Report of homework 3: natural language processing

Clara Eminente

February 15, 2021

1. Introduction

In this assignment an improvement of the *RNN* architecture, the *LSTM*, is used for natural language processing at word level: this architecture, along with a word embedding layer, is trained over six books by Jane Austen in order to be able, given a word, to predict the following one in a sentence.

The books used for this assignment have been downloaded from *Project Gutenberg* and are *Emma, Pride and Prejudice, Mansfield park, Sense and Sensibility, Northanger Abbey and Persuasion*.

The main analysis is contained in the Jupyter Notebook *Eminente_HW5.ipynb*, whereas the script *trained_model.py* uses a pre-trained network to predict *n* words given a seed (i.e. a sentence to expand). The analysis was performed using *Google Colaboratory* to be able to exploit GPU acceleration.

2. Implementation

The basic code expands the one provided during the the fourth laboratory session which consisted in a *LSTM* network with a linear output layer implemented using *Pytorch*.
Since the goal is to predict the next token in a sentence at word level the main addition to the basic architecture is a pre-trained embedding layer, that interfaces the input (sentences) with the network. In the following section the process that leads to the preparation of the dataset and the embedding process are described. In the final paragraph the final architecture of the network is presented and explained.

2.1. Preprocessing

As mentioned in the introduction the dataset is composed of six books by Jane Austen. In order to feed them to the architecture, however, some pre-processing is necessary.

The first step is to create a single text, loading each *".txt"* file and appending it to the previous ones. Then, chapter headings are removed along with chapter numbers and all the text is transformed in lowercase. At this point a list of all unique characters in the text is created and all "odd" characters are substituted by spaces (e.g. &, _, % , numbers, ecc). Moreover, the only punctuation kept is commas (,), full stops (.) and question marks (?). All other punctuation (e.g. !, ;) is substituted by full stops as this won't compromise the overall meaning of the sentence.
Since during the word embedding process commas, full stops and question marks will be considered (and thus "learned") just as normal words, their symbols are substituted with actual words: *"commapunct"* for comma, *"fullstoppunct"* for full stop and *"questpunct"* for question mark.
At this point of pre-processing the only characters in the text should be spaces, newlines and alphabetic characters. In particular the only newlines should be the ones that mark the end of a paragraph in the text i.e. those preceded by a full stop (this characteristics will be important in the following steps).
The full pre-processing is carried out with a custom function, *preprocess*: in its original form the function takes the titles of the books as input and performs all the aforementioned tasks. Setting the boolean variable *is_seed* to *True* the function takes in input a string of text (this will be useful for prediction task).

At this point the dataset consists of a unique string. In order to perform the next steps (word embedding and training) it is necessary to split it into same length *"sentences"*, i.e. parts of text with equal number of words. To do so each paragraph is separated and kept only if its length is greater or equal than a certain threshold (in the case of this project, *25*). A paragraph is defined as that part of text that starts after the couple *fullstoppunct* and ends in the same way. Each remaining paragraph is split in sentences of length *25*. If a sentences ends up being shorter than *25* words it is not added to the dataset. An example can be found in the

appendix.

The choice to not end a sentence when just a full stop is found has two main reasons: first, the sentences must be of same length (which would require some type of padding if full stops were used to end sentences). Second, it was considered interesting to include the full stop as a character and "learn" it.

After this process the dataset consists of *30635* 25-word-long sentences.

2.2. Word embedding

As previously mentioned, a fundamental ingredient in order to be able to learn the next token in a sentence is *word embedding*: it consists in associating to each unique word a vector of floating points in such a way that similar words end up "close" in the embedding space (see Fig. 2 in the appendix). The bigger the embedding space the more complicated the connections between words that it is possible to learn.
For the purpose of this project *gensim* library is used to learn word embedding, and in particular its function *word2vect*. The function takes in input the previously created list of same-length sentences and learns a matrix of dimensions (*n_unique_words*, *embedding_dimension*), that allows, given a word in its "one hot encoded" form (i.e. represented as a vector of dimension *n_unique_words* with all zeroes except for a one at the index corresponding to the word), to obtain its representation in the embedding space (i.e. a embedding_dimension long vector of floating points). From a practical point of view it won't be necessary to build the one hot encoded vector for each word and it will be enough to pass the index corresponding to the unique word to the embedding layer.
Since the embedding procedure takes roughly three minutes and can be carried out just once, the weights of the matrix are pre-trained and saved. In the next section the final structure of the network will be presented and the embedding layer will be added before the *LSTM* cells as pre-trained (i.e. its weights are not updated with backpropagation).

