UDC - 004.855.5

RHYME FINDER: GENERATING SONG LYRICS USING BACKWARDS RHYMES

Arman G. Zakaryan
Institute for Informatics and Automation Problems of NAS RA
1 P.Sevak Str., Yerevan, RA
arman_zakaryan20@alumni.aua.am
ORCID: 0000-0002-1622-256X
Republic of Armenia

Abstract

The article describes a method of generating song lyrics in a backwards fashion. We present our own Rhyme Finder, a Python library which generates rhymes for the given word or phrase based on a large and robust input data and ranks the potential rhymes based on the built-in rhyme quality checker. After finding the appropriate rhymes, it uses LSTMs to generate the lyrics backwards thus showing some promising results by being trained only on a small sample of data with limited resources.

Key words: LSTM, Recurrent Neural Networks, deep learning, song lyrics, rhyme generation.

Introduction

Recurrent neural networks are very powerful tools for dealing with sequential data such as text, time series etc. Researchers noticed how an RNN of a fairly simple architecture can be used to generate text using the style of the author on the texts' of which it was trained. A simple LSTM network was able to generate structurally very similar text after being trained on Shakespeare's works [1]. This presents a great potential for RNNs and deep learning models in general as it is interesting to see how far this can be taken, in terms of "imitating art", as it's one of the hottest topics of debate when it comes to AI. And while song lyrics could also be considered a piece of text, they have some big technical differences from prose. In songs, all lines should have about the same number of syllables, while the last couple of syllables must rhyme with each other. Furthermore, it should be original, in a sense that it shouldn't just repeat whatever's in the dataset but create new/never seen before content. This presents some constraints to the typical text generation problem. In this work we describe how we overcome that problem and generate rhyming song lyrics.

Problem Statement and Related Work

Inspired by Karpathy's results [1], many researchers took on the task of using deep learning techniques to generate song lyrics/poetry. Most of the work on this subject, however, is not very formal and one can find a lot of examples like this scattered all over the GitHub and Medium. As a result, there's no formal evaluation of the models, just some examples of the predictions. Furthermore, even when the approaches are well documented, they are very different from each other which presents an issue in itself - comparability. The quality of one model's generated lyrics is difficult to compare to another one. That's why most of these models in the end are evaluated by humans, by rating them on a 1-5 scale in several categories (e.g. meaning, coherence etc.).

One approach to the problem is Dope Learning [3], which is a model that generates hip-hop lyrics. Their approach is to use entire lines as an input to the model and the output is another line.

While this approach performs good in terms of coherence and meaning, it lacks creativity. Nothing new is generated as a result of this, it's just a well rhyming collection of lines.

Another approach is the "content controlled" lyrics generation [2] which strips down the lines of the original songs to the most "key" phrases and then uses those to generate a full line. This approach provides more originality while losing some of the coherence and lyrical depth presented by the previous work.

In Neural Poetry [4], the authors generate poems by using a syllable level data which increases the originality even more but again loses more coherence as a tradeoff.

In all of these approaches, the authors end up with rhyming text generators, the comparison of which, as mentioned, is non-trivial. Hence, using human evaluation is not very accessible to most researchers while being effective.

Background

We leverage Recurrent Neural Networks (RNNs) in our approach, especially GRU and LSTM. RNNs are designed for processing sequential data. Unlike feedforward neural networks, RNNs can use their internal memory to process any sequence of characters. They have a non-linear mapping from a current input X_t and previous hidden state h_{t-1} to current hidden state h_t and the output o_t. And hidden state has a predefined size and stores features which are updated on each step and affect the final result of mapping.

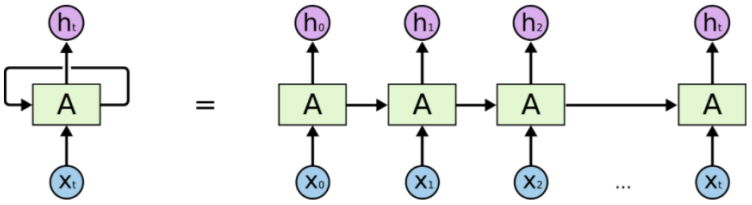

Fig. 1 Unrolled Recurrent Neural Network

More rigorously, the current state formula is:

$$h_t = f(h_{t-1}, x_t) \tag{1}$$

and with the activation function it looks like:

$$h_t = tanh(W_{hh}\, h_{t-1} + W_{hx}\, x_t) \tag{2}$$

where W_{hh} is the weight at the previous hidden state and W_{hx} is the weight at the current hidden state and x_t is the current input.

To be more effective, the network has to keep a "history" of the past inputs, instead of looking only at the previous layer. This is where LSTM comes in.

Long Term Short Memory (LSTM) networks are a variation of the traditional RNNs, which includes an input gate, forget gate and an output gate.

