Language Modeling RNN network with LSTM cells

Giovanni Lorenzini (223715)
University of Trento
Via Sommarive, 9, 38123 Povo, Trento TN
giovanni.lorenzini@studenti.unitn.it

Abstract

This work focuses on implementing a Recurrent Neural Network (RNN) Language Model with Long Short-Term Memory (LSTM) units. This report details the realization of a LSTM cell and a RNN network. It shows also the performance obtained, so the perplexity (PPL) on the Penn Treebank word level dataset.

1. Introduction

A recurrent neural network is a class of artificial neural networks that can process variable length sequences of inputs. They are especially powerful when dealing with speech related tasks. There are a lot of variants of RNNs [6]. This particular work employs the so called LSTM version, which is basically an enhancement over the classic Elman network. Generally speaking, it is known that RNNs suffer from vanishing gradients as data from past hidden states eventually gets multiplied in cascade and their contribution gets smaller and smaller to the point that it becomes negligible. LSTMs were ideated in order to prevent such unfortunate behaviour. The basic idea is that by adding some parallel branchings along the computations, it can be possible to track what had happened in the history of the input sequence, so that it the vanishing gradient issue of the RNNs could be prevented.

This work put effort in obtaining a neural network that achieves a probability model that predicts words in a sentence. Perplexity is an evaluation metric commonly used by computational linguists in context of language models that indicates how good is the model at predicting the next words given a text sequence.

Current state-of-the-art perplexities on the Penn Treebank dataset can reach values up to 20.5 by Brown *et al*. [2], but have lots of parameters and require a model that has a complex structure, while here instead I show that even a simple neural architecture with far less parameters can achieve satisfactory results.

2. Problem statement

Language modeling is the task of predicting what word comes next. We use language models every day, look for example at the iPhone keyboard in Figure 1 or at the Bing suggestions in Figure 2.

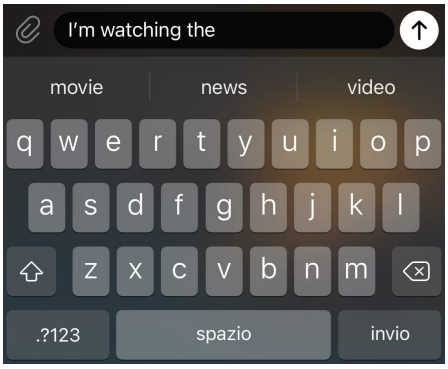

Figure 1. iPhone keyboard suggesting the next word.

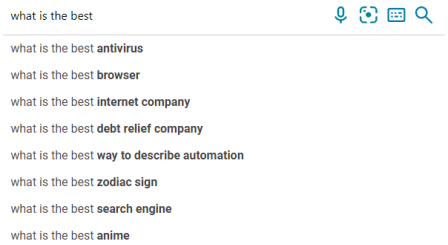

Figure 2. Bing search suggestions.

More formally: given a sequence of words $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$, compute the probability distribution of the next word $x^{(t+1)}$:

$$P\left(x^{(t+1)} \middle| x^{(t)}, \ldots, x^{(1)}\right) \tag{1}$$

where $x^{(t+1)}$ can be any word in the vocabulary $V = \{w_1, \ldots, w_{|V|}\}$ [10].

To evaluate a language model is necessary to compute the perplexity (PPL), the lower the better. The perplexity is defined as [10]:

$$PPL = \prod_{t=1}^{T} \left(\frac{1}{P_{LM}\left(x^{(t+1)} \middle| x^{(t)}, \ldots, x^{(1)}\right)} \right)^{1/T} \quad (2)$$

that is also equal to the exponential of the cross-entropy loss $J(\theta)$:

$$PPL = \prod_{t=1}^{T} \left(\frac{1}{\hat{y}^{(t)}_{x_{t+1}}} \right)^{1/T} \quad (3)$$

$$= \exp\left(\frac{1}{T} \sum_{t=1}^{T} -log\left(\hat{y}^{(t)}_{x_{t+1}} \right) \right) \quad (4)$$

$$= \exp\left(J(\theta) \right) \quad (5)$$

3. Data analysis

The dataset used is called Penn Treebank Dataset[1] (Taylor *et al*. [11]). It contains several english sentences, taken from manuals, journal articles, telephone conversations and others. For this work I have used only the world level dataset, so the one with the files names as: *ptb.(test|train|valid).txt*. Every line is a chunk of a sentence. The train dataset is composed of 42068 lines, while the test dataset is composed of 3761 lines. The vocabulary of the PTB dataset is capped at 10k unique words.

