Word-level LSTM text generator

Zihadul Azam
Artificial Intelligence Systems (DISI)
zihadul.azam@studenti.unitn.it
Student Id: 221747
Acad. year: 2020/21

***Abstract*—Long short-term memory (LSTM) units on sequence-based models are being used in translation, question-answering systems, classification tasks due to their capability of learning long-term dependencies. Text generation models and the application of LSTM models are recently popular due to their impressive results. LSTM models applied to natural languages are great in learning grammatically stable syntaxes. In this paper, we have proposed a LSTM model which is capable to generate text starting from a sequence of input words. We have used the Penn Treebank Dataset to train our model. Then we have tried to set different hyperparameter values such the model can produce texts with an acceptable perplexity.**

1. Introduction

In the past few years, **Natural language processing** (NLP) has seen rapid developments and the reasons are mainly attributed to the developments in learning-based algorithms and advancements in computational power. As the availability of computational resources increased, active research and development happened in the architecture of deep learning algorithms. The use of feed-forward neural networks in lots of applications yielded impressive results. In order to stretch the limits of artificial intelligence and Deep Learning, we have developed a text generator, an automatic text generator that can generate a sequence of words automatically, after being trained. The primary objective of the text generator is to understand the sequence of words and write a new text, similar to human thinking. We have used the Penn Treebank Dataset to train the model.

In this paper, first we will see a little bit about Language Modeling. Then we will try to stretch **Recurrent Neural Network (RNN)** and **Long short-term memory (LSTM)**. After explaining the theoretical part, we will focus on Problem Statement, Dataset and Model. In the end, we will conclude this paper by highlighting the obtained results.

1.1 Language Modeling:

Language modeling (LM) is the use of various statistical and probabilistic techniques to determine the probability of a given sequence of words occurring in a sentence. Language models analyze bodies of text data to provide a basis for their word predictions. Language models determine word probability by analyzing text data. They interpret this data by feeding it through an algorithm that establishes rules for context in natural language. Then, the model applies these rules in language tasks to accurately predict or produce new sentences. The model essentially learns the features and characteristics of basic language and uses those features to understand new phrases.

There are several different probabilistic approaches to modelling language, which vary depending on the purpose of the language model. From a technical perspective, the various types differ by the amount of text data they analyze and the math they use to analyze it. Some common statistical language modelling types are:

- N-gram
- Unigram
- Bidirectional
- Exponential
- Continuous space

The models listed above vary significantly in complexity. Broadly speaking, more complex language models are better at NLP tasks, because language itself is extremely complex and always evolving. Therefore, an exponential model or continuous space model might be better than an n-gram for NLP tasks, because they are designed to account for ambiguity and variation in language.

A good language model should also be able to process long-term dependencies, handling words that may derive their meaning from other words that occur in far-away, disparate parts of the text. An LM should be able to understand when a word is referencing another word from a long distance, as opposed to always relying on proximal words within a certain fixed history.

In this paper, we will focus on N-gram. **N-grams** are a relatively simple approach to language models. They create a probability distribution for a sequence of n, then n can be any number and defines the size of the ”gram”, or sequence of words being assigned a probability. For example, if $n = 5$, a gram might look like this: ”can you please call me.” The model then assigns probabilities using sequences of n size. Basically, n can be thought of as the amount of context the model is told to consider. Some types of n-grams are unigrams, bigrams, trigrams and so on. N-gram models can outperform neural network models on small datasets. Estimating 2-gram probabilities formula can be defined as:

$$P(w_n|w_{n-1}) = \frac{P(w_{n-1}, w_n)}{P(w_{n-1})}, \quad (1)$$

Where:

w_n : is a token
w_{n-1} : is the preceding token

As we mentioned before, N-gram is not enough to have a robust text generator. A special type of RNN called LSTM is a good solution to handle longer word history. [1]

1.2 RNN and LSTM:

RNNs are neural networks that are good with sequential data. It can be video, audio, text, stock market time series or even a single image cut into a sequence of its parts. Standard neural networks (convolutional or vanilla) have one major shortcoming when compared to RNNs - they cannot reason about previous inputs to inform later ones, so they are not suitable to generate texts. But RNN is, it maintains in memory some information about past inputs and use them to predict the next word. Specifically, we will use LSTM, a special type of RNN, which are equipped to handle very large sequences of data. Simple RNNs have a problem called the vanishing gradient problem, because of which they cannot handle large sequences. LSTMs are designed to handle long-term dependencies. [2]

Now we will try to understand **the vanishing gradient problem** and how LSTM solves this problem.

