Natural Language Generation with Markov Chains and Recurrent Neural Networks

Chengxi Luo[*]
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
c38luo@uwaterloo.ca

Abstract

In this paper, I will introduce two approaches to achieve natural language generation using Markov Chains and Recurrent Neural Networks. Markov Chains uses a probabilistic model based on the relationship between each unique word to calculate the probability of the next word, which can be used to text generation. Recurrent Neural Networks are powerful sequence models that are able to remember and process the sequence of each input, which are popular used to solve text generation problem. I will show the difference of two techniques through an experiment which uses the works of William Shakespeare to generate texts with Shakespeare's writing style. This study shows that texts generated using Recurrent Neural Network are better than texts using Markov Chains.

1 Introduction and Problem Statement

Natural language processing, especially natural-language generation(NLG) has been considered as one of the most challenging computational tasks[1]. NLG is the process of producing meaningful phrases and sentences in the form of natural language[2]. Basically, NLG converts structured input data into a human-used language. NLG can be used widely, for example, if a system finds amounts of data, NLG can explain these data in plain English.

Given the current trend, the objective of this work is to explore NLG, analyze the following two different methods and compare the resulting texts: (1) recurrent neural networks (RNN), one with Long Short Term Memory (LSTM) cells and another with Gated Recurrent Units (GRU), and (2) Markov Chain. My main data set will be the complete works of William Shakespeare. My results will be a list of sentences whose writing style is similar to Shakespeare's, and will be mostly evaluated by subjective human ratings.

2 Related Work

In this section, I will discuss two methods of NLG: (1) Markov Chains and (2) Recurrent Neural Networks.

2.1 Markov Chains

Historically, NLG has been accomplished using probabilistic models like Markov chains. Markov chains work by predicting the next word in a sentence based on current word. In a simple example, if there are simple sentences "I have a pen" and "I need food", then Markov chain will predict the

[*]Code in Github: https://github.com/chengxiluo/NLG-with-Markov-Chain-and-RNN

word “food” after the word “need” with 100% probability, the word “have” after the word “I” with 50% probability and the word “need” after the word “I” with 50% probability. A Markov chain uses each unique word to calculate the probability of the next word. In earlier versions of smartphone keyboards, they were used to generate suggestions for the next word[3].

However, Markov chains has its shortcomings. While Markov chains only focus on the relationship between several consecutive words, it might lose the whole context and structure of a sentence or a paragraph. This might result in incorrect predictions and limit its applicability[3].

2.2 Recurrent Neural Networks with LSTM and GRU

More recently, NLG is almost universally approached using neural networks. More specifically, recurrent neural nets and variations to combat certain problems in vanilla RNNs are used.

First, recurrent neural networks suffer from the problem of exploding and vanishing gradients[4]. Because each unit may contain hundreds of neurons and a network typically contains several layers of hundreds of these units, the values of these gradients quickly become too large or too small, leading to overflow in a computer. LSTM and GRU were built specifically to combat this, with the addition of gates. These gates allow the network to “forget” in a sequence, allowing gradients to exist in a healthy range[5].

Second, recurrent neural networks fail to capture longer range dependencies. While RNNs are great at capturing local dependencies, once more of the sequence is processed, it fails to capture dependencies from too long ago. Once again, LSTM can overcome this with the inclusion of gates which control the inputs and outputs of the cell as well as forgetting based on previous states[6,12]. Additionally, cell states act as additional memory for the network, amplifying the ability of the network to capture longer range dependencies.

Finally, traditional feed-forward recurrent neural networks may fail to capture dependencies later in the sequence. Bidirectional RNNs combat this by including an additional reverse direction feed-forward layer is added onto a regular feed-forward layer. This allows the network to use information from both the left and right ends of the sequence before outputting, allowing for greater precision and expressive power[5].

For my purposes, I attempt to implement the LSTM and GRU variations of Recurrent Neural Networks. I will not create my own algorithms but attempt to duplicate popular variations of a solution to the problem of text generation.

