# Implementation of a Recurrent Neural Network for Language Modelling

Diego Planchenstainer (223728)
University of Trento
Via Sommarive, 9, 38123 Povo, Trento TN
d.planchenstainer@studenti.unitn.it

## Abstract

*This work focuses on the implementation of a recurrent neural network for language modelling, in particular on the construction of the Long Short Term Memory (LSTM) cell, the core of the network. The model is trained on the Penn Treebank dataset and its performances are evaluated with the perplexity measure. In this work some dropout techniques from the literature are examined to see if they improve the results. Also, it is inspected the effects of the identity connection inspired from ResNet.*

## 1. Introduction

Recurrent neural networks are a simple but effective tool thanks to their ability to model and learn from sequential data. In 2015 they achived state-of-the-art results in different scenarios, one of them is language modelling, which this paper will be interested in. Language modelling (LM) is one of the many applications of NLP and it is used to generate text as an output. Possible applications include question answering, machine translation and parts-of-speech tagging. Everyday we come in touch with such applications, think for example to the suggestion when searching online or the autocomplete function in the smartphone's keyboard. Language modelling consists in determining the probability of a given sequence of words occurring in a sentence and predicting the most probable next word, but this topic will be discussed in the next section.

So, what are RNN? Recurrent neural networks are a class of neural network that allows previous outputs to be used as inputs while having hidden states. The main advantages of these models are their capability of processing input of any length and taking into account historical information. Beyond being slow in computation and having difficulty to access very old information, the main problem that affects RNNs is the vanishing/exploding gradient. This problem arises when trying to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers. To deal with this problem the gradient clipping technique was introduced, it consist in capping the maximum value for the gradient to control the problem. Another way to reduce this problem was found with the introduction of Long Short-Term Memory (LSTM) cell. This cell is an upgrade of the vanilla RNN proposed by Elmann [2] and it consists in adding different gates to optimize its functions, those gates will be explained in depth later in 4.1. Perplexity is the most used evaluation metric for language model and is chosen over others because is task-unrelated, so it evaluates the "goodness" of the model itself. This kind of evaluation is called intrinsic evaluation and is useful to quickly compare models. The language modelling state of the art performance achived on Penn Treebank dataset is currently 20.5 ppl, obtained by GPT-3 model [1] with 175 trillion parameters. When talking about LSTM the best performance is currently achived by the Mogrifier LSTM [7] and is equal to 44.8 ppl. The Mogrifier LSTM is a considerably very small model with respect to GPT-3, in fact it has 24 million parameters. This implies that even not so complex models are capable of good performances.

## 2. Problem statement

Language modelling aims at modelling the probability of a given sequence of words occurring in a sentence. It can be used either to generate new sentences or to estimate how natural a sentence is. The language model compute the probability of a word occurring in a sentence by considering the previous words. There are different kinds of models, the most simple is the unigram model. Here, the probability of a each word depends only on the own word probability, formally:

$$P_{uni}(w_1, w_2, w_3) = P(w_1)P(w_2)P(w_3)$$

A more sophisticated model is called $n$-gram. Here, the probability of the occurence of a word is approximated as:

$$P(w_i|w_1, ..., w_{i-1}) \approx P(w_i|w_{i-(n-1)}, ..., w_{i-1})$$

So the probability of that particular word is influenced only by the $n$-words that occur before.

The next model use embedding of words to make predictions, here neural networks should be used, as in this report. Word embeddings are a representation of words that encodes their meaning into a vector such that words that are closer in the vector space are expected to be similar in meaning. This encoding helps to reduce the curse of dimensionality. Word embeddings avoid the problem of sparseness that afflicts one-hot encodings by representing words in a distributed way. Combined with this method different neural architectures could be used for this task, but recurrent ones are more common.

