LSTM for Model Language

Simone Luchetta (223716)
University of Trento
Via Sommarive, 9, 38123, Povo, Trento TN
simone.luchetta@studenti.unitn.it

Abstract

This work aims at giving an intuition of what can be done in order to implement both a GRU cell and an LSTM cell, which are the basic building blocks of a common RNN architecture. The concept is to build a neural network model that will provision the composition and the construction of sentences. The work makes use of the Penn Treebank text corpus. The code is developed using Python, and can be found inside a Google Colaboratory notebook. Then, networks' performances and a brief comparison between the two implementations in Pytorch are discussed.

1. Introduction

Recurrent neural networks (RNNs) are powerful tools that we need to understand. Whenever we make use of the internet's search engines, or whenever we type words on a smartphone we hiddenly use applications that leverages RNNs. In particular, it is known that a RNN is especially ideal for sequential problems, in which a point in a dataset is dependent on other points. This happens because with their particular architecture, RNNs are able to remember the analysis that was done up to a given point by constantly updating their internal "memory" blocks. For this reason, this kind of neural network finds its place tasks such as speech recognition, language translation, stock pricing prediction, and to a lesser degree in image recognition to describe the content in the pictures.

The goodness of the neural network developed in the code presented in this paper is accessed in terms of the perplexity score, the lower the better. In order for a RNN network to obtain slightly better generalization capabilities, it can be possible to follow the hints presented by Zaremba *et al*. [3].

This paper will be structured as follows: firstly, a refreshing overview of probability models for language modelling is discussed. Then, a walkthrough on how the basic building blocks of a simple RNN are built is provided. Lastly, a brief comparison amongst them is debated.

2. Probability models for LM

A word embedding can be thought as a vector that encodes the meaning of a given word. Substantially, sentences found in the treebank text corpus need to be converted into some numeric form. Word embeddings allows us to represent a word by reflecting its relations between similar other words, and they can be learned through all the data at our disposal in the corpus.

Neural networks can use embeddings of words to make word predictions and can hence represent words in a distributed way as non-linear combinations of weights.

Note that tipically, fully connected networks do not perform well when they have to deal with long sequences of data. For this reason RNN were deployed to overcome such poor behavior.

Usually, the task of most of the neural networks that have to deal with language modelling is to learn a probability distribution for each word, given its context:

$$\sum_{i=1}^{W} P(w_i | w_{i-k}, ..., w_{i-1}, w_{i+1}, ..., w_{i+k}) \qquad (1)$$

Where W is referring to all of the words in the corpus, k is an indice that represents other previous (or future) words.

However, instead of using neural networks to produce a probability distribution, it is also common to learn the aforementioned word embeddings which will constitute the parameters of the network itself. The network's embedding could then be manipulated so that they can turn useful to visualize concepts, but that is not strictly inherent from the scope of this work. There are two major approaches for learning word embeddings: one of them is called "count-based", in which the factorization of a global co-occurrence matrix of the words is computed; the other one is "context based" which observes the context of each word and extract embeddings as explained in the work of Mikolov in *et al*. [2].

3. Building blocks of an RNN

RNNs are used to model sequences, which is a kind of information put in a particular order in which one thing follows another one. Sequence data comes in vary forms. For example, audio is a natural sequence: we can chop up an audio spectrogram into chunks and feed it to a RNN. Text is another form of sequence: we can break up a sentence into a sequence of words.

RNNs are basically built by consequential memory blocks to kind of resemble the human behavior of processing informations through time. Basically, the RNN consists in a modified version of a fully connected node, in which a looping connection inside the hidden states of the neuron are added, so that the previous informations can affect later stages of processing, as can be seen in figure 1.

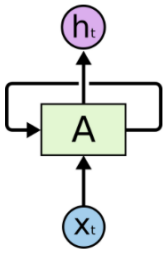

Figure 1. Basic RNN. The information in the loop is also called *hidden state*, which is a representation of the previous inputs

Though, this rudimental way of processing information, altough promising, suffers from the problem of vanishing gradients when implemented and ran in a network. The reason why this happens is because as the RNN processes the information, it becomes difficult to retain the information collected at the early stages. When performing the backpropagation step of the training, each node in a layer of the network calculates its gradient with respect to the effects of the gradient of a loss measure calculated at the layer before; therefore, if the adjustments of the gradient in the layer before are small, then they will cause the adjustments in further layers to be inevitably smaller; this causes the gradient to exponentially shrink as the gradient gets propagated backtrough.

So, because of the vanishing gradients, the RNNs are unable to learn long term dependencies. Over time there have been deployed different variants of the RNNs, namely the LSTM and the GRU, that make effort in getting rid of this problem and hence are capable of learning longer term dependencies using mechanisms called *gates*.

Figure 2 qualitatively allows us to see what happens inside the simple RNN depicted in figure 1.

