DD2417 Language Engineering
Assignment 4

Johan Boye
2024-04-29

All programs should be written in Python 3. Unlike the previous assignments, you will now work in a JupyterHub environment, with access to more computational resources.

If you have never worked with a Jupyter Notebook before, there are plenty of tutorials to be found on the Internet. But it's easy! The code is divided into cells. You execute a cell by clicking on it and pressing SHIFT ENTER. The output will be shown underneath the cell. In this assignment, some cells should just be executed as they are, but in some cells, you will be asked to add some code before executing them.

When you click on the link above, you might be given a choice of different environments to use. If so, please click on "Python3" under "Notebook". After doing this, you should see the Jupyter Hub environment, with a file tree on the left-hand side of the window, and a notebook (containing code and text) on the right-hand side.

Exercises

0. (Pytorch training) In assignment 4, we will make heavy use of the `torch` library for neural network training. Regardless if you have never seen `torch` before, or if you already are an expert, it is useful to do our two training exercises, which you find in the folder 'exercises'. The first exercise only involves tensor manipulation, whereas the second will allow you to train a simple character-level language model.

   These exercises are **not** mandatory, and I will go through the answers in the problem-solving class on Tuesday 7 May. But I would recommend doing them yourself before!

Mandatory part

1. (Named Entity Recognition with GRUs) The code skeleton is found in the folder 'NER'. In this task, we will make a final crack at the named entity recognition (NER) problem, using bi-directional recurrent neural networks (RNNs), specifically so-called Gated Recurrent Units (GRUs), After training your network on the training set, you'll use that model to classify words from a test set as either 'name' or 'not name'.

   As features, we will use both word vectors and character vectors. This is done in order to represent both syntactic and semantic information, as well as morphological and typographic information, about the word. As word vectors, we will re-use existing Glove vectors. The character vectors, however, will be trained from scratch.

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

   To solve the NER problem, we are going to use the neural network architecture presented in Figure 1. Let us explain how it works by considering an example sentence "My older brother Jim lives in Sweden".

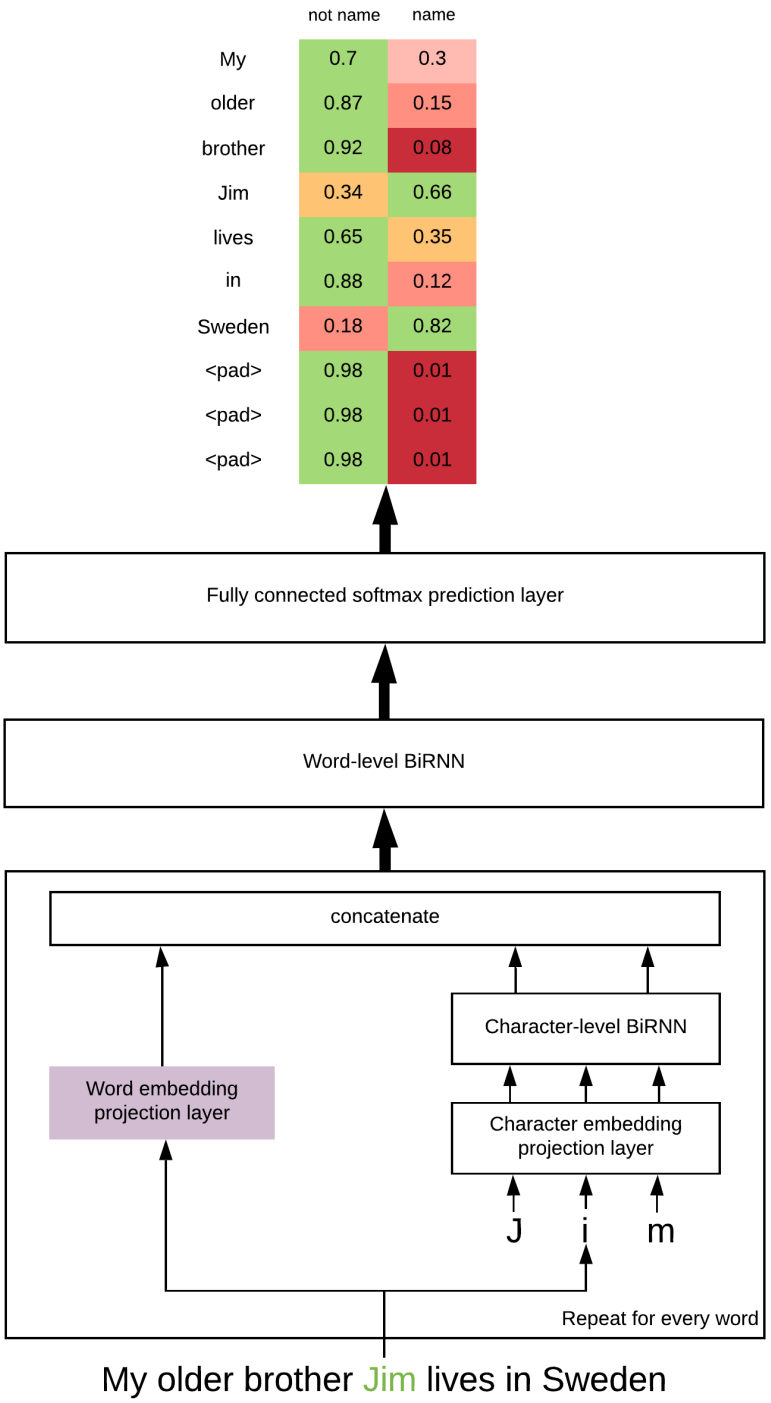

Figure 1: The architecture for a neural network that we're going to use for classifying named entities