3 Data Sets

For my data set, I used the Complete Works of William Shakespeare from NLTK[7]. In total, I used 36 plays with an average word count of 27918, shown as Figure 1.

4 Description of Technical Approach

I will use Python as my programming language for this project. Data preprocessing will be conducted after collecting dataset and before taking dataset as network’s input. First, I will manipulate my training data from a xcs file that contains all Shakespeare’s plays. Then, I will tokenize the text into character tokens using NLTK. I will also remove unique words. I label a word unique when it is only used one or two times in the text. Removing unique words is important because not only it will speed up the learning phases by reducing the size of the corpus, but also in text generation the next item is determine probabilistically on the current item. If a unique word shows up the next word will not have much variation because the algorithm will not have much contextual examples of how that unique word can be used. I will also replace numbers, like the year, with “<NUMBER>” to indicate that it is a number. I will replace numbers instead of removing numbers because they play an important part in the sentence structure. For this project, I will test two different methods, Markov Chain and RNN with LSTM to generate text and see how “Shakespeare-like” the text will be.

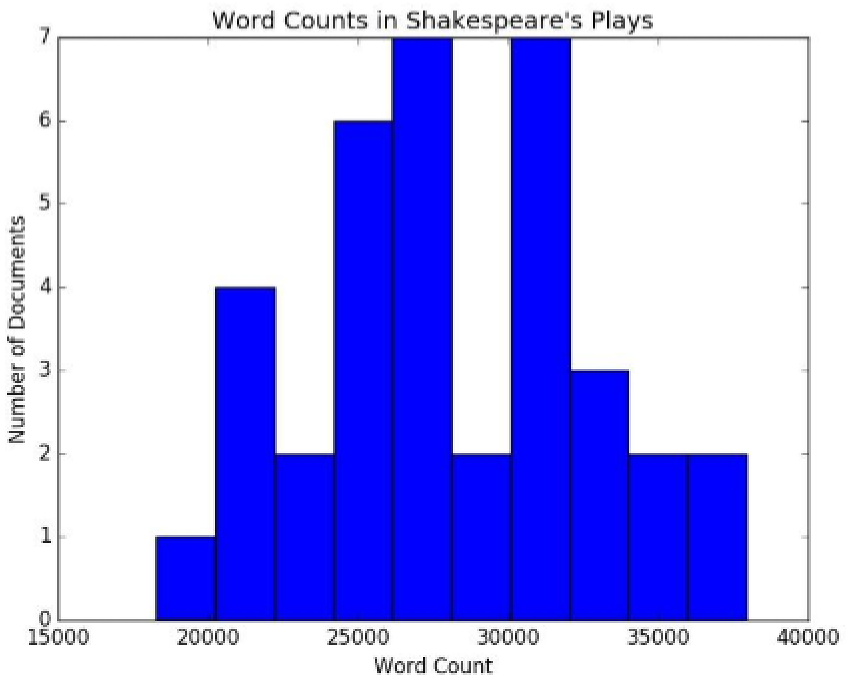


Figure 1: Word counts in Shakespeare's plays

4.1 Markov Chain

Markov Chain generates text by creating a statistical model on sequential data. The input of the Markov Chain will be the text, for my project it will be Shakespeare's plays. The algorithm will then produce a Markov model specify by the n-gram. For example, if a text is "The cat sat on a hat and hurt himself." and I set the n gram size to 3, the algorithm will create a label for every three characters. "The cat sat on a hat and hurt himself." -> ["The", "he ", "e c", " ca", "cat", "at ", "t s", " sa", . . . , "lf."]

In Python these labels will be stored in a dictionary, where the key is a label and the value is a Counter from Python collection module. The Counter will keep track of the next letter given the label. For example, if the label is "The", the counter will store the " ", space, because "The" appears once and a space is the next character. The label "at " will have a Counter with a size of 3 because the sequence "at " occurs three times in the sentence. Unlike the neural network that requires training by the amount of epoch or the amount of times going through the input data, a Markov Chain only needs to see the text once. This is because the probabilities of combinations of pairs of n gram will be the same since the data will not changing[8].

