Language Model with LSTM architecture

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 report, I will explain about the implementation of a RNN language model using LSTM architecture. I have used the Penn Treebank Dataset to train and test the model. Then I have tried to set different hyperparameter values such the model can produce results 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, I have developed a RNN language model with LSTM architecture, which can learn language features and predict next sequence of words automatically, after being trained. The primary objective of this model is to understand the sequence of words and predict a new sequence, similar to human thinking. I have used the Penn Treebank Dataset to train the model.

In this report, first I will explain little bit about Language Modeling. Then I will try to stretch **Recurrent Neural Network (RNN)** and **Long short-term memory (LSTM)**. After explaining the theoretical part, I will focus on Problem Statement, Dataset and Model implementation. At the end, I will conclude this report 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 and predict 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 report, 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. Here is the formula of 2-gram probabilities:

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

Where:

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

As mentioned above, N-gram is not enough to have a robust text predictor, it struggles to handle long dependencies. A special type of RNN called LSTM is a good solution to handle longer word history.

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, in this report we will see LSTM, a special type of RNN, which is 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.

Now let's try to understand briefly **the vanishing gradient problem** and how LSTM solves this problem.

As mentioned above, RNNs suffer from the problem of vanishing gradients, which can create problems during the 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. So even though we pass new input data, the network will not learn more.

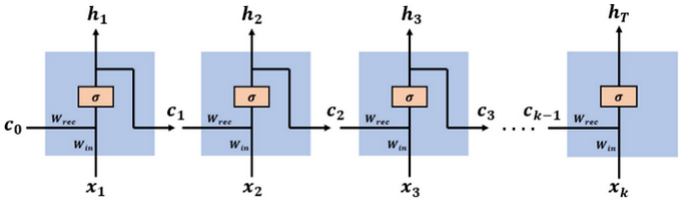

Fig. 1: RNN network

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** (produce outputs within the range of(-1,1)), the outputs for negative values are too small to overcome the vanishing gradient problem. So changing activation functions is 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. So it is necessary to find an alternative or impovement of RNN.

Long Short Term Memory (LSTM) is a good solution to this problem. It is capable of storing processed information about the longer sequence of data. With it's input gate and forget gate mechanism it can infer about which information should be retained and which be forgotten.

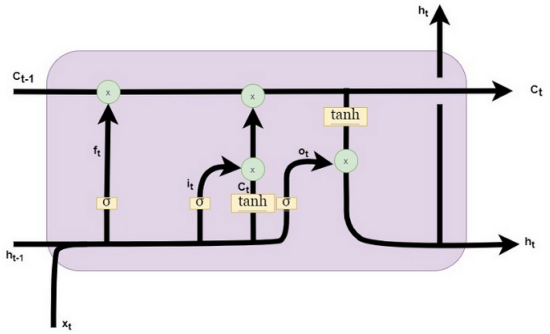

Fig. 2: Structure of LSTM

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.

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 my case, I have chosen the LSTM architecture, then to implement it I have used **Tensorflow Keras** python library. Once the model is implemented, I have trained the model with **Penn Tree Bank (PTB)** dataset and I have also experimented with different hyperparameters to compare the perplexity of the models.

3. Dataset

To train the model I have 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. According to the official documentation, 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. The dataset contains also the <**unk**> tag, which is used for rare words such as uncommon proper noun. Without <unk> tag, the dataset contains in total **9998 unique words**.

4. Data preprocessing

The training corpus of PTB in total contains **38219 sentences**. First, I have read them from the input file and have split them into words, this process is known as tokenization. During the tokenization, I have used <**eos**> tag (end of sentence) to mark the end of the sentence. After splitting I have gotten a list of **929589 words**, from which I have extracted all the unique words and have created the vocabulary. A vocabulary is nothing else than a list that contains all the unique words. In the PTB dataset, I have found **10000 unique words** (9998 unique words + <unk> tag + <eos> tag). Once I have extracted all the vocabulary words, I have associated an unique ID to each of them using an encoder. Then using these unique IDs, I have transformed the whole training corpus into a list of *integer* (substitute the word with its unique ID, using the **word2int encoder**). This was necessary because the neural network and the machine can process numbers only, cannot understand physical words. Then I have done 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, ...]

During a supervised learning process along with the training data, we have to also provide the target data to the network, so after the feedforward process, it can compare its prediction with the real target data and calculate the loss, then using this loss can optimize its parameters. In our case the target data is nothing else than the input data just left shifted by one position (for each word, the target word is its next word). Here is an example:

Input:
[48, 700, 189, 876, 8, 48, ...]

Target:
[700, 189, 876, 8, 48, 999, ...]

Finally, after all these pre-processing phases now we are ready to transform our data into lists of sequences. As mentioned before, LSTM model as input takes a sequence of words and predict the next words of that sequence. Then at the end of the data pre-processing, I have divided all sequences into batches, so we can feed our network one batch at a time.

5. Model

To create the LSTM model I've used **Tensorflow Keras** library. Tensorflow Keras is a powerful library to implement deep neural networks. My network is composed with different layers. First, I have added an **embedding layer** to 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 my 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, I have tried to fix *embedding_size* equal to 350, 500 and 650.

Then I have added a dropout layer with probability 0.5. Dropout layers helps us to reduce generalization erros, it is a regularization method. During training, some number of layer outputs are randomly ignored or "dropped out." This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different "view" of the configured layer. This report explains in details how dropout is an effective technique for regularization and preventing the co-adaptation of neurons: [7].

After these two layers, I have added two LSTM layers. E.g., stacking two LSTMs together to form a stacked LSTM means, with the second LSTM taking in outputs of the first LSTM and computing the final results. After each LSTM layer I have also added another dropout to minimize generalization erros.