In this work, individual words are stored in a dictionary and sentences are put in a list. The network receives batches of sentences; each sentence is padded up to the maximum lenght of the longest phrase in the batch, to account for different lenghts. The padding element is defined as <*pad*> and will be excluded in the loss computation. More precisely, the network is fed with a sequence of words from a sentence, excluding the last one, and its output predictions have to match the ground truths which are all the elements from the corresponding sentence but the first one.

4. Model description

Instead of neurons, LSTM are built so that they resemble memory blocks. Their inner connections are depicted in Figure 3.

A basic block contains four layers, each with different purposes.

4.1. Forget layer

First there is the *forget gate layer*, which conditionally decides what information to throw away from the cell state. Figure 4.

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \quad (6)$$

[1] https://deepai.org/dataset/penn-treebank

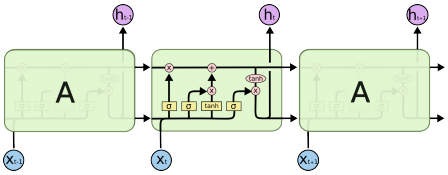

Figure 3. A single layer LSTM network [7].

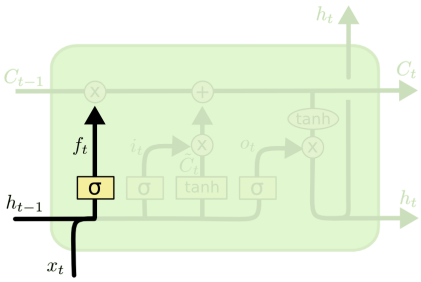

Figure 4. LSTM forget layer [7].

That corresponds to the following python code:

```
f_t = torch.sigmoid(torch.matmul(x_t_h_t1, self.W_f) + self.b_f)
```

4.2. Input layer

Then comes the *input gate layer*, that decides what new information can be stored in the cell state. Figure 5.

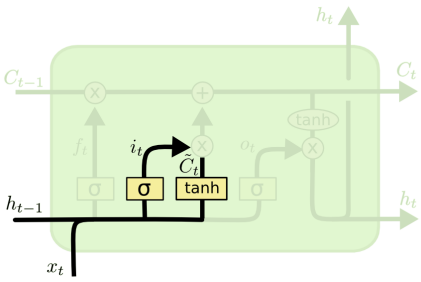

Figure 5. LSTM input layer [7].

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \quad (7)$$

$$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \quad (8)$$

That corresponds to the following python code:

```
i_t = torch.sigmoid(torch.matmul(x_t_h_t1, self.W_i) + self.b_i)
Ctilde_t = torch.tanh(torch.matmul(x_t_h_t1, self.W_Ctilde) + self.b_Ctilde)
```

4.3. Update cell state

Next the cell state can be updated with the decisions of the previous layers. Figure 6.

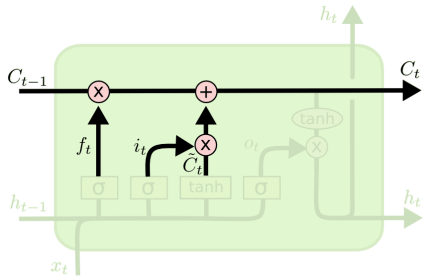

Figure 6. LSTM updating cell state [7].

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{9}$$

That corresponds to the following python code:

```
C_t = f_t * C_t1 + i_t * Ctilde_t
```

4.4. Output layer

Lastly, there is the *output gate layer* that decides what will be the output of the cell basing on its input and the current memory of the block. Figure 7.