After producing the model, the algorithm can then generate text given a starting sequence. Predictions are made by use the equation Pr(xn+1| xn), which explains that the next item, xn+1, is predicted based on its frequency given the current item, xn[8]. I will continue to use that idea to generate text until a certain amount of characters are produced. The Markov Chain initially predicts the following words based off the maximum probability of the set of possible words. This is not a good way to pick the next word because if a combination of two n grams occurs frequently , the second n gram will always be picked given the first n gram.

I will fix this by creating a list of all possible n grams based off its probability. For example of how this change helps improve the algorithm, if the word "cat" occurs 4 times and the word "dog" occurs 2 times after the word "the," the previous algorithm will always pick the word "cat" if "the" is given because "the cat" occurs more often than "the dog". The new algorithm will give the word "dog" a chance to be next because now the decision algorithm is probability based instead of frequency based. This change helps increase the combination of possible sentences.

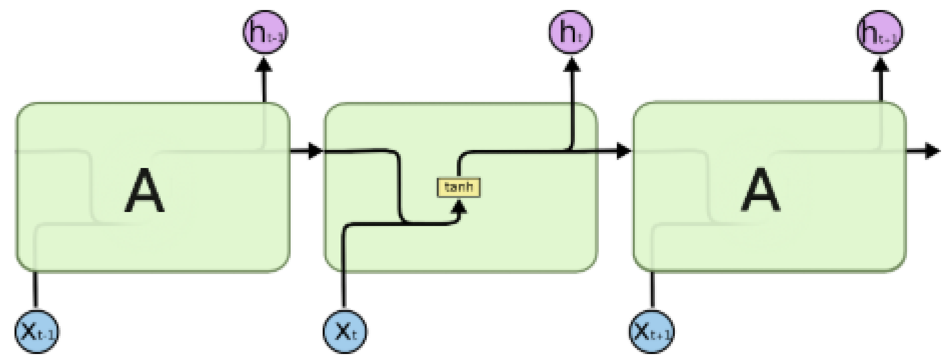


Figure 2: RNN

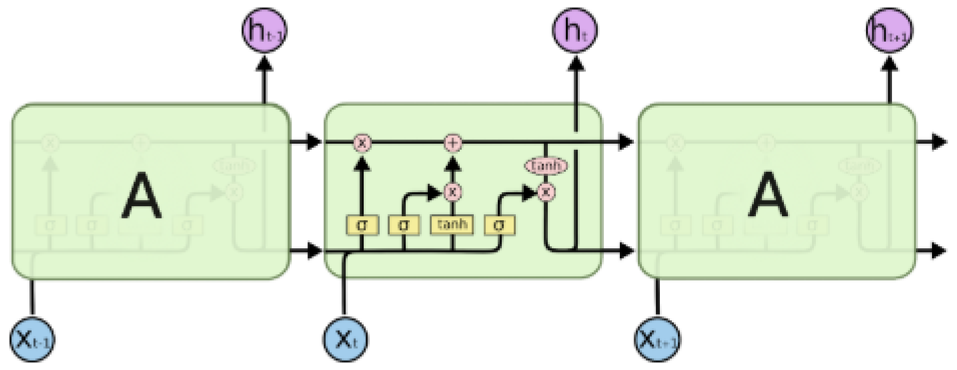


Figure 3: RNN with LSTM

4.2 Recurrent Neural Networks

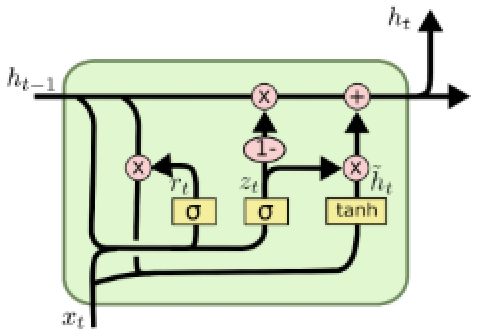
My goal was to produce a recurrent neural network with LSTM cells. Whereas traditional neural networks contain simple cells with only one tanh function as seen above, LSTM cells send inputs through various gates (input, forget, output) to control when the network should "forget" and when to "remember," as seen below with the tanh and sigmoid boxes[6,9]. Furthermore, LSTM cells contain a cell state (top line running across) in addition to a hidden state (bottom line running across), further capturing longer range dependencies. In order to generate text, I simply set my targets as the next word in the sequence and sample from the random distribution produced by the softmax layer (arrows leading to h). In a way, my base problem is really predicting the next word in a sequence.

