Final NLU Project: Language Modeling with RNN

Olga Zaghen (224436)

University of Trento

`olga.zaghen@studenti.unitn.it`

1. Introduction & Task Formalisation

The goal of the assignment was to implement and evaluate a Neural Language Model through a Recurrent Neural Network of some kind. It is fairly ascertained that Neural Language Models outperform N-gram Language Models in their ability to capture long distance dependencies in word predictions, thanks to the presence of hidden states in RNN cells and not being limited by the upper bound N of N-gram's span.

Specifically, the task consisted in implementing a baseline RNN Language Model, consider its performance as a starting point and then try to improve its strength and results by fine-tuning it with any possible kind of techniques, such as regularization or optimization procedures.

2. Data Description & Analysis

In order to evaluate and compare different approaches, the Language Modeling task is performed over a preprocessed version of the Penn Treebank (PTB) dataset. It consists of three `.txt` files, corresponding to training, test and validation sets, containing 42068, 3761 and 3370 sentences respectively. The dimension of the training set is a fundamental value that must be taken into account for the definition of the capacity of the network.

The dataset is constructed from a corpus that has been heavily preprocessed: all the words it contains are already lowercased, there is no punctuation, and all the numbers have been replaced with the symbol `N`. Additionally, it already contains the special token `<unk>`: the large number of out of vocabulary (OOV) tokens is the result of the corpus' preprocessing, with the maximum vocabulary size being fixed to 10,000 unique words. Also, it can be noticed that all the words from the test and validation sentences are already present in the training set. This implies that, when defining a reference vocabulary with the words from the training set, there will be no OOV words in the test and validation sets.

The maximum length of the training sentences is 82. This low value guarantees that, when training an RNN on this dataset, simple Backpropagation Through Time can be adopted without too serious consequences for what concerns vanishing and exploding gradients. Also, none of the training sentences has length 0 (there are no empty lines in the file), while many of them have length 1. For what concerns the validation and test sets, the maximum lengths of their sentences are 74 and 77, respectively. It is interesting to also plot and observe the histograms that display the frequencies of the various sequence lengths in training, validation and test: besides following a symmetric "bell curve" shape, they clearly show that for all the three cases the most frequent sequence length coincides with a value between 15 and 20, and most of the sentences are characterized by a length value that is below 25.

The last statistical property of the dataset that is worth mentioning can be extracted through the analysis of the Frequency Dictionaries of the three sets: along with the most common

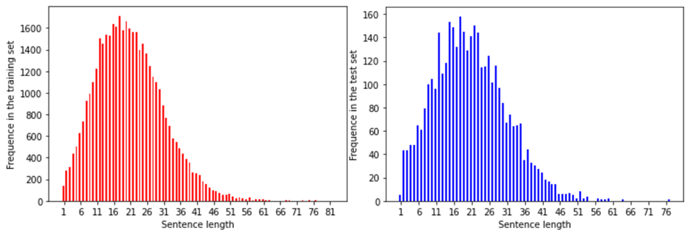


Figure 1: *Frequencies of lengths in training and test set*

prepositions of use, such as `the` and `of`, also the special tokens `N` and `<unk>` are part of the words that appear most frequently.

3. Models

In this section the structure of the implemented models is presented, as well as the general procedure adopted to perform experiments on the dataset and train the networks.

3.1. General-purpose pipeline

Since neural networks deal with numerical vectors, the first step consisted in constructing a vocabulary, `vocab`, composed by two dictionaries that map each word from the training set to a specific index and vice-versa. The vocabulary also contains the end-of-sentence token `</s>` and the `pad` token, which is equal to 0. These two are needed to construct the batches for the training and evaluation phases of the network. The end-of-sentence token is firstly added at the end of each sentence. Then, a `SentsDataset` is constructed mapping words to indexes. During the experiments, the `Dataloader` provides batches containing all the same fixed number of sentences. For what concerns the second dimension, it can vary from batch to batch and it corresponds to the length of the longest sentence in the batch itself: padding is performed through the `pad` token on the right of the sentences whose length is not the maximal one.