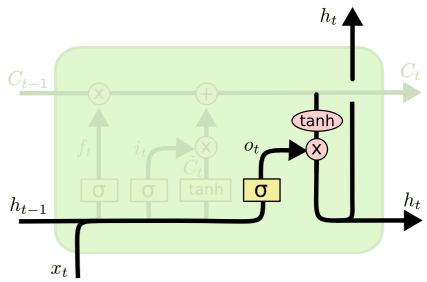

Figure 7. LSTM output layer [7].

$$o_t = \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) \tag{10}$$

$$h_t = o_t * \tanh(C_t) \tag{11}$$

That corresponds to the following python code:

```
o_t = torch.sigmoid(torch.matmul(x_t_h_t1,
self.W_o) + self.b_o)
h_t = o_t * torch.tanh(C_t)
```

4.5. Complete model

For each block and for each gate, there are several associated weights that will be learned during the training procedure. To obtain good performances in the language modeling task I implemented a two layers LSTM and the dropout as reccomended by Zaremba *et al.* [13]. In Figure 8 is possible to see a two layers LSTM. The dashed arrows indicate connections where dropout is applied and solid lines indicates connections where dropout is not applied.

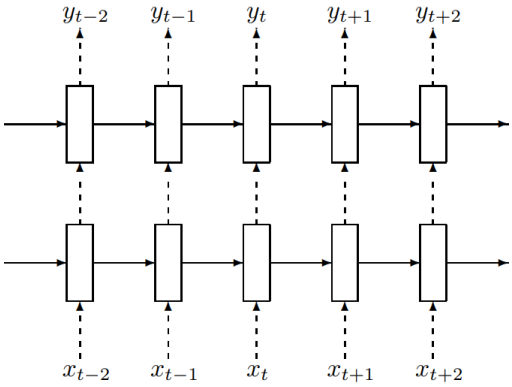

Figure 8. Two layers LSTM [13].

The network I designed is based on the parameters in Table 1 that are in part taken from [13]. The LSTM parameters are initialized uniformly in the range $[-0.05, 0.05]$ [13].

Parameter	Value
Layers	2
Hidden size	512
Input size	512
Batch size	64
Dropout	0.5
Clip	0.25

Table 1. LSTM network parameters.

5. Source code

In this section I will explain a bit the source code that I wrote. It's important to know that I took a cue from some open source codes. I will cite every source during the explanation.

5.1. Dataset

In this file the three different datasets are loaded and tokenized. The files are read line by line so sentence by sentence; the sentence are then splitted in correspondence of the space character; then every new word is added to the dictionary to have a conversion word to integer. At the end a list of sentences is returned.

To write this part I was inspired by [3] but the code is quite different because I wanted to get each sentence from start to finish whereas Yunjey Choi split the text every *batch_size* words.

5.2. LSTM

In this file I implemented a custom LSTM cell. The cell need to cycle over the input for all its length (the sentence length). Inside there are the necessary calculation as explained in Section 4.

To wrote this part I initially took part of the code from [4] but then I delete the code and rewrote all the class from scratch basing myself on the formulas of the LSTM cell.

5.3. Model

In this file the encoder and decoder for word embedding [9] are created. This code is necessary also to instantiate the LSTM cells and cycle over the various layers of the network.

5.4. Main

In this file multiple parts are present. All the code has been written by myself even if at the beginning I looked a bit at this code [3].

In the first part the parameters of the network are defined. In particular there is *is_training* to set the model in training or testing mode, and *my_lstm* that select to use the custom implemented LSTM cell or to use the PyTorch one.

Then there is some code that create the batches. It start from getting the sentences from the dataset and then padding the shorter sentences with the <*pad*> word to match the length of the longest sentence.

The next function *evaluate* is necessary to calculate the loss and perplexity on validation and test data.

Then I wrote *train* function that use the batch of data to train the network. The sentences are passed to the network, the loss is computed and then the optimizer step to update the weights.

The function *generate_sentences* generate some sentences starting from random words taken from the dictionary. The sentences are then stored in the *sample.txt* file.

At the end there is the code that is in charge of instantiating the model, calling the other functions and make a log file.