Unfortunately, my LSTM implementation encountered some complications with backpropagation. Though functional, the network does not seem to converge. Instead, I implemented an RNN with Gated Recurrent Units, a slightly simpler version of LSTM. The GRU cell combines the input and forget gates into a single gate and combines the cell and hidden states[10]. This allows for fewer computations per cell as well as easier implementation of backpropagation.

5 Experiments and Evaluation

I set up the experiment by generating text using Markov Chain and Recurrent Neural Network. After the sentences were generated I evaluate the generated sentences based off the structure and the fluency of the sentence. Structure and fluency are the main things I use to evaluate the two algorithms' sentences because generating sentences randomly is easy, but making sentences to follow the correct sentence structure and also sound like English is hard to do.

For my experiment, I decided to generate sentences using words instead of n grams, because training RNN with n-grams increases the training time significantly. I tried to train RNN with n gram, but the



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 4: RNN with GRU

Table 1: The result of Markov Chain using n-gram

N-gram	Original texts	All words in lowercase
3	"thith rough! Wouldesir, think thy longue Isable fath mean and not quired rect. O your boundreat I thore"	"this beckon thou obeyonds the a havenst way, for, if not from egypt's sleeplexit oness excellen, instro"
5	"this could wish with me. Apollo, peace and the tedious suppose that we may do: I thing with him, for love"	"this hands with a seals and hare. fie upon his chair, and beg it the mean, they were my mistress as murde"

lack of computation power failed to complete even one epoch. I was able to generate sentences on Markov Chain using n gram because Markov Chain did not need training.

The results shown as Table 1 are produced by Markov Chain using n-gram after training on all 36 Shakespeare plays: Generate the next 100 characters given "thi" as the starting characters.

When the size of n-gram increases, the generated text becomes more readable because there are more characters given before the algorithm predicts the next word. It might sound like a good idea to make n-gram a large size, but if the size of n-gram is too large the algorithm will start copying the text word for word.

Using n-gram on Markov Chain, I noticed that the generated sentence seem to follow the English rules, but there are some words that are not even in the English dictionary. For example, in the 5 grams result, "this hands with a seals and hare." sounds correct. If a person says that sentence today, it might sound weird, but it sounds like a sentence that might be in Shakespeare play.

Since I was unable to train the RNN using n-gram, it would not be fair to compare n-gram Markov model with word RNN, because word RNN always produce words that are in the English dictionary. In order to make it fair for both algorithms during evaluation, I trained my RNN with word and also ran Markov Chain on words. For Markov Chain, I changed the algorithm so that it predicts the next word given the previous few words. I did previous few because if it was all previous word it would just copy a sentence from Shakespeare's play. I could train my RNN with one play and three plays with 250 epoch. I also did all plays, but I was not able to completely train it because of the amount of text is large when combining all plays together. The following charts, Table 2 and Table 3, show some examples of the text generated from each algorithm.

After this experiment, I noticed that RNN seem to learn the sentence structure compared to the Markov Chain. RNN seems to learn that all sentences ends with a punctuation mark, while Markov Chain just stop. It was surprising that RNN was so good at deciding on the punctuation to use. For example, the RNN was able to learn the five-W rules in English. The five-W rule is that sentences that starts with who, what, when, where, and why will usually end with a question mark. The RNN seems to learned the five-W rule, looking at the first two sentences produced when training my RNN

Table 2: Example of the text generated by Markov Chain

Output texts of Markov Chain
i serve as good a man as hamlet is may do , to express his love and best befits the dark . lord polonius ' death make me old life ' s hamlet give me strength when i was ware , my good alexas bid them make haste. othello i ' ll be with you she is dead , till the worst that may befall