The pipeline followed in order to train and evaluate the models is quite standard. The weights of the architecture are firstly initialized to small random values, and this is a procedure that allows to stabilize the training phase. The neural networks are then trained by iterating for 200 epochs over the training set. Indeed, it is most likely that less epochs are performed in practice, since an *early stopping* technique is applied by monitoring the validation loss and stopping the training procedure when such loss is not subject to any improvement for a number of iterations equal to a fixed *patience* value.

In the context of the training phase setting, I fixed all hyper-parameters, such as the learning rates, to values that proved to perform well in practice, and I chose to adopt Adam optimizer for all experiments, because it has shown empirically to work better than other optimizers in my specific framework.

3.2. Baseline Models

In order to achieve a meaningful initial framework, I first implemented some N-gram baselines and considered them as starting point references. I trained and evaluated MLE, Laplace and Lidstone 2-gram and 3-gram Language Models, and I could verify that the results were not outstanding, and that in general bigram Language Models performed better with respect to trigram Language Models in this setting.

Test Perplexity values

	MLE	Laplace	Lidstone
2-gram LM	inf	918.96	918.96
3-gram LM	inf	3252.04	3252.04

After that, I started defining, training and evaluating two baseline Neural Language Models: a simple RNN architecture with a Vanilla RNN cell, and the same architecture with an LSTM cell in substitution of the Vanilla RNN cell.

They were composed of just the following simple modules:

- embedding layer
- vanilla RNN cell or LSTM cell
- linear decoder layer.

3.3. Improved LSTM

In order to improve the performance and results of the baselines, I decided to start from the LSTM model and introduce some regularization techniques to the training phase, as well as improve the optimization algorithm. Furthermore, I implemented and inserted a residual block with an attention module within the network's architecture: this additional layer will be described properly in the next section. In my case, stacking multiple LSTM layers as proposed in the paper *Regularizing and Optimizing LSTM Language Models* by Merity et al.9 [1], did not provide any substantial improvement in practice. For this reason I maintained just one hidden LSTM layer.

3.3.1. Optimization: Truncated BPTT

In this case, differently from the baseline setting, a variation of Truncated Backpropagation Through Time is implemented in order to reduce the problem of vanishing and exploding gradients, speed up training and achieve faster convergence.

At most one truncation is performed for each batch, specifically after the 25th position of the sentences' lengths. The first sub-batch is passed as input to the RNN model, and Backpropagation and weight update are performed. After that, the hidden state and the cell of the LSTM are detached from the computational graph (while their value is maintained unaltered), in order to avoid considering the first subset of timesteps for the computation of future gradients. The second sub-batch, composed by the remaining parts of the initial sentences, and that could have a lower batch size with respect to the initial one, is then passed as input to the RNN and the weight update is carried out.

If the maximum length of the sentences in the batch is less than 25, the truncation is not performed (it wouldn't be necessary) and a simple BPTT is applied to the batch. According to the frequency distribution of the sentences' lengths in the training set, most length values are lower than 25, and the most frequent one is around 20. For this reason, the specific choice of 25 for the truncation threshold allows to perform just one Backpropagation step for most sentences, while splitting and performing two BPTT steps for very few of them. This condition grants a more stable and fast training with respect to setting the truncation threshold to a randomly chosen value.

3.3.2. Regularization techniques

Most of the implemented regularization techniques were inspired and adapted from [1], and they are the listed below.

- A very simple but effective adjustment, which is fundamental for reducing the impact of exploding gradients during training, is represented by *gradient clipping*: a pre-determined gradient threshold of 0.25, as suggested on [1], is introduced.
- As well as adding *weight decay* regularization technique within the optimizer, I also used L_2 decay on the individual unit activations and on the difference in outputs of the LSTM at different timesteps. These two strategies are labeled as *Activation Regularization* (AR) and *Temporal Activation Regularization* (TAR) respectively. Their introduction was inspired by [1].
- *Embedding dropout* regularization technique was introduced in the embedding layer. In general, dropout avoids overfitting by adding some noise in the training phase. The value of 0.1 was suggested in [1].
- I applied *locked dropout* regularization technique to the inputs and outputs of the LSTM layer: it samples a binary dropout mask only once upon the first call and then it repeatedly uses that locked dropout mask for all repeated connections within the forward and backward pass. The probability values are set to 0.4 ([1]).
- I performed *weight tying* between the embedding matrix and decoder output matrix: this technique allows to substantially reduce the number of parameters of the model, easing the issue of overfitting and improving the results of the training procedure.