As we mentioned before, RNNs suffer from the problem of vanishing gradients, which hampers learning of long data sequences. The gradients carry information used in the RNN parameter update and when the gradient becomes smaller and smaller, the parameter updates become insignificant which means no real learning is done. Here is the RNN architecture:

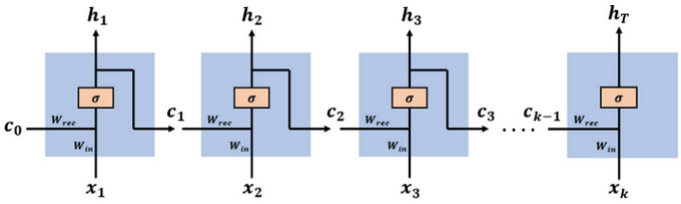

Fig. 1: RNN network

When an input *x* is passed into the RNN cell it will be multiplied with a weight *W* and a bias *b* will be added. Then the calculated value will be non-linearized by an activation function like sigmoid, ReLu etc. If we consider sigmoid function, for a range of input values, it will return an output within the range of *(0,1)*. If we consider a time-step of 4 and the output of sigmoid being 0.3. At the end of 4th time step the output *y* of the RNN will be *Y=0.3*0.3*0.3*0.3*x + b*.

x is being multiplied with a very small number which can be approximated to zero, which will result in zero gradient during the back-propagation, as a result, the weight won't be optimized. This issue is known as the vanishing gradient problem.

This will result in, initial layers not being optimized/trained. Since the latter layers depends on the low level features captured by the initial layer, overall performance of the model will be degraded. If the activation function is **ReLu**, then in case of negative input it will output zero, which then causes the same issue, even if we use Leaky-ReLu, the outputs for negative values are too small to overcome the vanishing gradient problem. So changing activation functions was not enough to deal with vanishing gradient problem.

The main reason for using RNN is to retain the information gained from previous data, but if we are facing vanishing gradient issues the amount of data that can be stored in the memory is very much limited.

LSTM is introduced as a solution to this problem. Long Short Term Memory is capable of storing processed information about the longer sequence of data. How does LSTM solve the vanishing gradient problem?

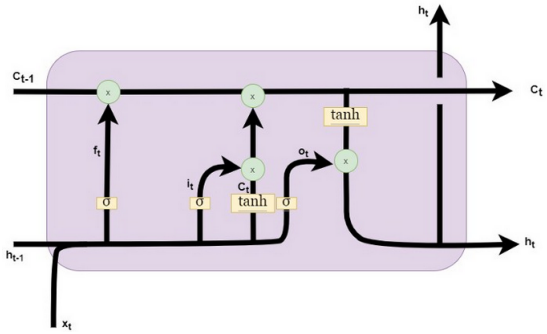

Fig. 2: Structure of LSTM

In the above diagram, the horizontal line noted by *Ct* is known as cell state, the memory of LSTM. It carries the information along all the cells, each cell based on the input it gets, modifies the information in the cell state. Gates are used to modify the cell state. Gates are the structure that decides whether the cell state to be modified or not. For example, if the gate generates *0*, then no new data will be added to the cell state, and if the gate generates *1*, then the full data will be added to the cell state. Altering the data in the cell state is simply multiplying the incoming data with the output of the gate and adding it to the cell state.

There are 3 types of gates in a LSTM cell, **forget gate**, **input gate** and **output gate**. Each one of them will decide what portion of the older data have to be forgotten, what

portion of newer data have to be remembered and what portion of the memory have to be given out correspondingly.

Here the gates regulate how much data get into next time-step, similarly, the same gates regulate how weights are to be optimized. If the gradient is to be vanished during the back propagation, the same thing will happen in the forward propagation as well. Hence that weight will not influence the prediction, therefore it will not be optimized during the back propagation. [3]

2. Problem statement

The Basic Problem Statement here is to Implement an RNN Language Model using one of the most famous architectures i.e., Vanilla or LSTM (in our case, we choose LSTM). Training the model with **Penn Tree Bank (PTB)** dataset and also experimenting with different hyperparameters to compare the perplexity of the models. In the end, use the pre-trained model to generate a new text given as input a sequence of starting words.

