Recurrent Neural Networks for Language Modeling*

*NLU final project - ML2

Raffaele Pojer
mat. 224012
University of Trento
raffaele.pojer@studenti.unitn.it

***Abstract*—In this project, the aim was to implement an RNN Language Model using one of the famous architectures. In this work, I decided to implement several architectures (LSTMs and GRUs) with different characteristics. In the beginning, I started with an LSTM cell and then with a faster implementation that saves time during training. The proposed architectures make heavy use of regularization with dropout and layer normalization. With this work, I will try to make a comparison between the various implementation. The code was written and trained using Google Colab.**

I. Introduction

It has been shown [1] that neural language models outperform standard backoff n-gram models thanks to the smoothing implicitly solved via back-propagation.

A. Recurrent Neural Networks

In natural language understanding, language generation and many other sequence-based tasks, recurrent neural networks (RNNs) models are frequently used. Differently from typical feedforward neural networks where the history is represented by the context of $N-1$ words, RNNs recurrent connections give the possibility to form short term memory to deal better with position invariance. Recurrent networks have an input layer x, a hidden layer s (also called context layer) and an output layer y. The input vector $x(t)$ is formed by concatenating vector w representing the current word and a vector from the context layer at the layer $t-1$.

Figure 1 shows a simple example of a RNN. Here I will focus on the problem of comparing different types of RNNs with different settings on the task of language modeling.

II. Problem Statement

Given a sequence of words of length n, the probability of the whole sequence is $P(w_1, \ldots, w_n)$, a language model is a probability distribution over a sequence of words. Recurrent neural networks Language Models make use of the chain rules to model the joint probability of a sequence of words, more formally:

$$P(w_1, \ldots, w_N) = \prod_{i=1}^{N} p(w_i \mid w_1, \ldots, w_{i-1})$$

where the context of all the previous words is encoded in a LSTM or GRU cell. This type of problem is called *n-gram* model, where the probability of a word depends by the *n-words* occurred before.

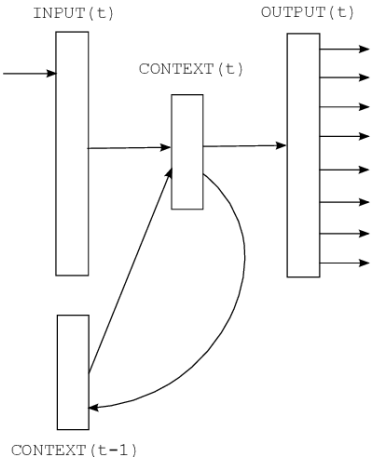

Fig. 1. Simple recurrent neural network taken from [2].

III. Model

In this section, I will describe in detail the various models adopted for this project, specifically the equation used for computing the output for each cell.

A. LSTM

One of the typical problems with RNN is the vanishing of the gradients, this phenomenon happens during backpropagation when the partial derivatives of the weights of the network start to become too small. Small gradients prevent the model to learn since that the little changes are ineffective to converge in some local minima. LSTM cells were created to mitigate this problem, introducing an internal cell c_t that is not subject to matrix multiplication or squashing. With these new connections gradients can flow better, as described in [3]. The LSTM cells implemented in this project are from [4]. Each element $x = (x_1, \ldots, x_t)$ passes thorough the cell with

the following equations, from t = 1 to T, producing another sequence $y = (y_1, \ldots, y_t)$.

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \quad (1)$$
$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \quad (2)$$
$$g_t = \tanh(W_{ig}x_t + b_{ig} + W_{hf}h_{t-1} + b_{hg}) \quad (3)$$
$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \quad (4)$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \quad (5)$$
$$h_t = o_t \odot \tanh(c_t) \quad (6)$$
$$y_t = W_{yh}h_t + b_y \quad (7)$$

To speed up the training, I implemented another LSTM cell to reduce the number of matrix multiplications. Different frameworks use similar techniques in a more general implementation. Here I develop a specific adaptation for each cell. The idea is to embed the weights involved in the learning, into two matrices W_i and W_h of dimensions *input-size* by *(4*hidden-size)*. The computation for the gate can be done with:

$$\big(\begin{pmatrix} x_t \end{pmatrix} \cdot W_i\big) + \big(\begin{pmatrix} h_{t-1} \end{pmatrix} \cdot W_h\big) \quad (8)$$

The result of **8** is then divided in 4 equal parts along longest dimension and assign each part for the *input, forget, output* and c_t cell. This four parts are then passed through a *sigma* or a *tanh* according to the type of function.

B. GRU

GRU cells are another type of RNNs, details of GRU can be found in [5]. The characteristic for this cell is the removal of the state cell c_t, and a reduction in the number of gates. GRUs have two gates, the *reset gate*, used for the model to decide what to forget from the past information, and the *update gate* used to decide how much of past information can be carried on in future steps. As in LSTM, this cell compute each input element $x = (x_1, \ldots, x_t)$, from *t*=1 to *T*. The equations used are:

$$r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{t-1} + b_{hr}) \quad (9)$$
$$z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{t-1} + b_{hz}) \quad (10)$$
$$n_t = \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{t-1} + b_{hn})) \quad (11)$$
$$h_t = (1 - z_t) * n_t + z_t * h_{t-1} \quad (12)$$
$$y_t = W_{yh}h_t + b_y \quad (13)$$

The cell was implemented directly using the method which speeds up and reduces the number of matrix multiplications. Given two matrix W_i and W_h of size *input-size* by *(3*hidden-size)*, the gate computation is made similar to equation **8**, but with the new dimensions. The result is then divided into 3 parts for the respective gates or inner parts.