Since the learning will be carried out using the encoded version of each word two dictionaries are created: *w2i* allows, given a word, to obtain the corresponding index (i.e. the only non zero entry in its one hot encoded version) and *i2w* does the opposite. The class *MyDataset* is a custom Pytorch dataset class that contains all the sentences previously generated paired with their encoded version.

2.3. Final architecture

The code below shows the initialization and forward function of the class that instantiates the complete architecture:

```
class Network(nn.Module):
def __init__(self, len_vocab, emb_dim,
    hidden_units, layers, dropout):
super().__init__()


embedding_matrix = torch.load(path+"
    embedding_model/embedding.torch")
embedding_matrix = torch.FloatTensor(
    embedding_matrix)
self.embedding = nn.Embedding(len_vocab,
    emb_dim).from_pretrained(
    embedding_matrix)
#is pretrained
self.embedding.weight.requires_grad = False
#LSTM cells
self.rnn = nn.LSTM(
    input_size=emb_dim,
    hidden_size=hidden_units,
    num_layers=layers,
    dropout=dropout,
    batch_first=True)
#Output layer
self.out=nn.Linear(hidden_units, len_vocab)

def forward(self, x, state=None):
#Embedding
x = self.embedding(x)
# LSTM
x, rnn_state = self.rnn(x, state)
# Linear layer
x = F.leaky_relu(self.out(x))
return x, rnn_state
```

Listing 1: *class Network instantiation*

It is possible to notice that the embedding layer has dimension len_vocab (ie. number of unique words), emb_dim (dimension of embedding space) and thus maps each word in its embedded vector. It then passes the vector to the *LSTM* cells (whose input has dimension equal to that of the embedding space). The final stage is a linear layer whose dimension corresponds, again, to the number of unique words. During training and prediction the final output of the whole architecture can be interpreted has the probability of each word in the dictionary.
The parameters to be set are the number of *LSTM* cells (*layers*) and the dimension of the hidden layers (*hidden_units*). These parameters will be part of a simple grid search, as explained in the following section.

3. Results

At the beginning of the training procedure the previously created dataset is split in 80% train set and 20% test set. Both sets are then transformed into *Pytorch Dataloaders* to be fed in batches to the network. In articular, the network is trained to predict the next token (word) given a word in a sentence. In order to do so, given an entry of the dataset (i.e. a sentence) its encoded version is selected and all words except the last are used as input and all words except the first as targets. In this way the network receives a sentence one word at a time, tries to predict what will follow and compares the result with the actual word. The loss function of the model is the *cross entropy loss*.
A custom training function that also includes early stopping is implemented inside the class *Network*.

3.1. Grid search and final training

Training this type of architecture proved to be very computationally demanding so a very simple grid search approach was used to compare some architectures and choose the best one. All combinations of the following parameters were tested:

- *hidden_units*: [32, 64, 128, 256]
- *layers* : [2,3]
- *optimizer* : [Adamax, Adam]

The training was carried out using the following fixed parameters:

n_folds	3	dropout	0.3
patience	10	len_vocab	13316
num_epochs	120	emb_dim	100

It was observed that the number of layers does not seem to have much influence on the performance of the network. On the other hand the performances increase with the number of hidden units.
The best set was found to be the combination with *256* hidden units, *2* layers and *Adam* optimizer.
Fig. 1 shows the loss trend during the final training of the network. For this purpose the number of epochs was set to 200 and no k-fold cross validation was performed (although a validation set was extracted from the training set to monitor early stopping).