3.4. Improved LSTM with Attention

Although attention modules are mostly adopted in the context of Machine Translation, because of their ability of providing a good measure of alignment between the words in the input sentence and translated sentence, I tried implementing a sort of attention layer that could suit and be useful also for the considered Language Modeling task. I took inspiration, to some extent, from the general ideas underlying the structure of the *self-attention* module of the Transformer proposed in the *Attention Is All You Need* paper by Vaswani et al. [2].

Specifically, I introduced the attention layer through a residual block between the LSTM layer of the network and the linear decoder layer. Each LSTM's output timestep h_t goes through two different paths: in one case it is processed with the attention layer and a final *context vector* c_t is obtained from it, in the other case a simple identity function is applied. The result of these operations is the element-wise sum $h_t + c_t$, input of the final decoder layer.

As shown with their introduction in *ResNet* [3], skip connections implemented through residual blocks can improve the model's expressive capability because they help it better approximate a broader set of functions, and they allow the network to ignore ("skip") some layers when they are not useful or not necessary. Also, residual connections encourage feature reuse.

Each output timestep h_t of the LSTM layer goes through the following steps:

1. Two new tensors are defined: one is the repetition of h_t for $t - s + 1$ times, and the other consists in the last $t-s+1$ timesteps, $h_s, ..., h_t$, where $s = max(0, t-6)$.

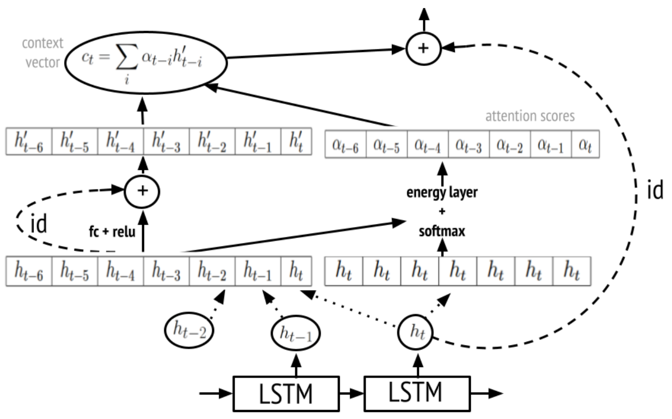

context vector
$c_t = \sum_i \alpha_{t-i} h'_{t-i}$
+
attention scores
h'_{t-6} h'_{t-5} h'_{t-4} h'_{t-3} h'_{t-2} h'_{t-1} h'_t
α_{t-6} α_{t-5} α_{t-4} α_{t-3} α_{t-2} α_{t-1} α_t
energy layer
+
softmax
id
fc + relu
id
h_{t-6} h_{t-5} h_{t-4} h_{t-3} h_{t-2} h_{t-1} h_t
h_t h_t h_t h_t h_t h_t h_t
h_{t-2}
h_{t-1}
h_t
LSTM
LSTM

Figure 2: *Computations performed through the attention module residual block for a specific timestep h_t output of the LSTM*

The latter tensor provides additional contextual information, that can be exploited by h_t for a more accurate prediction. I choose 7 as maximal length for these vectors because a very long context may be misleading (and lead to too heavy and time-consuming computations), while a too short one may not be informative enough. This value proved to perform well empirically.

2. The two aforementioned tensors are processed together in the *energy layer*, which performs a bilinear transformation. I implemented this particular type of operation because, with respect to linear transformations, it better captures the intrinsic correlation between the elements of the two inputs. The output of this layer is a vector of $t - s + 1$ scores: each one expresses the level of correlation between h_t and a particular h_i, $i \in \{s, ..., t\}$ (obviously, the one of h_t with itself will be higher than the others). The obtained values are then normalized through a *softmax layer*, and the *attention scores* are obtained: each one expresses the level of relative focus to put on each timestep $h_s, ..., h_t$ for a better prediction.