C. Comparisons

To compare the two cells versions, in particular LSTM, I averaged over 50 sentences without randomizing them in the dataset. The result is that the LSTM cell with less matrix multiplication is approximately 22% faster than the LSTM "non-optimized".

- LSTM non-optimized: 26.61 seconds
- LSTM optimized: 20.81 seconds

Also, the training is faster due to the parameter sharing involved in only one matrix differently from the version where all the weights of each gate are disjoint from the others. All the models have the possibility to disable the bias term during the computation to reduce the computation cost, it was not tested if produce better or worse results..

IV. Regularization

It is known that recurrent neural networks can easily overfit during training, to mitigate this problem I decided to rely on strong regularization as in [6]. Here the idea is to apply regularization to the non-recurrent connections to avoid overfitting. Another simple technique adopted to reduce overfitting is to shuffle each batch of data to make the gradients more variable. Considering that the size of the hidden features for each cell was quite high, the amount of dropout used was the same as in [6].

V. Layer Normalization

To reduce training time and increase training stability, batch normalization is highly used in several deep learning problems to reduce the internal covariance shift. Layer Normalization [7] directly estimates the normalization statistics from the summed inputs to the neurons within a hidden layer. The normalization does not introduce any new dependencies between training cases. To compute the layer normalization I used the `torch.nn.LayerNorm` of PyTorch which relies on the following equations:

$$y = \frac{x - \mathbf{E}[x]}{\sqrt{\text{Var}[x] + \epsilon}} * \gamma + \beta \quad (14)$$

From now on, *gamma* and *beta* parameters are called element-affine terms as in the PyTorch documentation, that can be set True or False.

VI. Pre-Padding

The dataset was preprocessed by adding at the start of each sentence the `<sos>` tag with the `<eos>` tag at the end. Along with these two tags, all the input sentences are padded with the `<pad>` tag to make all sentences of the same batch the same length. [8] demonstrates how by adding the padding at the start of the sentence, it is possible to obtain higher accuracy during the training. In the code, I also add the possibility to remove the pre-padding and the padding for further investigation. I notice almost immediately the effectiveness of these methods by running smaller tests. This tests, remove or add the padding and pre-padding during training.

VII. Results

I conducted different experiments with different models on the Penn Tree Bank (PTB) dataset. This dataset consists of 929k training words, 73k validation words and 82k test words in a total of 10k words in the vocabulary. The sentences of the dataset were preprocessed using the pre-padding. All the cells (LSTM and GRU) has 1500 units per layer, with the bias weight present. The parameters are initialized from $\upsilon = (-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{hidden\_size}$. The hidden states are initialized to zero. The size of each mini-batch is 64, and on the non-recurrent layer, it was applied a 65% dropout. All the networks were trained with a learning rate of 0.001 and after 14 epochs it was reduced with a factor of 1.12 at each epoch. The size of the batch for the validation is 32. **Figure 2** shows the training and validation perplexity during the 30 epochs with a LSTM and layer normalization. The best result was obtained at epoch 22 with a perplexity of 87.55. The gamma and beta value for the layer normalization was not used, since they do not help in a significant way the training. At the end of the training, the model was tested with a batch size equal to one to have better results that do not include extra predictions, this also saves time during training. In the latter, the perplexity obtained was 108.8.

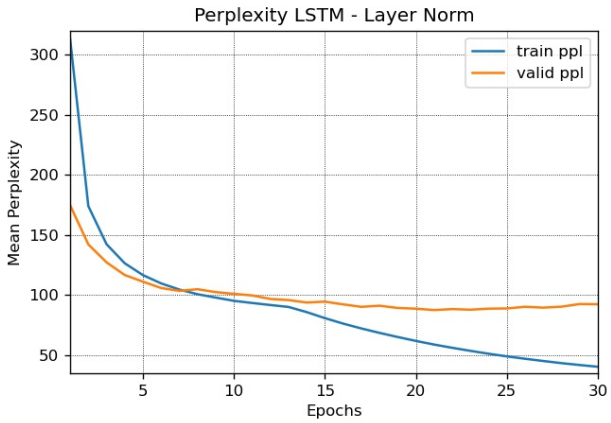

Fig. 2. Perplexity of an LSMT cell with layer normalization (*element-affine* parameter set to False) during 30 epochs of training

In **Figure 3** is possible to see the differences between various settings of a LSTM cell. The network that uses layer normalization, obtained the best result much faster compared with the LSTM without layer normalization or LSTM with layer normalization and the gamma and beta value (*element-affine* parameters).

