Final NLU project - Language Modeling

Chiara Camilla Rambaldi Migliore (231689)

University of Trento

cc.rambaldimigliore@studenti.unitn.it

This document describes the techniques used to develop the project for the MS.C. course "Natural Language Understanding" at the University of Trento. The goal of this project was to implement a language model using one of the RNN architectures and to improve it using some regularization techniques.

1. Introduction

In this section I will briefly explain what Language Modeling is and what are the main techniques to compute it. Then I will introduce my approach to problem.

1.1. Language Modeling

Language Modeling is a technique that uses probability and statistics to determine the probability that a sequence of words can occur in a sentence. Hence, a Language Model is a model that assigns probability to sequences of words. There are several ways to compute a Language Model:

- N-grams [1]
- Neural Networks [2]
- Recurrent Neural Networks [3]
- Transformers [4]

1.2. My approach

My work focuses on one of the most used methods nowadays: the Recurrent Neural Network (RNN). RNNs are neural networks with a special architecture that allows you to exhibit temporal dynamic behavior. Moreover, RNNs can process variable length sequences of inputs, in our case words, maintaining an internal state (memory). This internal state is the key to "remember" a huge variety of sequences of words, thus computing a more precise probability of which word will appear.

2. Task Formalisation

In this section will be described the aim of the project and the methodology used to achieve it.

2.1. Goal

The goal of this project is to manipulate the given dataset so that it can be fed to a Neural Network and then implement a base RNN and improve it using a better architecture and applying some regularization techniques. Afterwards, these two networks will be analyzed and compared in terms of performance.

2.2. Methodology

From the dataset I will extract a dictionary of unique words, each linked to an index. Those indexes will be used to tokenize the dataset, ending up with a list of numbers (each linked to a precise word). This list of numbers will then be embedded into a list of feature vectors, ready to be fed to the actual RNN. The better architectures that can be used to improve the baseline are:

- LSTM (Long Short Term Memory)
- GRU (Gated Recurrent Unit)

The more famous and widely used regularization techniques, including some described in the Merity et al. paper [5], are: The widely used regularization techniques, some of which are described into Merity et al. [5] paper, are:

- Gradient clipping
- Variable length backpropagation sequences
- Variational dropout
- Embedding dropout
- Weight tying
- Independent embedding size and hidden size

When the implementation of the baseline and the improved network will be done, a common metric will be chosen (e.g. perplexity), in order to analyze the performance equally.

3. Data Description & Analysis

This section analyses the origin and the statistics of the dataset. The data used for this project are a preprocessed version of the Penn Treebank (PTB) data set [6]

3.1. Origin and composition

Penn Treebank dataset comes from the Penn Treebank Project (1989-1996). The project intention was to collect a corpus consisting of 7 million words of part-of-speech tagged text, 3 million words of skeletally parsed text, over 2 million words of text parsed for predicate argument structure, and 1.6 million words of transcribed spoken text annotated for speech disfluencies [7]. The corpus has been annotated with Part-Of-Speech information and for skeletal syntactic structure during the years. This dataset has been widely used by a lot of researchers to perform NLP tasks on a common dataset, despite the availability of a widely used standard dataset is quite rare. This allows a fair comparison of the NLP methods among the research groups.
The actually used dataset is a Penn Treebank portion of the Wall Street Journal corpus.

3.2. Statistics

The dataset is composed of 10000 distinct words. It is split into train, validation and test sets. The train set is composed of 42068 lines, with a total of 929589 tokens. The validation set is composed of 3370 lines, with a total of 73760 tokens. The test set is composed of 3761 lines, with a total of 82430 tokens. The vocabulary length is kept to 10000 by a heavy preprocessing computation which also acts on the removal of capital letters, numbers and punctuation. This avoids the use of too much

computation time and complexity.
All the words in the vocabulary are present either in train, valid and test sets. *Out of vocabulary* words are avoided due to the preprocessing process that substituted a lot of words with the *"*$< unk >$*"* token.

4. Model

This section will provide information about what model has been used and the main points of the developed pipeline.

4.1. Model chosen

The model chosen is a Long Short-Term Memory (LSTM) Neural Network (NN), a particular type of Recurrent Neural Network. An RNN is a NN that has a feedback connection which gives a sort of memory to the model. The LSTM solves a very big problem of the Vanilla RNN: the vanishing gradient. However, it still has the problem of the explosion gradient which can be avoided by a regularization technique called *clipping gradient* is used. Some other regularization techniques, such as *dropout* and *weight tying* are used to improve the model results. The baseline used is the Vanilla RNN.