3. The vector $[h_s, ..., h_t]$ also goes through a residual block, in which a linear transformation and a ReLU activation are applied and new *values* are obtained for each timestep h_i, $i \in \{s, ..., t\}$. These operations aim at increasing the networks' expressive capabilities on the hidden states, in case it may be useful to achieve a more informative context vector.

4. The results of the previous step are then linearly combined, each one weighted with a specific *attention score*, to obtain the *context vector* c_t. It carries the information originally contained in h_t, enriched with the processed contextual information of the recent previous timesteps.

4. Evaluation

In this section the adopted evaluation metrics are explained, the results obtained by the different models are compared and error analysis is performed.

4.1. Evaluation metrics

The cost functions used to train and evaluate the model are the Cross Entropy loss and the Perplexity. Starting from the CE value, the Perplexity is obtained by simple exponentiation: $pp = exp(loss_{CE})$. The base of the exponent is taken as e since the `CrossEntropyLoss` provided by *Pytorch* performs log_e logarithms. In this setting it is defined by also requiring `ignore_index = PAD_TOKEN`, because the `pad` tokens in the target vectors must be ignored for a correct computation of the loss. Furthermore, the `reduction` parameter is set to `'sum'`: after adding up the CE costs of all the samples in the batch, the obtained raw sum is returned, and not the batch-wise mean. This allows to better handle the case in which batches contain a different number of elements: it is better to add up all the individual CE costs and then divide, in the end, with respect to the sum of the number of samples contained in all batches.

4.2. Results and models comparison

In order to evaluate the performance of the models, two main factors are compared: the trends of the training and validation losses across epochs during training, and the final Perplexity values achieved on the test set after training.

It is evident from the plots shown below that, during training, all models have a tendency to overfit after a certain number of epochs. This behaviour is particularly extreme for the baseline models, where the loss on the training set decreases steadily, while the generalization gap (the difference between training and validation errors) increases dramatically, although this gap is much more limited in the LSTM case. These issues are partially reduced with the introduction of the numerous regularization techniques and optimization tricks in the improved LSTM models. In these cases, both the training and the validation errors tend to converge to stable plateau values after some epochs, and the generalization gap between them is relatively smaller.

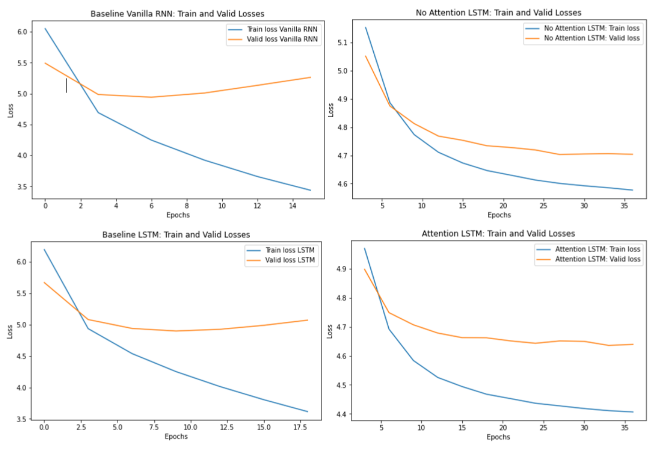


Figure 3: *Training and validation losses across epochs for baseline models (left column) and improved LSTMs (right column)*

The introduction of the attention module does not yield substantial improvements in the generalization capabilities of the model, since the gap between training and validation errors results to be slightly increased during training. I also tried introducing dropout layers in different positions of the attention block, but they did not bring satisfactory results. Nevertheless, it noticeably provides a meaningful reduction of the final Perplexity scores computed on both training and test sets.

To assess the results' reliability, I trained each model from scratch 5 times, I computed the Perplexity values on the test set after each run and I calculated the mean and standard deviation of the obtained results. The decrease in mean loss values for the models shows the improvements brought by the optimization and the regularization tricks, as well as the introduction of the Attention layer. The low standard deviation values are instead an indication of training stability.