### 2.1. Perplexity

The goodness of the model can be evaluated by computing the perplexity, less perplexity implies better results. There are two possible equivalent way to compute it. The first is to consider perplexity as the normalized inverse probability of the test set:

$$PPL(W) = \sqrt[N]{\frac{1}{P(w_1, w_2, ..., w_N)}} \tag{1}$$

where $W$ is the test set and $N$ is the number of token. Perplexity can also be computed as the exponential of the cross-entropy loss between the predicted probability distribution and the ground truth. Here:

$$PPL(W) = e^{H(W)} \tag{2}$$

The cross-entropy is equal to:

$$H(p, q) = -\sum_i p(x_i) \log q(x_i) \tag{3}$$

where $p()$ is the real probability distribution of the language and $q()$ is the distribution estimated by the network. The real distribution is unknown but can be approximated given a long enough sequence $W$ of words ($N$ long sequence) as: $1/N$. So equation 2 turns into:

$$PPL(W) = e^{-\frac{1}{N} log P(w_1, w_2, ..., w_N)} \tag{4}$$

By simplifying the exponential with the logarithm in eq. 4 the equivalence with eq. 1 can be accessed.

## 3. Data analysis

The dataset that has been used is the Penn Treebank dataset. It is widely used in NLP machine learning applications and is a relative small dataset given the fact that it has 10k unique words. Furthermore, the word-level dataset does not contain numbers, punctuations or capital letters. This dataset contain sentences taken from different domains such as journal articles, manuals, et cetera. Such heterogeneous composition force to learn each sentence separately form the others in order to capture only the semantic of that particular sentence.

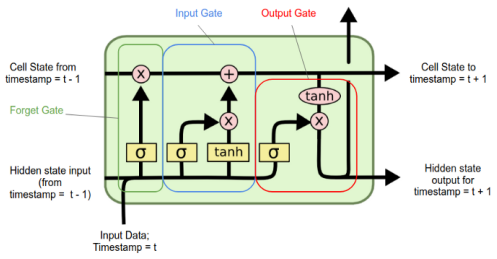


Figure 1. LSTM cell architecture [6]

## 4. Model description

The model consist in a first embedding layer, a multilayer LSTM cell and a fully connected layer.

### 4.1. LSTM cell

The LSTM cell is the core of the network. It consist in three structures called gates and require an input, which could be the previous hidden state, an hidden state and a cell state, it then returns the new hidden and cell state. The overall structure of the inside of the cell can be seen in fig 1

The first gate is called forget gate and is responsible to select which information has to be "forgotten" in the cell state. It takes in input the previous hidden state and the current input and compute the output as shown in equation 5:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{5}$$

Then $f_t$ is multiplied with the previous cell state to "forgot" the selected information. A speed up trick has been used with the initialization of the forget bias: "It can take a while for a recurrent network to learn to remember information form the last time step. Initialize biases for LSTM's forget gate to 1 to remember more by default." [4]

The next gate is the input gate, its task is to select the new information that should be stored in the cell state. The input gate is compose by the sum of $i_t$ and $\tilde{C}_t$. The first is computed as in equation 6, the second as in equation 7:

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \tag{6}$$

$$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \tag{7}$$

To store the result of this gate the cell state is updated as:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{8}$$

The last layer is called output layer and its function is to compute the output of the cell combining cell state with input and previous hidden state. The equation that describe the functioning of this gate is shown in 9

$$h_t = \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) * \tanh C_t \tag{9}$$

Here $h_t$ will be the updated hidden state vector which will be used in the next step of the computations.

### 4.2. Regularizers

Since RNNs are prone to overfitting different dropout techniques have been adopted from the literature. Zaremba *et al*. (2014) [9] proposed to apply dropout only to non recurrent element of the network, since they stated that recurrent elements need to remember information. This led to a major improve in the performances, thus it has been applied. Later in 2015/16 different works proposed to apply dropout even to the inner components of the cell, in particular Semeniuta *et al*. [8] analyzed two different dropout techniques at the cell level and explored another approach. Their idea was to apply dropout to the result of $i_t * \tilde{C}_t$ and they demonstrate the effectiveness of this approach. In their paper Semeniuta *et al*. supposed that the mixture of their dropout with the dropout proposed by Gal *et al*. (2015) [3] should give the best performance. Unluckily, this result did not match with the one obtained as without internal dropout the network reaches 104 ppl and with dropout the result rises till 116 ppl.