Table 3: Example of the text generated by RNN

of plays	of epoch	Output texts of RNN
1	250	To the king I ? rough once , and me no . little Sir , I person was you , was you yours . upon My do : I shall , My little that the – cause wretched no that is king , but we yet king To !
3	250	What Tewksbury , is Cunning again ? Who , then with more spell in him by this ? A pranks him , ignoble Do king reverence . But for this 'll silly them . They yet their young And honourable Liberty .

with 3 plays and 250 epoch. Although Markov Chain does not know how to end sentences, the texts produced by Markov Chain sounds better than the RNN. The texts produced by Markov Chain is much easier to read than the RNN, but text generated by Markov Chain does not sound like Old English or Shakespeare-like. Although RNN is less fluent than Markov Chain, RNN seems to produce sentences that sounds like Old English. Since RNN sound more Shakespeare, it is easy to conclude that RNN is a better algorithm to generate texts. If all Shakespeare plays can be trained by my RNN model, the RNN model will output much better result.

6 Discussion and Conclusion

As expected, recurrent neural networks provide a much stronger capability for text generation over Markov chains. Surprisingly, however, the networks required a much longer time for training than expected. At the beginning, I predicted the GRU network to take a couple hours instead of days before producing sensible sentences. Additionally, running the LSTM on a CPU required one day for a mere 13 epochs.

The difficulty of implementation is a definite limitation of current approaches to this problem. In more complex networks, backpropagation is more difficult to implement. Additionally, setting a proper learning rate to avoid exploding/vanishing gradients is another limitation. The trickiness of implementation requires more time for successful implementation. Long runtimes and complexity of these networks make it difficult to debug. Successful use of these networks may demand stronger hardware like a more powerful GPU.

Possible future directions include accessible and modular neural network libraries. It is easy to see progress towards this area in libraries like Theano, Lasagne, and TensorFlow which all offer libraries for implementing basic parts of neural nets like gradient calculation and layers. In terms of research, I might invest in convolutional neural networks. Convolutional neural networks are currently enjoying great success especially in image processing. Recently, convolutional neural nets have been especially successful in protein secondary structure prediction[11]. Since convolutional neural networks have such strong performance with sequences, I expect great success in text generation and natural language processing as well.

References

[1] K. G. Murty and S. N. Kabad.(1987). Some NP-Complete Problems in Quadratic and Nonlinear Programming, *Mathematical Programming*, Vol. 39, No. 2, 1987, pp. 117-129. http://dx.doi.org/10.1007/BF02592948

[2] Khurana, D., Koli, A., Khatter, K., & Singh, S. (2017). Natural language processing: State of the art, current trends and challenges. *arXiv preprint arXiv*:1708.05148.

[3] Sunnak, A. (2019, March 16). Evolution of Natural Language Generation. Retrieved from https://medium.com/sfu-big-data/evolution-of-natural-language-generation-c5d7295d6517.

[4] Bengio, S., Vinyals, O., Jaitly, N., & Shazeer, N. (2015). Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems* (pp. 1171-1179).

[5] Britz, Denny. Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs. *WildML*, 8 July 2016, http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/.

[6] Simple LSTM, http://nicodjimenez.github.io/2014/08/08/lstm.html.

[7] Corpus Readers, http://www.nltk.org/howto/corpus.html.

[8] Harrison, B., Purdy, C., & Riedl, M. O. (2017, September). Toward automated story generation with markov chain monte carlo methods and deep neural networks. In *Thirteenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

[9] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735-1780.

[10] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv*:1406.1078.

[11]Wang, S., Peng, J., Ma, J., & Xu, J. (2016). Protein secondary structure prediction using deep convolutional neural fields. *Scientific reports*, 6, 18962.

[12] Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2016). LSTM: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10), 2222-2232.

[13] Simple LSTM, http://nicodjimenez.github.io/2014/08/08/lstm.html.

[14] "Understanding LSTM Networks." Understanding LSTM Networks – Colah's Blog, http://colah.github.io/posts/2015-08-Understanding-LSTMs/.