Test Perplexity values across 5 runs

	Mean	Standard Deviation
Baseline Vanilla RNN	169.574	1.464
Baseline LSTM	138.444	0.856
Improved LSTM	105.735	0.281
Improved LSTM + Attention	100.331	0.538

4.3. Error analysis

As previously shown, the improved LSTM model with Attention is the one that grants the best results and performance. Nevertheless, the Perplexity value achieved by such model on the test set is still quite high. In order to understand the behaviour of the network when it deals with previously unseen sentences, a detailed error analysis is carried out by studying how the model deals with each sentence from the test set.

4.3.1. Special tokens and most frequent words

In order to evaluate the coherence of the predictions, we can start by verifying whether the model ever outputs the `pad` token. Indeed, even though the `pad` token is contained in the vocabulary and hence could be predicted, it should never be. I ascertained, by computing and analyzing the set of all predicted words, that the model behaves correctly in this sense.

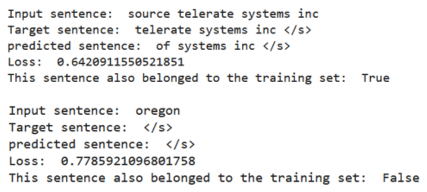
Furthermore, it results from the Frequency Dictionaries that the most frequently predicted words correspond also to the most frequent ones of the training and test sets, such as `<unk>`, `the`, `N`, `of`. This reflects the ability of the model to learn the statistical properties of training data, even though the frequency of tokens like `<unk>` and `the` is relatively higher in predictions: this probably happens because, when the model is ”confused”, it just outputs one of the most frequently observed words.

I tried testing whether the model had learnt how to use the end-of-sequence token `</s>` properly, that is, to mark the end of a meaningful sentence. In order to do this, I used the network as a generative model, starting from some plausible start-of-sentence tokens (such as `the`, `a`), and making it predict the successive word, given as input the previously predicted one. Disappointingly, the generated sentences show that when no proper and significant context is provided, the model tends to output some of the most frequently observed words during training, such as `<unk>`, `N`, `</s>`. It is also interesting to notice that, when given `</s>` as input, the model predicts `the`, which is a totally valid choice for the start of a new sentence.

4.3.2. Perplexity value on sentences

For the sake of understanding the reason behind higher and lower Perplexity values for different sentences, I carried out qualitative and quantitative studies on test sentences' properties and lengths, in correlation with their Perplexity scores.

Qualitative analysis. I extracted and printed the test sentences on which the model obtained the lowest and highest CE loss values (and hence Perplexity values). I also checked, for each sentence, whether it was also present in the training set, in the exact same form. Very interestingly, among the 30 sentences with the lowest loss, 18 also belonged to the training data, hence the model had already observed them. For what concerns the other 12, they were characterized by being very short or, if longer, very repetitive and with no underlying complex structure or meaning. With this kind of examples, the network is not required to put too much effort into capturing and exploiting complex contextual information.

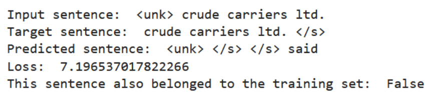


```
Input sentence:  source telerate systems inc
Target sentence:  telerate systems inc </s>
predicted sentence:  of systems inc </s>
Loss:  0.6420911550521851
This sentence also belonged to the training set:  True

Input sentence:  oregon
Target sentence:  </s>
predicted sentence:  </s>
Loss:  0.7785921096801758
This sentence also belonged to the training set:  False
```

Figure 4: *Example of test sentences with low Perplexity score*

Surprisingly, also the 30 sentences with highest loss values were quite short. Qualitatively, they often contain words that appear rarely, and they almost always consist in small cuts of longer periods, that are quite difficult to contextualize and make a sense of. For this reason, with no proper context, the model struggles to predict correctly and just outputs the guesses with the highest overall probability, such as `<unk>`, `</s>` and `the`.



```
Input sentence:  <unk> crude carriers ltd.
Target sentence:  crude carriers ltd. </s>
Predicted sentence:  <unk> </s> </s> said
Loss:  7.196537017822266
This sentence also belonged to the training set:  False
```

Figure 5: *Example of test sentence with high Perplexity score*

Quantitative analysis. In order to understand how Perplexity is correlated with the test sequences' length, I performed two types of analysis on the length-loss pairs. First of all I plotted how the typical sentence length varies with respect to the loss, for increasing loss values. Additionally, I computed and visualized the mean and standard deviation of the loss for each length value observed in the test data.