6. Results

Doing an hyperparameters tuning I found out that the network works well with the parameters reported in Table 1. The model has been trained on a RTX 3080 for 50 epochs taking around 3 hours. I have obtained a test perplexity of 105.2 with the model of epoch 26 (the best one). In Figure 9 the training perplexity and validation perplexity, during the training, are shown.

7. Conclusion

The LSTM cell that I have implemented works well because it gives the same results as the LSTM cell provided by PyTorch. At the same time it is way slower than the one provided by PyTorch. In fact the PyTorch LSTM is mainly coded in C with only a Python interface and the calculations are optimized.

Model	Author	Year	PPL
GPT-3	Brown [2]	2020	20.5
BERT-Large-CAS	Wang [12]	2019	31.3
Mogrifier LSTM	Melis [5]	2019	44.9
LSTM	Zaremba *et al.* [13]	2015	78.4
GRU	Bai *et al.* [1]	2018	92.5
RNN	Pscanu *et al.* [8]	2014	107.5
2 Layer LSTM	Lorenzini	2021	105.2

Table 2. Results comparison.

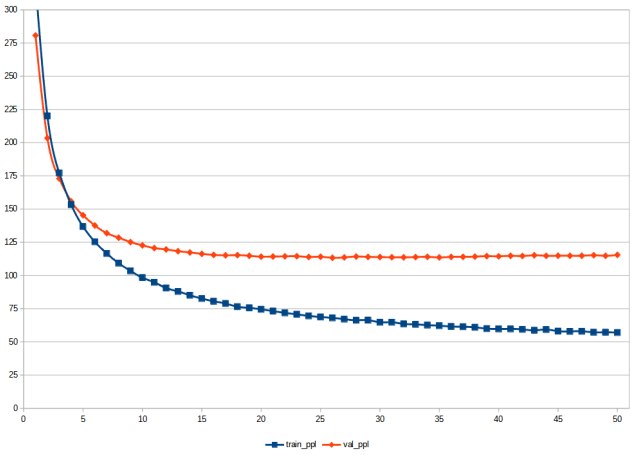

Figure 9. Training process. Blue: training perplexity. Red: validation perplexity.

The network performances are not so good, also after trying to apply some techniques found here and there, in particular in respect to Melis [5] work. In any case the Mogrifier cell he proposed has a different structure in respect to the base LSTM and so it's to be expected to obtain worse performances in respect to Melis work.

8. Sentences generated by the LSTM

Here are some sentences that have been generated by the network.

the panamanian democrats who had been a <unk> public hearing is likely to expand if congress <unk> it

in the $ N million stock price bonds and maturity bonds

a special profit and proposed gain from the sale of the two companies with its capital requirements <unk> merrill lynch & co.

use as <unk> reserves open to earthquake damage before the san francisco earthquake will show structural damage

References

[1] Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent net-

works for sequence modeling, 2018. 4

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. 2020. 1, 4

[3] Yunjey Choi. Pytorch language model tutorial. https://github.com/yunjey/pytorch-tutorial/tree/master/tutorials/02-intermediate/language_model, 2020. 3, 4

[4] Piero Esposito. Building a lstm by hand on pytorch. https://towardsdatascience.com/building-a-lstm-by-hand-on-pytorch-59c02a4ec091, 2020. 4

[5] Gábor Melis, Tomáš Kočiský, and Phil Blunsom. Mogrifier lstm, 2020. 4

[6] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. volume 2, pages 1045–1048, 01 2010. 1

[7] Christopher Olah. Understanding lstm networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015. 2, 3

[8] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks, 2014. 4

[9] PyTorch. Word embeddings: Encoding lexical semantics. https://pytorch.org/tutorials/beginner/nlp/word_embeddings_tutorial.html. 4

[10] Abigail See. Natural language processing with deep learning. https://web.stanford.edu/class/cs224n/slides/cs224n-2019-lecture06-rnnlm.pdf, 2019. 1, 2

[11] Ann Taylor, Mitchell Marcus, and Beatrice Santorini. The penn treebank: An overview. 01 2003. 2

[12] Chenguang Wang, Mu Li, and Alexander J. Smola. Language models with transformers, 2019. 4

[13] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization, 2015. 3, 4