3. Dataset

To train our model we used the Penn Treebank Dataset. The **English Penn Treebank (PTB)** corpus, and in particular the section of the corpus corresponding to the articles of Wall Street Journal (WSJ), is one of the most known and used corpus in NLU. In the most common split of this corpus, sections from 0 to 18 are used for training (38 219 sentences, 929589 tokens), sections from 19 to 21 are used for validation (5 527 sentences, 131 768 tokens), and sections from 22 to 24 are used for testing (5 462 sentences, 129 654 tokens). The corpus is commonly used for character-level and word-level Language Modelling.

4. Data preprocessing

Our training corpus in total contains **38219 sentences**. First, we have to read them from the input file and split them into words, this process is known ad tokenization. After splitting we get a list of **929589 words**, from which we extract all the unique words and create our vocabulary. A vocabulary is nothing else than a list that contains all the unique words. In the PTB dataset, we have found **10000 unique words**. Once we have extracted all the vocabulary words, we associate an unique ID to each of them using an encoder. Then using these unique IDs, we transform our whole training corpus into a list of *integer* (substitute the word with its unique ID, using the **word2int encoder**). We do the same also for the Testing corpus and the Validation corpus.

Word list:
['i', 'love', 'my', 'cat', '.', 'i', ...]

Word ID list:
[48, 700, 189, 876, 8, 48, ...]

Now we are ready to transform our tokenized corpus into a list of sequences. As mentioned before, LSTM model as input takes a sequence of words and predict the next word of that sequence. Then at the end of the data pre-processing, we have divided all sequences into batches. So that we can feed them to our model. We also implemented the **decoder (int2word)**, so later we can use it to convert our output id to word.

5. Model

First, we have added an **embedding layer**, so we can avoid One-hot encoding. One-hot encoded vectors are high-dimensional and sparse. Let's assume that we are doing Natural Language Processing (NLP) and have a dictionary of 2000 words. This means that, when using one-hot encoding, each word will be represented by a vector containing 2000 integers. And 1999 of these integers are zeros. In a big dataset like ours, this approach is not computationally efficient. Word embedding provides us a dense representation of words.

Since our vocabulary has 10k words, so the shape of embedding layers will be **[10000 x embedding_size]**, where *embedding_size* is a hyperparameter. In this project, we have tried to fix *embedding_size* equal to 650 and 1500, among them 1500 has produced a better result in term of perplexity. Then we have added a LSTM layer with *num_layers* equal to 2. *num_layers* is the number of recurrent layers. E.g., setting *num_layers=2* would mean stacking two LSTMs together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM and computing the final results. After the LSTM layer dropout has been added. This has proven that dropout is an effective technique for regularization and preventing the co-adaptation of neurons as described in the paper: [4]. For the final layer, we have used a fully connected layer with shape **[batch_size, number of vocab]**. So in the prediction phase, we have output with the shape of **[batch_size x bptt, 10001]**.
Here is the model architecture:

```
==========================================================================================
| Model structure
==========================================================================================
LSTMModel(
  (emb_layer): Embedding(10000, 650)
  (lstm): LSTM(650, 650, num_layers=2, dropout=0.65)
  (dropout): Dropout(p=0.65, inplace=False)
  (fc_layer): Linear(in_features=650, out_features=10000, bias=True)
)
==========================================================================================
```

Fig. 3: Model structure

During the training and evaluation phase cross entropy criterion and perplexity have been used. Cross entropy loss is commonly used in classification tasks both in traditional ML and deep learning. The cross entropy criterion combines *LogSoftmax* and *NLLLoss* in one single class.

For a model m modeling a reality, and the real probability distribution p of that reality, say the probability of a sentence following a sequence of sentences, then the cross entropy is defined as The loss can be described as:

$$H(p, m) = -\sum_{i=1}^{n} p(x_i)\, log_2\, m(x_i) \tag{2}$$

Where:

$$n \text{ : is the number of possible states}$$

Perplexity is a metric used essentially for language models. It is defined as the exponential of the model's cross entropy:

$$Perplexity = 2^{H(p,m)} \tag{3}$$

During the training phase as optimizer, we have used **Stochastic Gradient Descent (SGD)**. SGD is an iterative method for optimizing an objective function with suitable smoothness properties (e.g. differentiable or subdifferentiable). It can be regarded as a stochastic approximation of gradient descent optimization since it replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data). Especially in high-dimensional optimization problems, this reduces the computational burden, achieving faster iterations in trade for a lower convergence rate. We have also used the **annealing** technique, anneal the learning rate if no improvement has been seen in the validation dataset.