Finally **Figure 4** shows the difference between GRU and LSTM with layer normalization. After 26 epochs the best perplexity achieved from GRU was 93.73.

A. Proof of learning

To be sure that the model has learned something, I tried to generate the next words of a phrase by simply taking the *argmax* from the *softmax* in the last cell and continuing to iterate until the <eos> token is generated. Here I report some phrases generated by the LSTM network that obtained the best results, after 30 epochs of training:

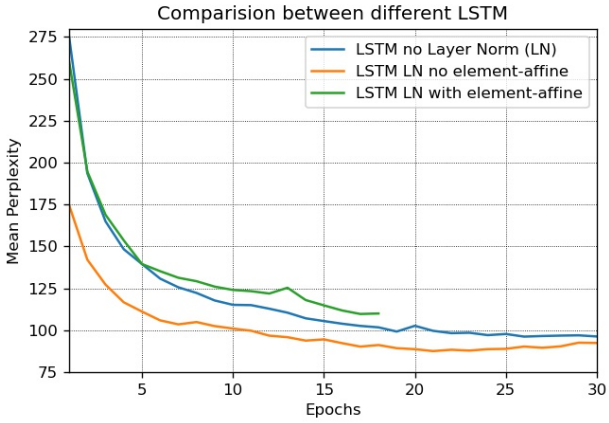

Fig. 3. This plot shows how layer normalization can impact during training, the network converges much faster compared to the others

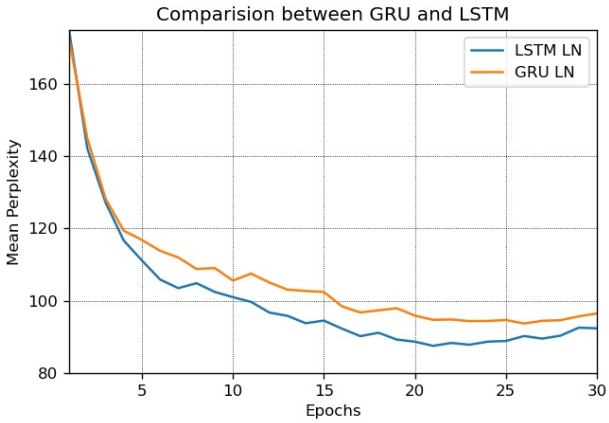

Fig. 4. Comparison between LSTM and GRU with layer normalization. LSTM cell still produces better results with this settings.

Generated phrases	
Input	**Output**
the weather	in the san francisco area
it is possible to have higher	debt and other businesses and <unk>
we expect	to be <unk>
house democrats	and the senate
new york is	the largest in the <unk>

It is possible to see that the phrases generated have some meaning with the context of the input one, i.e. *"house democrats and the senate"*.

VIII. Conclusions

This project demonstrates the difficulty to train a recurrent neural network for the task of language modeling. There are plenty of settings to set up to achieve great results that can be further explored (i.e. the type of cells used, the regularization adopted, etc.). Surely there are various parameters to be set

better but the constraints of coding in a notebook with a free version of Google Colab limits the flexibility of the project. Overall the achieved results are good and show the power of RNNs for this task when combined with regularization methods.

REFERENCES

[1] T. Mikolov, S. Kombrink, L. Burget, J. Černocký, and S. Khudanpur, "Extensions of recurrent neural network language model," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2011, pp. 5528–5531.

[2] T. Mikolov, M. Karafiát, L. Burget, J. H. Cernocký, and S. Khudanpur, "Recurrent neural network based language model," in *INTERSPEECH*, 2010.

[3] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[4] H. Sak, A. Senior, and F. Beaufays, "Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition," 2014.

[5] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," 2014.

[6] O. V. Wojciech Zaremba, Ilya Sutskever, "Recurrent neural network regularization," *arXiv:1409.2329*, 2015.

[7] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," 2016.

[8] M. Dwarampudi and N. V. S. Reddy, "Effects of padding on lstms and cnns," 2019.

APPENDIX

Here, I report some of the parameters chosen for the main tests for finding the best hyperparameters. Each run was done with multiple accounts in Google Colab (free account limitations) to be completed. The size of the hidden, batch, and other parameters were pre-determined with smaller tests sopped earlier.

MODEL	BATCH	start LR	epoch LR	scale LR	Layer Norm	Element Affine	Dropout	Hidden Dim	Shffle data
GRU	128	0,01	14	1,13	Yes	No	0,65	1500	No
GRU	64	0,01	14	1,13	Yes	Yes	0,65	1500	No
LSTM	64	0,01	14	1,13	Yes	No	0,65	1500	No
LSTM	64	0,001	14	1,12	Yes	No	0,65	1500	Yes start 17
GRU	64	0,001	14	1,12	Yes	No	0,65	1500	Yes start 12
LSTM	64	0,001	14	1,12	No	No	0,65	1500	Yes
GRU	64	0,001	14	1,12	Yes	No	0,65	1500	Yes
LSTM	64	0,001	14	1,12	Yes	No	0,65	1500	Yes
LSTM	64	0,001	14	1,12	Yes	Yes	0,65	1500	Yes