### 4.3. Residual connection

The residual connection introduced in ResNet by He *et al*. [5] allows deeper model to perform better by ensembling shallower models. Since a 3-layer LSTM cell has been tested and proved to be worst than the 2-layer cell, it was reasonable to verify if the residual connection could improve the performances of the 3-layer cell. Even though the residual connection did not help the 3-layer model to outpace the 2 one, it improved the results in both scenarios. The 2-layer model still remains the best one, the residual connection contributes to a slight upgrade in the performance equal that allows to reach 102 ppl.

## 5. Result

The backbone of the code can be found at: word language model github repo.
The network is composed by an embedding layer, a 2-layer LSTM cell and a linear "decoder" layer. Then several other configuration have been tested. Even a 3-layer cell layout was tested, but proved to behave worst than the two one. A comparison between their results can be seen in fig. 2 for the 2-layer models. For viewing purposes the first and the second epochs of the graph are omitted.

In fig. 2 there is the comparison of the perplexity obtained by the 2-layer model in the following configurations:

1. standard
2. forget bias init
3. residual connection
4. internal dropout
5. forget bias init + residual connection

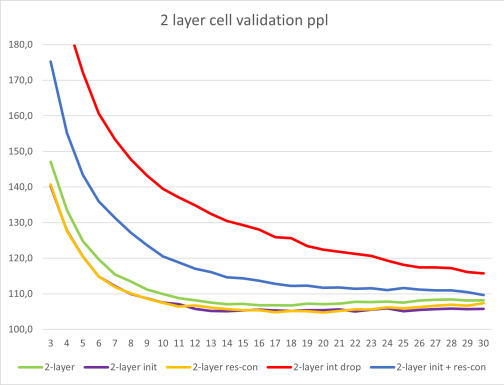


Figure 2. 2-layer cell graph comparison

Two particular observation can be made:

1. internal dropout seems not to work as intended, slowing down the learning process and making difficult to reach better results.
2. even though the initialization to the forget bias and the residual connection help respectively to speed up the learning process and upgrade the result, their combination returns worst results.

The best result is achived by a 2-layer cell with the residual connection that is able to obtain 102 ppl on Penn Treebank dataset and generate sentences that make sense more or less. The model took circa 8 minutes per epoch to train with Adam optimizer on a GTX 1050 and was trained for 30 epochs. The hyperparameters of the network are reported in Table 1.

| Hyperparameter | Value |
|---|---|
| Hidden dimension | 600 |
| Dropout | 0.5 |
| Learning rate | 0.001 |
| Gradient clipping | 0.25 |
| Batch size | 64 |
| Evaluation batch size | 1 |
| Weight decay | $1.2e^{-6}$ |

Table 1. Hyperparameters value

In order to generate sentences the <sos> character is feed in to the network and as long as the sentence generates new words the input is updated with them to let the

network capture the context of the sentence. Everytime that an <eos> character occurs the network deletes the previous inputs and restart with a new sentence.

## References


[1] 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. *CoRR*, abs/2005.14165, 2020. 1

[2] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990. 1

[3] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks, 2016. 3

[4] Danijar Hafner. Tips for training recurrent neural networks. Blog post, 2017. 2

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. 3

[6] Ryan T. J. J. Lstms explained: A complete, technically accurate, conceptual guide with keras. 2

[7] Gábor Melis, Tomás Kociský, and Phil Blunsom. Mogrifier LSTM. *CoRR*, abs/1909.01792, 2019. 1

[8] Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. Recurrent dropout without memory loss. *CoRR*, abs/1603.05118, 2016. 3

[9] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014. 3