4.2. Pipeline

4.2.1. Read Text

As a first step, the lines of text are read from the files.

4.2.2. Create Vocabulary

For each line in the train set, a **vocabulary** is created with all the words seen so far. While creating the vocabulary, a unique index is assigned to each word.

4.2.3. Tokenize Data

Using the unique index given to each word, all the dataset are **tokenized** and mapped to those indexes.

4.2.4. Create Dataset

In order to better work with Pytorch dataloader, a custom Dataset class is developed.
The core of the custom class is the **__getitem__(self, index)** method [8]. In fact, this is the method called by the **Dataloader** when dividing the items in batches. An item in the custom Dataset, is a vector of length *sequence_length*.

4.2.5. Create Dataloader

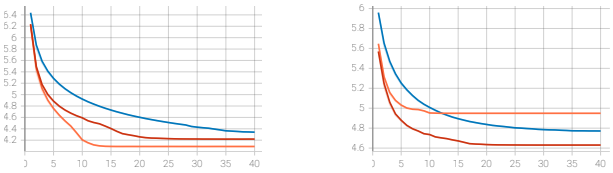
For each set, a Dataset object is created to be passed to a Dataloader.
The Dataloader creates the batches used during the training and test steps.

4.2.6. Word Embedding

Having the datasets all tokenized is not enough, in fact an embedding from word indexes to word vectors is needed. Luckily, Pytorch provides an embedding layer ready to be used for the purpose.

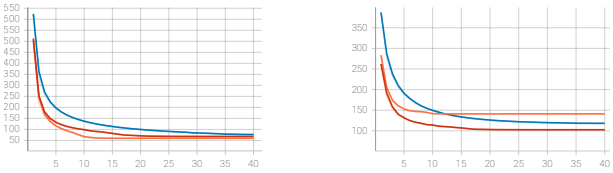
4.2.7. Modeling

Given all these ingredients, the model is then trained.
For the purpose of this project, a guideline has been taken from the Pytorch official examples [9]



(a) *Training Loss* (b) *Validation Loss*

Figure 1: *Training and Validation losses. Orange is for baseline, blue for intermediate and red for advanced*



(a) *Training Perplexity* (b) *Validation Perplexity*

Figure 2: *Training and Validation perplexities. Orange is for baseline, blue for intermediate and red for advanced*

4.2.8. Evaluation

At the end of each epoch during the training, an evaluation on the valid set is done.
In case of improvements, the weights of the model are saved as the *best model*, otherwise the learning rate is annealed. At the end of the training, the evaluation is performed on the test set, using the best model weights found during the training.

5. Evaluation

This section will provide a brief description of the main metrics used. It will then analyze the results achieved and compare the baseline with the actual model.

5.1. Metrics

The metrics used for the evaluation of the results are:

- Perplexity
- Loss

The perplexity (sometimes called PP for short) of a language model on a test set is the inverse probability of the test set, normalized by the number of words.
For a test set $W = w_1 w_2 ... w_N$, [1]:

$$\begin{aligned} PP(W) &= P(w_1 w_2 ... w_N)^{-\frac{1}{N}} \\ &= \sqrt[N]{\frac{1}{P(w_1 w_2 ... w_N)}} \end{aligned}$$

As loss has been used the Cross Entropy Loss, computed on every batch, summed up and divided by the number of batches. This loss allowed us to compute also the perplexity of the model, using a simple formula:

$$\begin{aligned} PP(w_{1:n}) &= 2^{H(w_{1:n})} \\ &= 2^{-\frac{1}{n}\sum_1^n \log_2 m(w_n)} \end{aligned}$$

where the value in the exponent is the *cross-entropy* of our current model with respect to the true distribution [1]. Since in **Pytorch** the *cross-entropy* is computed with a logarithm in base e and not in base 2, the perplexity in the code is computed as:

$$PP'(w_{1:n}) = e^{H(w_{1:n})}$$

5.2. Results

The model has been evaluated with three different configurations:

- Baseline
- Intermediate
- Advanced

The ***baseline*** is a Vanilla RNN without dropout; it uses a hyperbolic tangent as non linearity. The ***intermediate*** is an LSTM with *dropout* and *weight tying*. Finally, the ***advanced*** is actually the intermediate with in addition the *gradient clipping technique*. As shown in ***Fig. 1*** and ***Fig. 2***, both the *intermediate* and *advanced* configurations, outperform the *baseline*. In fact, the results on the test set are:

- ***Baseline*** - Loss 4.89, PP 133.26
- ***Intermediate*** - Loss 4.69, PP 109.25
- ***Advanced*** - Loss 4.58, PP 97.67

Moreover, the *intermediate* and *advanced* configurations have a similar trend. While the *baseline* tends to converge quickly to a high perplexity, the others converge more slowly but longer in the epochs. The graphs also show an interesting view on the differences between the training and validation trends during the epochs. The training loss and perplexity of the *baseline* are eventually better than the others, while the results on validation are definitely worse. Therefore, the *baseline* shows an overfitting problem, better handled by the *intermediate* and *advanced* configurations as they implement the dropout regularization technique. In fact, trying to run the advanced configuration without the dropout, it gains ≈ 14 points of perplexity At last, the only addition of the gradient clipping technique significantly improves the perplexity on the validation and test set as we can see comparing the trend of the *intermediate* and the *advanced* configurations.

5.3. Errors

The pipeline used for this project has been developed in such a way that some regularization techniques can't be applied. In particular, the *variable length backpropagation sequences* one. In fact, as described in the sub section 4.2, in this project the data is loaded with a **Dataloader** that needs a dataset object as input. This dataset object, can't change for each batch the size of the *sequence length*, thus removing the possibility of having a variable *BPTT* sequence for each batch.
A further improvement for this project can therefore be to change the way the data is loaded during the training step.

6. Conclusion

In conclusion, the overall results are good. Despite the simplicity of the models and the regularization techniques used, the aim of the project has been achieved. Moreover, the evaluations done on three different models, allowed me to better understand how much the models differ and how the *clipping gradient* technique improves the results. However, it could be interesting, as a further improvement, to develop a new way to load data, as mentioned in the sub section 5.3. And to implement more difficult regularization techniques described in the *Merity et al.* paper [5].

7. References

[1] D. Jurafsky and J. H. Martin, "N-gram language models," in *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 3rd ed., USA, 2022, ch. 3, pp. 30–56. [Online]. Available: https://web.stanford.edu/~jurafsky/slp3/ed3book_jan122022.pdf

[2] Y. Bengio, R. Ducharme, and P. Vincent, "A neural probabilistic language model," in *Advances in Neural Information Processing Systems*, T. Leen, T. Dietterich, and V. Tresp, Eds., vol. 13. MIT Press, 2000. [Online]. Available: https://proceedings.neurips.cc/paper/2000/file/728f206c2a01bf572b5940d7d9a8fa4c-Paper.pdf

[3] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990. [Online]. Available: https://www.sciencedirect.com/science/article/pii/036402139090002E

[4] C. Wang, M. Li, and A. J. Smola, "Language models with transformers," *arXiv preprint arXiv:1904.09408*, 2019. [Online]. Available: https://arxiv.org/pdf/1904.09408.pdf

[5] S. Merity, N. S. Keskar, and R. Socher, "Regularizing and optimizing lstm language models," *arXiv preprint arXiv:1708.02182*, 2017. [Online]. Available: https://arxiv.org/pdf/1708.02182.pdf

[6] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, "Building a large annotated corpus of english: The penn treebank," *Comput. Linguist.*, vol. 19, no. 2, p. 313–330, jun 1993. [Online]. Available: https://catalog.ldc.upenn.edu/docs/LDC95T7/cl93.html

[7] A. Taylor, M. Marcus, and B. Santorini, *The Penn Treebank: An Overview*. Dordrecht: Springer Netherlands, 2003, ch. 1, pp. 5–22. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.9.8216&rep=rep1&type=pdf

[8] Pytorch, "Writing custom datasets, dataloaders and transforms — pytorch tutorials 1.12.0+cu102 documentation," 2022. [Online]. Available: https://pytorch.org/tutorials/beginner/data_loading_tutorial.html#dataset-class

[9] ——, "Word language model," 2022. [Online]. Available: https://github.com/pytorch/examples/tree/main/word_language_model