The results are the same in both cases and they show, interestingly, that while the mean loss value is almost the same for all lengths, the standard deviation tends to decrease as the length increases. This happens because longer sequences yield more substantial contextual info, and the performance of the model on them only depends on its ability to capture and exploit that info, hence its score is quite stable with these samples. For what concerns shorter sentences, instead, as observed during the qualitative analysis, some may be difficult to interpret because of the lack of proper context, while others can be so repetitive that guessing right is particularly easy; for this reason the model could perform poorly or brilliantly depending on the specific case, and the scores are subject to high variability.

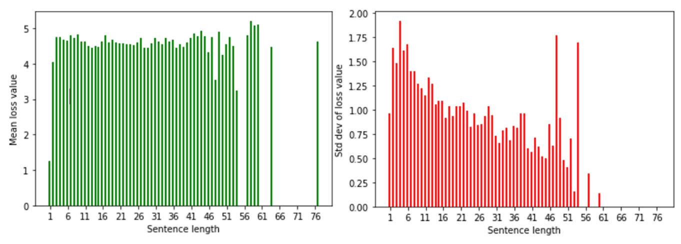


Figure 6: *Mean and standard deviation of test loss for every possible sentence length value*

4.3.3. Precision and recall

Ideally, the more frequently a word appeared during training, the more the model should be able to capture its influence and meaning in different contexts, and the higher should be the percentage of times it is predicted correctly at test time.

In order to verify this trend I plotted, for a certain set of words, the *recall* score, that is the percentage of times each of them was predicted correctly, with respect to the total number of appearances in the targets of the test set. In particular, I considered the 50 most frequent words from the training set, in descending order of frequency. The expected behaviour, as stated before, is that a higher *recall* in prediction should be associated to those words that appeared most frequently in the training set. This tendency is somehow validated by the plot displayed below, with the exception of some outliers characterized by high *recall* values even though their frequency in the training set is relatively low.

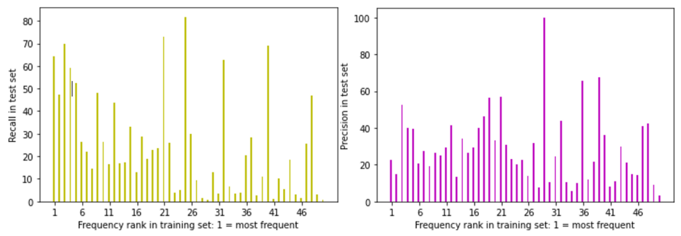


Figure 7: *Recall and precision of the model on the test sentences, computed for the 50 most frequent training words*

Another value I checked is the *precision* of the model on the test set, again for the 50 most frequent training words. The *precision* corresponds to the percentage of times a certain word was predicted correctly, with respect to the number of appearances in the predictions. In this case the plot does not show any meaningful trend.

5. Conclusion

Through the development of this project I had the chance to acknowledge that Neural Language Models can significantly outperform N-gram Language Models. Nevertheless, nothing comes from granted and the achievement of satisfactory results requires an accurate definition and enrichment of the architecture and of the regularization and optimization framework, as well as a big and well structured dataset.

6. References

[1] Merity S., Keskar N. S., and Socher R. (2017), “Regularizing and Optimizing LSTM Language Models”, *arXiv preprint arXiv:1708.02182*

[2] Vaswani A., Shazeer N., Parmar N., Uszkoreit J., Jones L., Gomez A. N., Kaiser L., and Polosukhin I. (2017), “Attention Is All You Need”, *arXiv preprint arXiv:1706.03762*

[3] He K., Zhang X., Ren S., Sun J. (2017), “Deep Residual Learning for Image Recognition”, *arXiv preprint arXiv:1512.03385*