LSTM implementation for language modeling

Alessio Dellsega 223962
University of Trento
Via Sommarive, 9, 38123 Povo, Trento TN
alessio.dellasega@studenti.unitn.it

I. Introduction

PREDICTING the word that comes after a given part of a sentence is one of the basic tasks in NLP and is the foundation for more complex tasks like speech recognition or machine translation. This work aims to implement a simple LSTM model in order to have a more precise idea on the fundamentals on which other, more recent and complicated models (like BERT), works and how a basic model like LSTM can solve a Language Modeling task. This project uses the Penn Treebank dataset in order to test the network.

II. Problem Statement

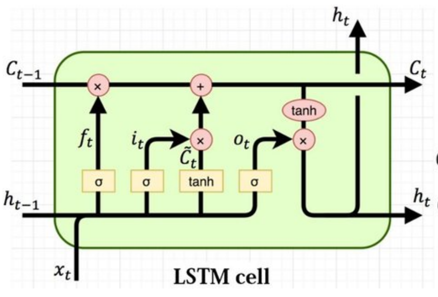
Given a sequence of tokens $< t_1, ..., t_n >$, we need to predict $t_{(n+1)}$ in order to have a meaningful sentence while preserving the grammatical integrity of the sentence (as much as possible). Then the prediction is compared with the actual target token in order to evaluate the prediction of the model.

III. Dataset

For this project I used the Penn Treebank dataset [1] for word-level predictions. The whole dataset is divided into three main corpuses, one for each ”phase” of the ML procedure (training, validation and testing). The whole dataset contains 888000 words for training, 70000 for validation, and 79000 for testing, with a vocabulary size of 10000. In order to define the vocabulary I used the training set because it contained the highest number of unique words (10000), so I used that vocabulary as a reference for the whole word-to-index conversion for the three sets. In addition to the Penn Treebank dataset I created a very small dataset (100 sentences) composed by sequences of *a,b,EOS* in order to test my basic implementation of a LSTM using only numpy for character-level prediction. I used a dataset this small due to the fact that numpy works on CPU, so it's extremely slow on a dataset of the same size as PTB.

IV. Model Implementation

The model used in this project is a standard model of Long-Short Term Memory Neural Network (i.e. LSTM) that uses standard gradient descent during training to learn weights and biases. The principle behind an LSTM network is quite simple: in order to solve the vanishing gradient problem of RNNs, we need a sort of long-term memory. LSTM aims to add this component to the whole architecture by saving a **cell state** in addition to the usual **hidden state** used also in vanilla RNNs. In order to compute both hidden state and cell state, is used a system of **gates** in order to select what the cell should remember and what should not. In the following image are represented the equations that are used inside a cell: In this project for both implementations, is used



$$i_t = \sigma\big(x_t U^i + h_{t-1} W^i\big)$$
$$f_t = \sigma\big(x_t U^f + h_{t-1} W^f\big)$$
$$o_t = \sigma\big(x_t U^o + h_{t-1} W^o\big)$$
$$\tilde{C}_t = \tanh\big(x_t U^g + h_{t-1} W^g\big)$$
$$C_t = \sigma\big(f_t * C_{t-1} + i_t * \tilde{C}_t\big)$$
$$h_t = \tanh(C_t) * o_t$$

Fig. 1. LSTM cell

a standard LSTM as shown in the figure above. Regarding the implementation using Numpy is a really basic LSTM that uses standard BPTT (taken from [3]), in order to be trained and initialize the weights orthogonally as stated in [2]. The main implementation of this project is the network implemented using PyTorch nn module: it is composed by an Embedding layer, a LSTM with 2 layers, a

Dropout layer and a Linear layer. I decided to use the Embedding layer instead of an external fine-tuned embedding (like Word2Vec) due to the easier usage in combination with PyTorch library and the internal optimization of the procedure. After trying the *Dataset* and the *DataLoader* classes offered by PyTorch, I decided to use a custom iterator that takes the whole dataset and divides it in batches of a given size dividing targets and inputs [4] for the acquisition and division of the sets. As criterion for calculating the loss during the experiments I used **torch.nn.CrossEntropyLoss()**. As for the optimizer, after trying a couple of optimizers from **torch.nn** module; I decided to use *AdamW* optimizer (Adam with weight decay) in order to avoid overfitting thanks to the weight decay.

V. Performances

The performances on the PTB of a LSTM for language modelling are around 78.93 of testing perplexity [5] under the *LSTM* category. In general my model is less performing than the state of the art but still reaches a fair perplexity score, and the general performances of a few experiments are indicated in the table below:

Test Perplex	Train Epochs	Hidden Size	Embedding Size
121.024	10	650	650
135.280	17	600	200
132.532	11	800	650

References

[1] https://deepai.org/dataset/penn-treebank

[2] Andrew M. Saxe, James L. McClelland and Surya Ganguli. *Exact solutions to the nonlinear dynamics of learning in deep linear neural networks*. Stanford University 2014.

[3] https://cs231n.github.io/neural-networks-case-study/grad

[4] https://github.com/deeplearningathome/pytorch-language-model/blob/5a0f888560ec6adfb366080f8f874f18b06caf14/reader.py

[5] https://paperswithcode.com/sota/language-modelling-on-penn-treebank-word