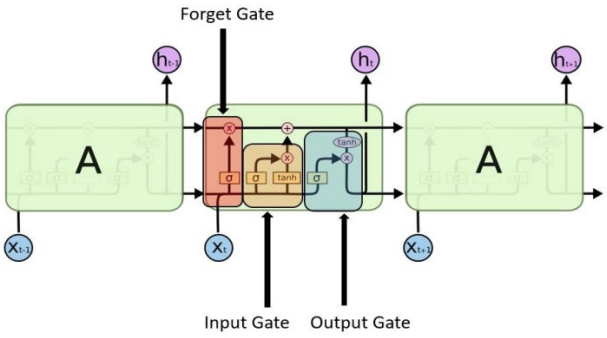

Fig. 2 Structure of an LSTM

The function of the input gate is to decide which values should modify the memory, leveraging the sigmoid function and assigns importance to each value using tanh.

$$i_t = \sigma(W_i\,[h_{t-1}, x_t] + b_i) \tag{3}$$

$$C_t = \tanh(W_C\,[h_{t-1}, x_t] + b_C) \tag{4}$$

Forget gate makes the decision on which block to ignore from the memory.

$$f_t = \sigma(W_f\,[h_{t-1}, x_t] + b_f) \tag{5}$$

And finally, the output gate combines the results of the previous gates to decide on the output and updates the hidden state.

$$o_t = \sigma(W_o\,[h_{t-1}, x_t] + b_o) \tag{6}$$

$$h_t = o_t * tanh(C_t) \tag{7}$$

In language modeling, LSTM is usually followed by a linear layer, which gives the probability of the next word from the set of all the words. This is essentially a multiclass classification problem, so the appropriate loss function here is the cross entropy.

$$L(y, y_{pred}) = -\Sigma y_i \, log \log \left(y_{pred_i}\right) \tag{8}$$

These layers are preceded by an embedding layer in our architecture which is used to represent the label encoded words by dense vectors.

So, our final architecture is:

1. Embedding Layer
2. LSTM Layer(s)
3. Linear Layer

Method

The data is scraped from the most reputable lyrics source - the website genius.com [9]. The scraping is done through their API, which, given a list of artists, scrapes all the songs' lyrics for which

the artist has the primary credits. The size of the dataset is about ~2.1 million characters/74K words, with lyrics from around ~20 artists (mostly hip-hop for now). Note that for this type of problems this is a relatively small dataset.

For rhyming we wrote our own module - RhymeFinder. We assume that in the dataset on which the model is being trained, we already have a lot of information about the rhymes. So, to find rhymes for a word or a phrase, we search for them in the dataset itself. RhymeFinder finds all the lines that end with the given phrase and analyzes all the lines that are either before or after the selected lines. It generates a candidate rhyme from it. RhymeFinder tries to match the number of syllables in the queried text with the candidate rhymes by leveraging The CMU Pronouncing dictionary [6]. Furthermore, it evaluates the rhymes based on syllable similarity using Levenshtein distance [7] and thus returns a ranked dictionary of candidate rhymes. From there we can control the quality of rhymes by fixing a minimum threshold for acceptable rhyme similarity.

We feed the data into the network in a reversed fashion. For example, the lines

Talk about some things we can't undo
You just send the pin, I can find you

becomes

you find can I pin the send just you
undo can’t we things some about talk

The generation process is as follows:

1. Pick the last word/phrase of the last line - “find you”
2. Reverse it - “you find”
3. Generate the line - “you find can I pin the send just you”
4. From the original selected phrase (find you), generate set of possible rhymes
5. Pick the desired rhyme - undo
6. (Optional for single tokens) Reverse it - undo
7. Generate the second line - undo can’t we things some about talk
8. Now after reversing the entire thing, we get the natural order.

This method allows us to generate the rhymes backwards, by using our desired rhymes, and the reverse order of lines provides us contextual coherence.

Experiment Results and Benchmarks

The first iteration of the model was a character level LSTM, which ended up generating a lot of nonsense words. While it also provided a lot more originality, the amount of random text tokens were ruining the aesthetics of the final result. We decided to move forward from there with word level models as it will eliminate the issue of nonsense words.

At first, we tried to use random samples during each epoch and train on that. This approach didn’t take us far, as the loss hardly changed over ~50 epochs. Dealing with limited GPU resources (NVIDIA Quadro P2000), we decided to train on the entire dataset for each epoch. This approach was significantly better, but it took a really long time to train so much that we decided not to move forward with it as it would take weeks to finish. We found a compromise which was simply not utilizing most of the dataset and just train it on a smaller sample. This method was a good tradeoff between training time and results.

We observed the following perplexity scores and losses for the models after 50 epochs:

Table 1

Comparison of our approaches

-	Random Sampling	Down sampling	Entire Dataset
Cross-Entropy Loss	4.8	**3.1**	-
Perplexity	121.5	**24.1**	-

Comparing it with other solutions such as Creative GANs [5] which were trained on significantly larger datasets and with more advanced methods, we observe results:

Table 2

Our model vs Creative GAN

-	Creative GAN	Our Model
Perplexity	**17.11**	24.1

With our available resources, the results are very promising.
One result of our generated verse is:

bout to f--k that b---h yes
she got y'all know this gangster chest no lisp
tried to pull up for the 40000 dollar dress
find a stack it up to your so fresh

More could be found and generated in our GitHub repo [10].

