Recurrent Neural Network in PyTorch for Natural Language Processing on C.Dickens Books

Matteo Guida[†]

[†] *Physics of Data M.D. - Dipartimento di Fisica e Astronomia G. Galilei, Università degli Studi di Padova, Via Marzolo 8, I-35131 Padova, Italy.*

January 10, 2021

1 Introduction

In the present work a recurrent neural network (RNNs) is implemented in PyTorch and it is trained on some books written by Charles Dickens: The Chimes, David Copperfield and Pickwick Papers. The plain text of the works are downloaded by project Gutenberg in UTF-8 standard, to have access to project Gutenberg a VPN is required from Italy. The files are loaded in Python as standard strings. The goal of the homework is to generate a fixed number of words given an editable initial seed.

2 Program description

The assignment is solved through three python scripts network.py, dataset.py, preprocessing.py and a jupyter notebook EX3_Matteo_Guida.ipynb.

2.1 Preprocessing

In the script preprocessing.py, after loading the 3 books with the function loading, using the function preprocessing first of all we clean the text from some strange or uncommon symbols leaving only the letters of the English alphabet, numbers, full stops, commas and question marks. Semicolons and exclamation points are transformed in full stops. As the latter appear rarely in a normal text, the current complexity of the model and the amount of training data does not allow to learn how to use correctly that subgroup of punctuation. Then we convert the text to lowercase and finally the remaining punctuation marks are transformed into words according to the following list:

- "." $\longrightarrow$ pointpunctuation
- "," $\longrightarrow$ commapunctuation
- "?" $\longrightarrow$ questionpunctuation

The reason for this choice will be explained in 2.2.

2.2 Word Embedding

Because the neural language model is trained at **word level** we need to convert all the words, that are present in our dataset, into vectors. First of all each word is associated biunivocally to an index so that the two dictionaries word2index and index2word are created. The obtained vocabularies have dimension : 19820. The second operation to be perform is to map each word in a dense vector of floating point values in which similar words have a similar encoding. The values for the embedding are trainable parameters. The higher the dimension of the embedding the more sophisticated relationships between words can be described, but you need mode data to learn them. Those operations are performed with the open-source library **Gensim**. Also the words related to the punctuation at this point have their representation. Another dictionary word2wordwithpunct is created which maps all the words into their-self apart from the punctuation which is in turn mapped into its own symbols. This is for sure a crude solution, however, it does not impact significantly on the efficiency of the program and no other solutions are found in a short time to manage the punctuation with Gensim. Because the original string dataset needs to be converted into a sequence of numbers, i.e. the indices which correspond to the words, if we had created a vector for each sentence then we would have obtained vectors of different lengths. To work around this problem, the simplest solution is selected: given the paragraphs of the full text, windows of length equal to a cut_value are considered. If some paragraph with length smaller than cut_value are found, than they are discarded. Also this choice is quite rough, another possibility that could be explored would be to implement a sentence-wise model where the full text is divided in sentences and each sentence is pad to a fixed vector length (e.g. the longest sentence length). The final dataset is composed by 41459 vectors, where each value corresponds to an index related to a word and at the end is divided into training-test sets (90 % - 10%) and the training set is split again (90% - 10%) to form the validation set. Torch data loaders are used during the training, fixing the batch size to 512.

2.3 Model and Training

The implemented model is composed by an embedding layer, which is pre-trained and loaded during the training procedure and it allows the conversion of the words into the dense vector representation. After that, there is the LSTM module composed by a fixable number of layers and hidden units followed by two feed-forward layers. The first feed-forward layer has Leaky ReLU activation function while the last layer has the size of the vocabulary, so the problem is a classification one with many categorical variables, which needs the cross entropy loss function. The model is defined inside the Network class in the the network.py script where the usual functions to train the network can be found. To try to reduce the overfitting an early stopping condition is implemented. Concretely, given the total amount of epochs, if in the last 20 % of the total epochs the validation loss is not reduced of a fixable percentage, which is fixed to 1%, w.r.t. the previous 20 % of the total epochs, the training is stopped. As a systematic tuning of the hyperparameters is not required, due the computational complexity of RNNs, a simple grid search for the hyperparameters reported in table 1 is performed.

# Hidden Units LSTM	[128,256]
# Layers LSTM	[2,3]
Optimizer	[Adamax, AdamW]
Dropout Probability	[0.2,0.3]

Table 1: Hyperparameters Grid Search.

The other fixed hyperparameters are reported in table 2.

Embedding size	100
# Epochs	200
# Linear Feed-Forward Layers	2
Activation Function 1st Hidden Layer	LeakyReLU
Learning Rate	10^{-3}
L2 Penalty	$5\ 10^{-4}$
Batch Size	512
Cut Size	20

Table 2: Fixed Hyperparameters.

The hyperparameters for the best found hypothesis class are reported in table 3.

# Hidden Units	256
# Layers LSTM	2
Optimizer	Adamax
Dropout Probability	0.2

Table 3: Hyperparameters Best Found Hypothesis Class.

The best found hypothesis class is trained on the union of the training and validation set and than the generalization error is estimated on the test set. The final training is early stopped at the epoch 145 and the learning curve is shown in the figure 1.

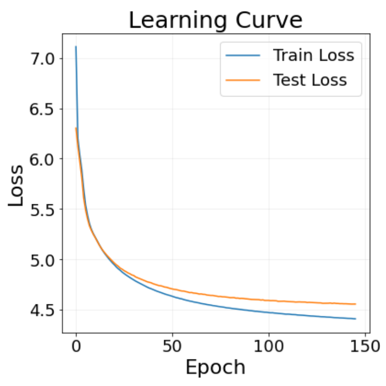

Figure 1: Learning curve for the selected hypothesis.

2.4 Text Generation

The assignment requires to generate a fixed number of words given an initial seed. This task is solved at the end with the function generate_words and softmax_extraction which can be found in the script network.py. Given the initial seed, each following word is obtained by sampling from the last layer of the network according with a softmax distribution, as suggested in the explanation of the assignment. In the following some obtained results are shown.

Initial seed : "*Procrastination is the thief of time. Collar him.*" (David Copperfield).

Generated words : *by three, that i my aunt, with me. micawber, dry, more there s a fire after hounds of ladies errands scrutinised, and that ere, said committed that an affectionate countenance wearing, having likened desirous to render the stage girlish of that i*

Initial seed : "*So may the New Year be a happy one to you, happy to many more whose happiness depends on you.*" (The Chimes).

Generated words : *but, which his spectacles. i sat at the compassion of the entrance. indeed be known back, on the secret, and trust and of i know he had reached her chair. perhaps my aunt had contumely little replenished. there was here so quite bracelet*

For sure the obtained results are not so much satisfactory. Any human being barely literate would generate a result infinitely better. The network did not grasp the semantic of the text, we can see that the words often make sense with the nearest neighbours, but in general the sentences are not coherent. The results on punctuation and syntax are a little better but they remain very basic anyway. One possible improvement could be to enlarge the dataset including many other books and increasing the embedding dimension.