6. Experiments and Results

He have tained and tested our model with different settings. We have tried to figure out the best parameter values, the parameters are: embedding size, hidden size, bptt and dropout. Here are the results:

Embedding size	Hidden size	Bptt	Dropout	Test perplexity
650	650	35	0.5	82.77
650	650	60	0.5	84.61
1500	1500	35	0.65	79.25
1500	1500	60	0.65	82.64

TABLE I: Results

Here we can see that the **test perplexity** is best with the third settings, which is: **Embedding Size = 1500, Hidden size = 1500, Bptt = 35, Dropout = 0.65** . With this setup we have achieved test perplexity equal to **79.25**

Here are some plots which highlight the loss and perplexity per number of epochs during the training and validation phases:

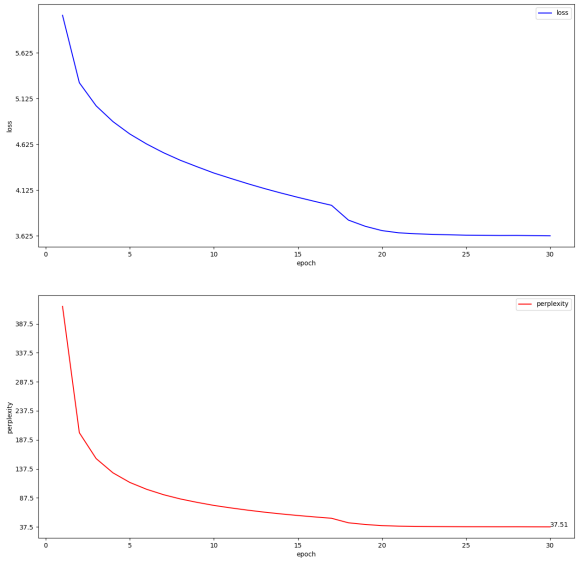

Fig. 4: Training loss and perplexity

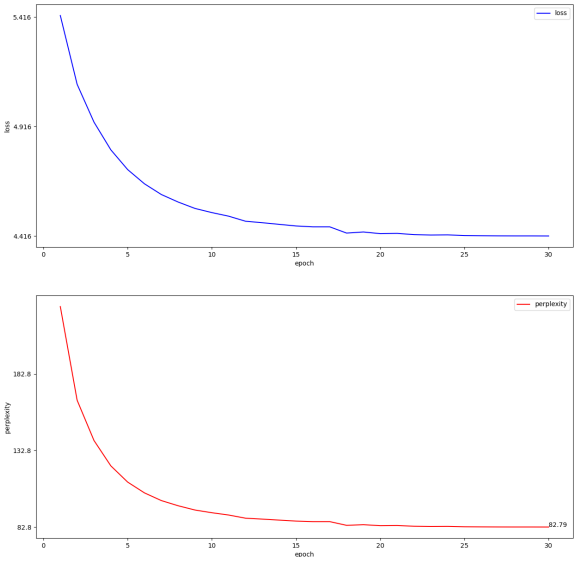

Fig. 5: Validation loss and perplexity

References

[1] Saurav Chakravorty, "Language Models", https://towardsdatascience.com/language-models-1a08779b8e12.

[2] Colah's blog, "Understanding LSTM Networks", https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[3] Nir Arbel, "How LSTM networks solve the problem of vanishing gradients", https://medium.datadriveninvestor.com/how-do-lstm-networks-solve-the-problem-of-vanishing-gradients-a6784971a577.

[4] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, Ruslan R. Salakhutdinov - "Improving neural networks by preventing co-adaptation of feature detectors", https://arxiv.org/abs/1207.0580.

[5] Mikolov, Burget, Kombrink - "Extensions of recurrent neural network language model", https://www.researchgate.net/publication/224246503_Extensions_of_recurrent_neural_network_language_model.