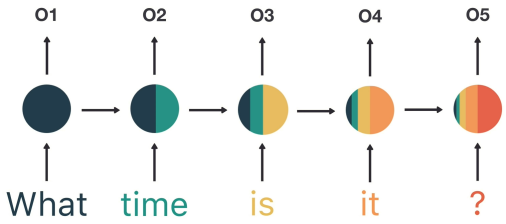

Figure 2. Sequence of text represented inside a simple RNN unit. It is clear that in the final step that the RNN has encoded informations from all the words in the previous ones.

3.1. GRU

GRU stands for *gated recurrent unit*. The gates inside this cell are described by the following mathematical behavior:

$$R_t = \sigma\left(X_t \cdot Reset_X t + hstates \cdot Reset_H t\right) \tag{2}$$

$$Z_t = \sigma\left(X_t \cdot Update_X t + hstates \cdot Update_H t\right) \tag{3}$$

$$Q_t = R_t \cdot hstates \tag{4}$$

$$H1_t = \tanh\left(X_t \cdot Output_X t + Q_t \cdot Output_Q t\right) \tag{5}$$

$$H2_t = (1 - Z_t) \cdot hstates \tag{6}$$

$$H3_t = Z_t \cdot H1_t \tag{7}$$

$$H_t = H2_t + H3_t \tag{8}$$

Please refer to picture 3 comparing the formulas.

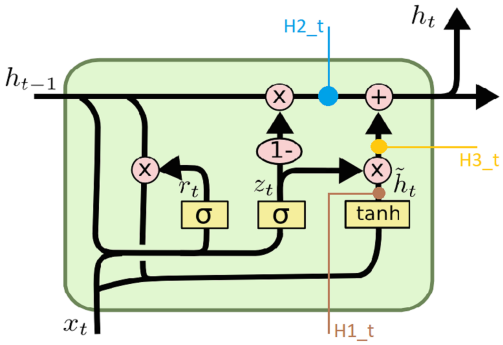

Figure 3. Gru cell building block

Down in terms of basic components, the GRU is built around two main gates: the *reset* gate and the *update* gate.

The reset gate allows us to control how much of the previous state we might still want to remember. The *Rt* component can be easily obtained through Eq. (2). This gate helps to capture the so called short term dependencies.

On the other hand, the update gate allow us to control how much of the new state is just a copy of the old state. This leads us to *Zt* which is obtained through Eq. (3). This gate is engineered so that long term dependencies can be captured inside the cell.

Then there are other components, such as the *candidate hidden state* named $\tilde{H}_t$ on the right in picture 3; recall that these are just mathematical supports that are then combined together so to arrive at Eq. (8), which determines which will be the new hidden state *Ht*.

3.2. LSTM

LSTM is the abbreviation of *long short term memory*. Equations that describe how the cell behave are presented as follows:

$$F_t = \sigma\left(X_t \cdot U_i + H_t \cdot V_f + B_f\right) \tag{9}$$

$$I_t = \sigma\left(X_i \cdot U_i + H_t \cdot V_i + B_i\right) \tag{10}$$

$$G_t = \tanh\left(X_t \cdot U_c + H_t \cdot V_c + B_c\right) \tag{11}$$

$$O_t = \sigma\left(X_t \cdot U_o + H_t \cdot V_o + B_o\right) \tag{12}$$

$$Ct = Ft \cdot Ct + It \cdot Gt \tag{13}$$

$$Ht = Ot \cdot \tanh Ct \tag{14}$$

Please refer to picture 4 comparing the formulas.

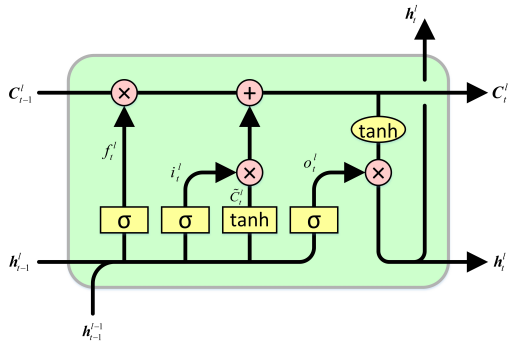

Figure 4. LSTM cell building block

The LSTM cell composes of three gates, namely the *forget gate*, the *input gate* and the *output gate*.

The forget gate is responsible for removing information from the cell state. This gate takes in a pair of inputs, which are the input vector of the LSTM unit and the hidden state vector at a given time-step; these two inputs are multiplied by the weight matrices and then the bias is added, as shown in Eq. (9).

The input gate is responsible for the addition of information to the cell state. All happens during a three step process: firstly, *It* is computed as in Eq. (10). Then, the same goes for *Ct* as described in Eq. (13). Then the two are combined and the result leads to the amount of useful information that is added to the cell state.

The output gate can again be thought as a three step process: values are scaled to the range [-1 , 1] thanks to the *tanh* function. Then, *Ot* is computed as described in Eq. (12). The two values are then multiplied as in Eq. (14) and the output is branched into two, forming the two *Ht* pairs on the right depicted in figure 4.

Note that every *sigmoid* operation can be thought as a filter, which its outputs being 0 mean that the information is not particularly relevant, and being 1 means that there might be useful information instead.