Conclusion

In this work we presented an approach of generating song lyrics using LSTMs. We developed our own rhyming module with the help of which we ensured that the generated lines rhyme (to some extent). Using very limited resources, our method was able to generate lyrics that are comparable with significantly complex models. Many aspects of the work can be improved starting from increasing the RhymeFinder efficiency (to make it avoid circular rhymes) to adding more training data and training with stronger machines to achieve more diverse results.

References

1. Karpathy A. The Unreasonable Effectiveness of Recurrent Neural Networks (May 21, 2015) http://karpathy.github.io/2015/05/21/rnn-effectiveness/ (accessed: April 06, 2021)
2. Nikolov et al, Conditional Rap Lyrics Generation with Devoicing Auto encoders (April, 2020) https://arxiv.org/pdf/2004.03965v1.pdf (accessed: April 06, 2021)
3. Malmi et al, DopeLearning: A Computational Approach to Rap Lyrics Generation (June, 2016) https://arxiv.org/pdf/1505.04771.pdf (accessed: April 06, 2021)
4. Zugarini et al, Neural Poetry: Learning to Generate Poems using Syllables (Sep, 2016) https://arxiv.org/pdf/1908.08861.pdf (accessed: April 06, 2021)
5. Saeed et al, Creative GANs for generating poems, lyrics, and metaphors (Sep, 2019) https://arxiv.org/pdf/1909.09534.pdf (accessed: April 06, 2021)
6. CMU Dictionary: http://www.speech.cs.cmu.edu/cgi-bin/cmudict (accessed: April 06, 2021)
7. Gilleland M., Levenshtein Distance, in Three Flavors https://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Spring2006/assignments/editdistance/Levenshtein%20Distance.htm (accessed: April 06, 2021)
8. Gudikandula, Recurrent Neural Networks and LSTMs explained, Medium (Mar, 2019) https://medium.com/@purnasaigudikandula/recurrent-neural-networks-and-lstm-explained-7f51c7f6bbb9 (accessed: April 06, 2021)

9. Genius.com https://genius.com/ (accessed: April 06, 2021)
10. Our GitHub: https://github.com/armzak1/lyrics_generator (Training, Generator) (accessed: April 06, 2021)

RHYME FINDER: ԵՐԳԻ ԲԱՌԵՐԻ ՍՏԵՂԾՈՒՄ՝ ՕԳՏԱԳՈԾԵԼՈՎ ՀԵՏԱԴԱՐՁ ՀԱՆԳԱՎՈՐՈՒՄՆԵՐ

Ա.Գ. Զաքարյան
ՀՀ ԳԱԱ Ինֆորմատիկայի և ավտոմատացման պրոբլեմների ինստիտուտ

Այս աշխատանքում մենք նկարագրում ենք հետադարձ եղանակով երգի բառեր ստեղծելու մեթոդ: Մենք ներկայացնում ենք մեր սեփական RhymeFinder-ը՝ Python գրադարան, որը ստեղծում է հանգավորումներ տրված բառի կամ արտահայտության համար՝ հիմնված ընդարձակ և վստահելի մուտքային տվյալների վրա, և դասակարգում է պոտենցիալ հանգավորումները՝ հիմնվելով ներդրված հանգի որակի ստուգիչի վրա: Համապատասխան հանգավորումներ գտնելուց հետո այն օգտագործում է LSTM-ներ՝ հետադարձ կերպով երգի բառեր ստեղծելու համար՝ այդպիսով ցույց տալով որոշ խոստումնալից արդյունքներ՝ դուրս բերվելով միայն սահմանափակ ռեսուրսներով տվյալների փոքր նմուշից:

Բանալի բառեր. LSTM, ռեկուրենտ նեյրոնային ցանցեր, խոր ուսուցում, երգի բառեր, հանգավորումների ստեղծում:

RHYME FINDER: СОЗДАНИЕ ТЕКСТОВ ПЕСЕН С ИСПОЛЬЗОВАНИЕМ МЕТОДА ОБРАТНЫХ РИФМ

А.Г. Закарян
Институт проблем информатики и автоматизации НАН РА

В этой работе мы описываем метод создания текстов песен в обратном порядке. Мы представляем вам разработанную нами программу RhymeFinder, которая генерирует рифмы для данного слова или фразы на основе больших и надежных входных данных и ранжирует потенциальные рифмы на основе встроенного средства проверки качества рифм. После нахождения подходящих рифм он использует LSTM для создания текстов в обратном порядке, показывая тем самым некоторые многообещающие результаты, обучаясь только на небольшой выборке данных с ограниченными ресурсами.

Ключевые слова: LSTM, рекуррентные нейронные сети, глубокое обучение, тексты песен, генерация рифм.

Ներկայացվել է՝ 06.04.2021թ.
Գրախոսման է ուղարկվել՝ 12.04.2021թ.
Երաշխավորվել է տպագրության՝ 28.04.2021թ.