For the final layer, I have used a TimeDistributed dense (fully connected) layer with softmax activation. This layer takes output of the previous LSTM layer (size: batch_size, sequence_size, hidden_size) as input and creates an independent dense layer for each time step. Then using the softmax activation function produce normalized output values.
Here is the model architecture:

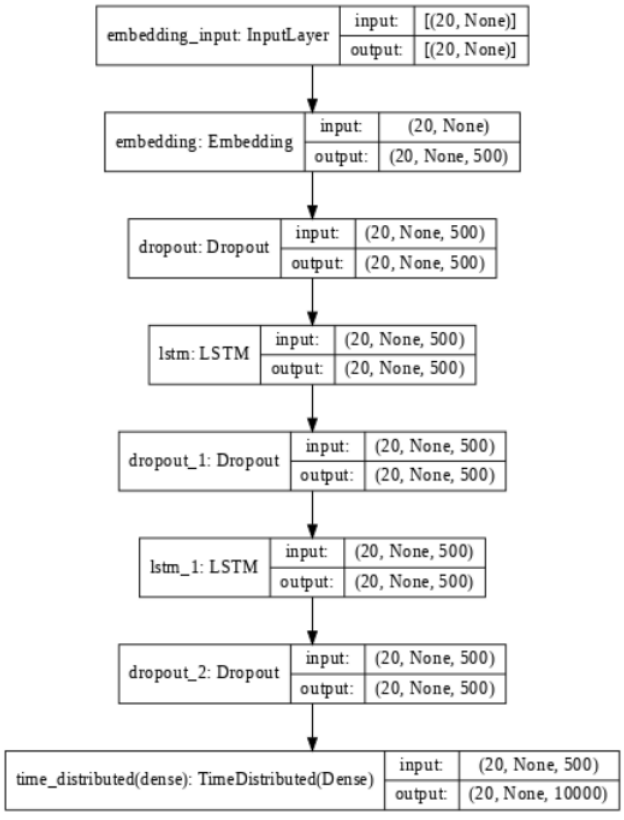

Fig. 3: Model structure

During the training and evaluation phase *Sparse Categorical Crossentropy* and *Perplexity* have been used. Sparse Categorical Crossentropy is almost same as Cross entropy, both have the same loss function. Crossentropy is used only if we have hot-encoded output. But in our case we have integers as output, indexes of classes, and Sparse Categorical Crossentropy is suitable to deal with this type of output. With Sparse Categorical Crossentropy we can leave the output integers as they are, but yet can benefit from crossentropy loss.

Now let's see how Crossentropy loss is formulated. For a model *m* modeling a reality, and the real probability distribution *p* of that reality, say the probability of a word following a sequence of words. The cross entropy can be described as:

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

Where:

n : 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, I have used **Stochastic Gradient Descent (SGD)** with **Momentum** (momentum value equal to *0.9*). I have tried also ADAM as optimizer but the result was not that good.

6. Experiments and Results

The model has been tained and tested with different configurations. I have tried to figure out the best embedding size, hidden size, which minimize the perplexity value.
Here are the results:

Embedding size	Hidden size	Early stopped (epoch)	Test loss	Test PPL
350	350	37	4.669	116.553
500	500	26	4.631	112.536
650	650	37	4.638	114.356

TABLE I: Results

Here we can see that the **test perplexity** is best with the second configuration, which is: **Embedding Size = 500, Hidden size = 500** . With this setup I have achieved test perplexity equal to **112.536**. We can also notice that, the difference between three configurations in term of perplexity is not that much, gain of perplexity is vey little.

Here are some plots which highlight the loss and the perplexity per number of epochs during the training and validation phases (see next page):

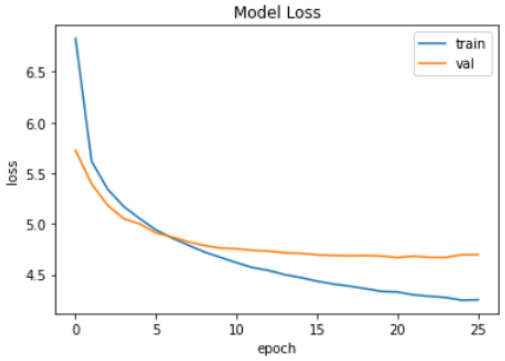

Fig. 4: Loss during training and validation

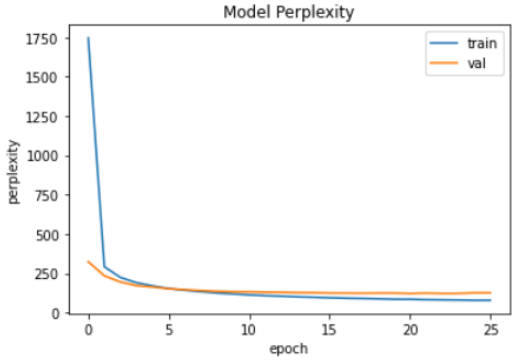

Fig. 5: Perplexity during training and validation

References

[1] Kushal Vala - “N-gram Language Models”, https://towardsdatascience.com/n-gram-language-models-af6085435eeb.

[2] Ben Lutkevich - “Language modeling”, https://searchenterpriseai.techtarget.com/definition/language-modeling.

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

[4] Colah’s blog, “Understanding LSTM Networks”, https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[5] Ampatishan Sivalingam - “Why do we need LSTM”, https://towardsdatascience.com/why-do-we-need-lstm-a343836ec4bc.

[6] 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.

[7] 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.

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

[9] Medium - “ICLR 2019 -Fast as Adam and Good as SGD - New Optimizer Has Both”, https://medium.com/syncedreview/iclr-2019-fast-as-adam-good-as-sgd-new-optimizer-has-both-78e37e8f9a34.