4. Networks architecture

The architecture of the networks used in this paper is depicted in figure 5

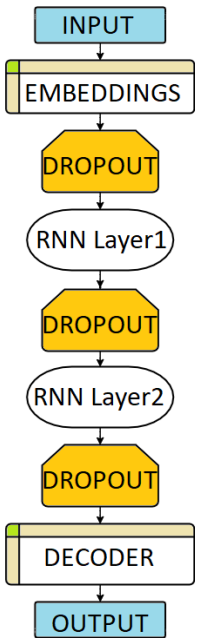

Figure 5. Network architecture

Firstly, word embeddings are being stored inside a lookup table structure (*nn.Embedding*). Then, dropout is applied to these embeddings since in many models they

contain the majority of the total parameters (and hence might be responsible for a lot of overfitting).

After this, a pair of LSTM or GRU layers, one stacked on top of the other is inserted. As mentioned in previous paragraphs, these layers are accountant for learning the input sequence dependencies. A dropout layer is interleaved within each of the LSTM's output layers, so to increase the generalization capability.

Finally, a fully connected layer takes in the processed word sequences and decodes them back into the original word space, which contains about 10000 words.

Softmax is then applied so to rescale the so-obtained tensor.

The applied optimizer during the training process is *Adam*, that will update the network's parameters basing on the computed gradients.

The criterion for the loss is the *negative log likelihood*.

5. Results

The code is written in Python, taking over PyTorch's framework. The whole code is available within a separate .ipynb notebook on Google Colaboratory, and either the GRU and the LSTM classes can be found. The LSTM for this particular dataset has an edge over the GRU, achieving better performance. Training perplexity reaches around 85.6 points on *val corpus* sentences, which is a remarkable result considering the fact that the implementation of the cell itself is not something very hard to come up to.

Here are the hyperparameters used during training are reported here below in table 1.

If using the LSTM cell, performance could be achieved if slight changes are applied to the cell itself, as in Melis *et al*. [1].

Hyperparameter	Value
Embeddings	650
Hidden layers	650
Learning rate	$0,001$
Gradient clip	$0,35$
Dropout	$0,5$
RNN Layers	2

Table 1. LSTM hyperparameters

Odds are to encounter overfitting as the training process goes on for very long epochs. To face this issue, it is possible to recur to other overfitting prevention techniques such as placing some fully connected layers before or after the RNN layers, or changing the values of the embedding size and the hidden layers sizes. Placing more fully connected layers though increases the computation time required by the network to update the parameters (up to 383 seconds per epoch running on Google's GPU) and while it could help in becoming less prone to overfitting, it might not entirely get rid of the problems at all.

However, a brief experiment has been conducted, consisting in placing a pair of fully connected layers between the last RNN layer and the output. This helped in achieving a better perplexity performance, arriving at 70.64 on *val corpus* after 20 epochs; perplexity on *test corpus* reaches 77.04.

Note that there is a leap in performance whenever evaluating on *test corpus* or *val corpus*, and it might be due to the fact that these set may have a different distribution of the data.

In any case, it is way better with respect to before, where only 85.6 was obtained.

6. Code

Hereinafter the various parts of the code are presented. Refer to the github repository language_modeling_lstm which has been used as a backbone structure.

6.1. Load corpus

This portion of code assures that all the sentences in the Penn Treebank dataset are being loaded inside dictionaries. The dictionary contains another object in position zero, which is called *<pad>*. The *Dictionary* class remains untouched, as it is pretty simple. On the other hand, the *tokenize* function of the *Corpus* class has been changed drastically.

	Coded by me	Taken from a guide
Percentage	50	50

6.2. LSTM Cell

This portion of the code contains the structure of the building element of the LSTM. Refer to pytorch-lstm-by-hand by piEsposito, that provides a quick guide on how to construct the cell. Notice that he initializes the weights from *nn.uniform_*, but They could also be initialized differently (CTRL + F, "Fills" @ following link: Tensors) I tried to use *nn.log_normal_* to see if there were any improvements, but it worsened the learning by a lot. Other than that, parameters for every gate have been provided. Note that the output of the cell has been changed a little to match with what they'll be used for later on.

	Coded by me	Taken from a guide
Percentage	20	80

6.3. GRU Cell

Once becoming familiar with the LSTM Cell, I decided to implement my own GRU Cell without looking at nothing

else but figure 3. The math might not be very well optimized but it gets the job done as explained before in the paper.

	Coded by me	Taken from a guide
Percentage	100	0

6.4. Model

GRU or LSTM elements appearing in the previous subsections are instantiated herein. The model differs from language_modeling_lstm by the fact that forward procedures are changed, outputs are passed from an RNN layer to the other and there is an intermediate fully connected layer that helps generalization.

	Coded by me	Taken from a guide
Percentage	50	50

6.5. Main

All previous classes appearing before are instantiated here. The optimizer, the criterion for the loss metric, the *get_batch* function, the *evaluate* function, and the *train* function have been completely redesigned in order for them to work as I wanted to.

	Coded by me	Taken from a guide
Percentage	60	40

References

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

[2] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality, 2013. 1

[3] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization, 2015. 1