Source/Result | jag  | borde | gå   | och  | lägga | mig  | nu   | .    | <END> |
+---------------+------+-------+------+------+-------+------+------+------+-------+
| i             | 0.86 | 0.00  | 0.00 | 0.00 | 0.00  | 0.00 | 0.00 | 0.00 | 0.03  |
| should        | 0.02 | 0.95  | 0.05 | 0.00 | 0.00  | 0.00 | 0.00 | 0.00 | 0.01  |
| go            | 0.00 | 0.04  | 0.68 | 0.04 | 0.00  | 0.00 | 0.00 | 0.00 | 0.01  |
| to            | 0.00 | 0.00  | 0.19 | 0.22 | 0.06  | 0.01 | 0.00 | 0.00 | 0.01  |
| bed           | 0.01 | 0.00  | 0.07 | 0.73 | 0.71  | 0.66 | 0.04 | 0.03 | 0.05  |
| now           | 0.06 | 0.00  | 0.00 | 0.01 | 0.23  | 0.32 | 0.78 | 0.27 | 0.30  |
| .             | 0.05 | 0.00  | 0.00 | 0.00 | 0.00  | 0.01 | 0.17 | 0.69 | 0.60  |
+---------------+------+-------+------+------+-------+------+------+------+-------+

> i don't like reading books.
jag gillar inte att läsa böcker . <END>
+---------------+------+--------+------+------+------+--------+------+-------+
| Source/Result | jag  | gillar | inte | att  | läsa | böcker | .    | <END> |
+---------------+------+--------+------+------+------+--------+------+-------+
| i             | 1.00 | 0.00   | 0.00 | 0.00 | 0.00 | 0.00   | 0.01 | 0.04  |
| do            | 0.00 | 0.57   | 0.00 | 0.00 | 0.00 | 0.00   | 0.00 | 0.00  |
| n't           | 0.00 | 0.02   | 0.55 | 0.02 | 0.00 | 0.00   | 0.00 | 0.00  |
| like          | 0.00 | 0.41   | 0.44 | 0.20 | 0.00 | 0.00   | 0.00 | 0.01  |
| reading       | 0.00 | 0.00   | 0.01 | 0.74 | 0.27 | 0.06   | 0.01 | 0.06  |
| books         | 0.00 | 0.00   | 0.00 | 0.05 | 0.71 | 0.94   | 0.13 | 0.15  |
| .             | 0.00 | 0.00   | 0.00 | 0.00 | 0.01 | 0.00   | 0.85 | 0.75  |
+---------------+------+--------+------+------+------+--------+------+-------+
```

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*.