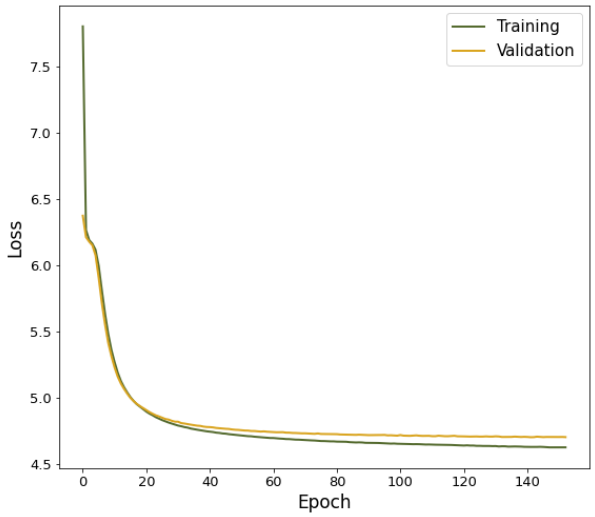

Figure 1: *Cross entropy loss during final training of the network*

3.2. Predictions

As mentioned before the architecture outputs a vector of *13316* entries, which are interpreted by the cross entropy loss as the probability of each word to be the next one. In order to output a single word the softmax of the output is computed (so that it can be actually interpreted as a normalized probability distribution) and then a word is sampled from this distribution. This approach was considered more suitable for this task then selecting the word with maximum output.
A *predict* function is implemented inside the class *Network*. The function takes a string in input that is used as seed, pre-processes it (as done for the text at the beginning), encodes it and feeds it to the network to generate the next word as explained in the previous paragraph, i.e. sampling from a softmax distribution. The function takes as argument the number of words to generate so, after each next word is computed, it is passed as input to the network along with the previous state (the *context*) to generate the following one, and so on and so forth. Each time a word is output it is decoded and printed on screen. Since also punctuation is decoded as a word (see Section 2.2) a new dictionary was implemented that maps each word in itself and each "punctuation word" in its corresponding symbol.
Here some examples of sentences obtained in this way (in bold the seed and in italic the predicted words):

- **Emma Woodhouse, handsome, clever, and rich, with a comfortable home** *, he knew well , and william were suited gaieties at slyness , as he might really understood his finances of interested with your own regard in a little respect*

- **It is a truth universally acknowledged, that a single man** *of had till hearing that not been so hurt . he could think it . and emma roused she would be assured her father health . as well as she*
- **One sunny morning she** *had been purchases to least . and was , though she now heat might forgot himself , these too , a information of the fascination aristocratic , back which body*

It is possible to notice that the sentences do not make any sense. It is however interesting to see that sometimes the order in which word follow each other is grammatically correct.
It would be interesting to investigate the role of the embedding dimension. Also, working with a larger amount of data (i.e. sentences) would probably help in obtaining better results.

Appendix

Example of paragraph split into sentences:

- Original paragraph: *sixteen years had miss taylor been in mr woodhouse s family commapunct less as a governess than a friend commapunct very fond of both daughters commapunct but particularly of Emma fullstopppunct between them it was more the intimacy of sisters fullstoppunct even before miss taylor had ceased to hold the nominal office of governess commapunct the mildness of her temper had hardly allowed her to impose any restraint fullstopppunct and the shadow of authority being now long passed away commapunct they had been living together as friend and friend very mutually attached commapunct and Emma doing just what she liked fullstopppunct highly esteeming miss taylor s judgment commapunct but directed chiefly by her own fullstoppunct*
- First sentence: *sixteen years had miss taylor been in mr woodhouse s family commapunct less as a governess than a friend commapunct very fond of both daughters*
- Second sentence: *commapunct but particularly of Emma fullstopppunct between them it was more the intimacy of sisters fullstoppunct even before miss taylor had ceased to hold the*
- ecc..

Example of PCA performed on embedded words:

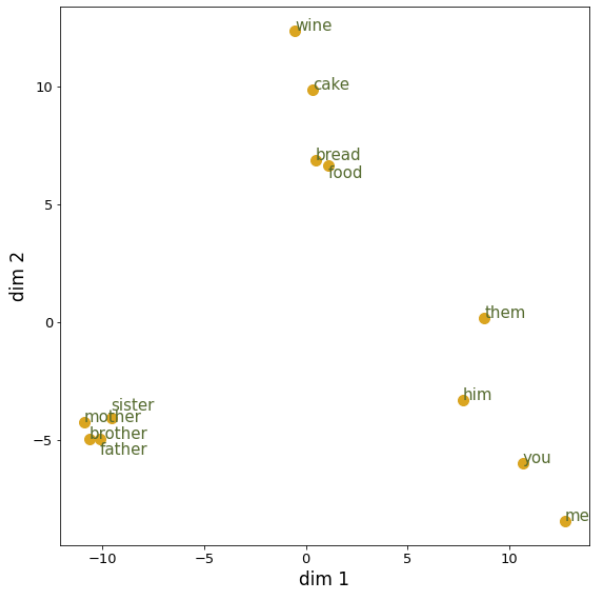

Figure 2: *PCA over the embedded version of some words. The embedding space is 100, nevertheless it is possible to notice how, even if PCA retains only two dimensions, words belonging to similar semantic contexts are very close*