- Every word is represented by the concatenation of its pre-trained word vector and a trainable character-level word vector.
- For word vectors, we use pre-trained 50-dimensional GloVe vectors. Every sentence will then be represented by a 3D tensor with the following shape

  $$(\text{batch size}, \text{max sentence length}, 50)$$

  Note that all sentences in the batch are already padded for you to the length of the longest sentence in the batch.
- Character-level vectors address the problem that many named entities lack pre-trained word vectors. Each character-level vector is a concatenation of the last hidden states of the forward and backward cells of a character-level BIRNN. For instance, consider the word "Jim". It 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 the corresponding character embedding from a randomly initialized embedding matrix. Each word now becomes a 2D tensor, where each row corresponds to the character embedding of every letter of the word.
  - All neural networks in PyTorch operate on tensors, which must have a fixed size. As words have different lengths, they must be padded to the length of the longest word in the entire corpus. As the padding symbol, we have chosen the special character `<UNK>`. As 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 need to input the padded 4D tensor to the forward and backward GRU cells of your character-level BIRNN. However, RNNs in PyTorch operate on 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

    $$(\text{batch size} \times \text{max sentence length}, \text{max word length}, \text{char embedding size})$$

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

    $$(\text{batch size} \times \text{max sentence length}, \text{char hidden size})$$

    Note that the last hidden state of the forward cell corresponds to the final character, whereas the final hidden state of the backward cell to the first character. 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 in one vector for each word, i.e., a tensor of the following shape

    $$(\text{batch size} \times \text{max sentence length}, 2 \times \text{char hidden size})$$

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

    (batch size, max sentence length, 50)

    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 has already been padded on the sentence-level for you), which is a 3D tensor to the word-level BIRNN.
- 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.

2. (Translation using RNNs) The code skeleton is found in the 'translate' folder. In this assignment, we will 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 below 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

Optional part

3. In this task, you will implement a character model based on the Transformer architecture, starting from the provided notebook in the 'charlm' folder.

   The model you will implement here will have a context of 32 characters, i.e., it will consider the preceding 32 characters when estimating the probabilities of the possible character coming next. Due to the clever transformer architecture, the model will have less than 50,000 trainable parameters. As a comparison, the simpler model in exercise 2 only had a context of 8 characters but had more than 300,000 trainable parameters.

   The architecture we will explore consists of one or several *transformer encoders*. The core step in each encoder is the *multi-head self-attention calculation*, which transforms vectors representing the tokens of a sentence or a text into "contextualized vectors", representing contextualized versions of the same tokens. After the encoders have been applied, a final linear layer with a softmax function are applied, giving probabilities for the next token.

   Your task is to implement a part of the self-attention calculation. Look for "YOUR CODE HERE" and "REPLACE WITH YOUR CODE" in the `MultiHeadSelfAttention` class.

   In large language models, the tokens are words (or parts of words), whereas we will only deal with characters to get a more manageable task. When implemented, the program will consider the latest 32 characters and suggest the next character to follow.

   The program will train its model on the first Harry Potter book. After each training epoch, the program will generate 200 characters. If everything is correctly implemented, each epoch should take about 15–20 seconds, and the output should look more and more like real text for each epoch.

   You might expect a behavior similar to this:

   After 1 epoch:

   ```
   and aeaned y aoe donn to uannd t aoeed taoon nae tre aee tro yoo tne hae t aounn
   taoone taoun t the erera enera yne asd s aoune aande g aaonn gno tte the e arane
   yre sae yre sae doen donn ea tane ter roee yannd aaonn gt toee aantne aapng taoen
   roouu tait tne the s aoone tane erery taane
   ```

   After 10 epochs:

   ```
   sting and and all and alled and and alled and they and and alled and thery said
   Harry was all and and alled and thery fore said Harry was and alled and the poing
   to and and peard and all all and and and one poper and the poing to the poing to
   the poing to the poing to the poing to the poing to the
   ```

   After 30 epochs:

   ```
   and said of all at and the Gryffindor and the points and and the see said Ron the
   poing and a stall off a for the the first off of at of the poing to at the points
   and the poing of at of the the said Harry, squeere and as all as fame and the poing
   to at the poing of the said at Mrs.  Norrie starily.
   ```

   and so on. It won't go on improving forever - remember, this is a character model, not a word model, and it is a small network.