A Study in Recurrency: Generating a Sherlock Holmes Story with an LSTM RNN

Amy Bryce, Theresa Pekarek-Rosin
a.bryce9@gmail.com, 4pekarek@informatik.uni-hamburg.de
Knowledge Processing in Intelligent Systems: Practical Seminar
Knowledge Technology, WTM, Department of Informatics, University of Hamburg

***Abstract*—Literary authors can only produce a finite amount of original work. That is to say, once a favored author retires from their talent, their fans are often left wanting more. A way to feed this yearning has been for others to adopt the original stories into a multitude of media. This is especially evident with the much beloved Sherlock Holmes stories written by Sir Arthur Conan Doyle. Since Doyle's retirement, there have been endless variants of the Sherlock Holmes stories produced, but nothing is quite like the original stories. To address this, we asked ourselves if we could use machine learning techniques to mimic Doyle's writing style and produce new Sherlock Holmes short stories that would have an indistinguishable style from the original author. We think this could be a new and unique way to produce original-like work from long-lost favored authors. In our paper, we train a long short-term memory recurrent neural network, aided by a pre-trained vocabulary, and using a word-based text generation approach. By carefully prepossessing our training data to capture what we believe are the essential elements to Doyle's writing style, we were able to produce a similar grammatical structure with reasonable, albeit incoherent at times, storytelling. We believe that our implementation could be used as a stepping stone for further research to achieve our goal of producing original-like works.**

I. Introduction

Sherlock Holmes, created by Sir Arthur Conan Doyle, is one of the most famous literary characters of all time. The stories about Mr. Holmes, a private detective, and his side-kick, Dr. John Watson, have been adapted in almost every medium conceivable. The reason for the stories' enduring success was Doyle's keen understanding of contemporary reader engagement and publishing. By keeping the stories connected through one main recurring character, but also keeping every individual story self-contained, reader loyalty to the publishing newspaper increased while new readers were easily won [13]. Within these stories, themes and plots were often repeated [18] and thanks to the particular structure of the short stories and Doyle's unique word usage [17], newly written stories are immediately recognizable.

This presents a unique challenge for artificially intelligent (AI) text generation. Since recurrent neural networks (RNNs), especially long short-term memory (LSTM) [12] RNNs, are known to pick up the underlying structure of both sentences and longer dependencies [10], we wanted to examine how well one of them would be able to replicate the structure of a Sherlock Holmes short story. For that we built a word-based story generator using an LSTM model. In our model, we use Word2Vec word embedding [14] that we train on the collection of all Sherlock Holmes short stories. We hypothesize that our model is able to identify and learn the underlying structure of the stories and to replicate it when prompted.

We will present our implementation and findings in this paper as follows: First, we cover some background knowledge in Section II. Then, we discuss in detail how we implemented our model in Section III using Subsections III-A, III-B, and III-C. We then evaluate our model and discuss our findings in Sections IV and V, respectively. Finally, we conclude this paper with a brief summary in Section VI.

II. Background

Before we discuss the implementation of our model, we need to first define a few concepts in more detail: recurrent neural networks, long short-term memory, and word representations in vector space.

A. Recurrent neural networks (RNNs)

RNNs are an optimal tool for the generation of sequences, allowing for the influence of prior information about multiple time steps on the generated output [10]. RNNs usually consist of the following basic structure: an input layer, a hidden layer, and an output layer. The input layer receives the initial pre-processed data, the hidden layer has a recurrent connection that allows the use of the aforementioned prior information, or context, for the calculation of the output [4], and the output layer produces estimates on what comes next in a sequence. An abstraction of the basic structure of an RNN can be viewed in Figure 1.

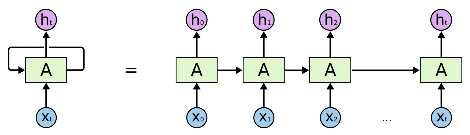


Figure 1. An example how RNNs retain information over multiple steps [15]. x_i is the input, A is a neural network and h_i is generated output. Recurrency can be seen when observing the RNN as several networks that pass on their prior knowledge as output to the next network as input.

B. Long short-term memory (LSTM)

LSTMs are especially useful if the context of the input data needs to span over a large number of iterations. While RNNs tend to forget certain input values after a longer series of steps, LSTMs are able to recognize both short-term and long-term dependencies. How much and for how long input is remembered in the cell state of the LSTM model is regulated through an input gate, a forget gate and an output gate. The input gate handles the amount of new values entering the long-term memory, while the forget gate decides which of the stored information to remove. The output gate is responsible for choosing what is actually used for the calculation of the output from the prior LSTM cell state. The three gates and the cell states are each controlled through separate functions [12]. A visualization of the internal structure of an LSTM model can be seen in Figure 2.

LSTMs work really well with a word-based approach to text generation, because they are able to identify and learn the underlying structure within the training data [10]. This is one reason why we chose a word-based approach, another being that character-based text generation has a common problem with low accuracy, e.g. misspellings, and higher computational costs when training a model [4].

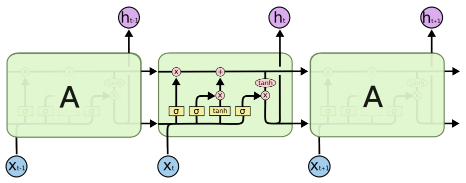


Figure 2. The internal structure of an LSTM model [15]. The A's represent neural networks, the boxes marked with σ and $tanh$ represent the input, forget, and output gates.

C. Word representations in vector space (Word2Vec)

Word2Vec is used as a form of text processing in an LSTM model, through word embedding, that helps to improve the quality of the generated output by grouping words that often appear close to each other in the training data set. This makes it possible for the model to either use the surrounding context to predict a word (Continuous Bag of Words) or generate context for a chosen word (Skip-gram) [14]. A diagram of these two approaches can be seen in Figure 3. In our implementation, we used the Skip-gram version of Word2Vec. The Word2Vec model, a simple neural network, is trained to learn the distance between words in the vocabulary based on how often they appear near each other. Then the weights of the model are saved and applied to the input layer of a generative model, e.g. our LSTM model, for word embedding. Therefore, using Word2Vec for text generation means that sequential words won't be chosen completely at random. Instead, they will be chosen based on their context within the training data set, with semantically related words tending to be closer to each other.

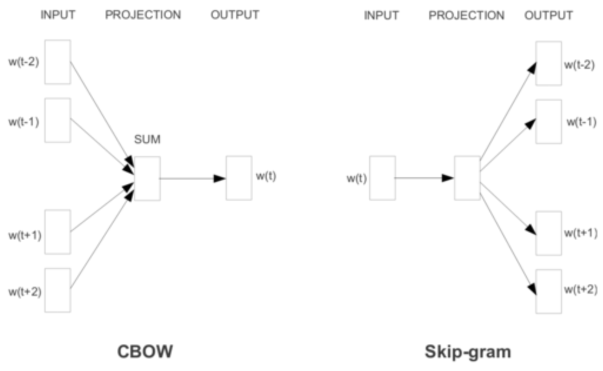


Figure 3. The two approaches to Word2Vec [14]: Continuous Bag of Words (CBOW) and Skip-gram. CBOW predicts a word based on context words, while Skip-gram predicts the context words based on a given word.

III. Method

We began the implementation of our story generator by following two tutorials on LSTM text generation models: most of what we used can be found in the Keras documentation [6], and we expanded on our original model with inspiration from a lyric generator implemented by blogger, Enrique A [2]. In addition, we opted to embed a Word2Vec model that we trained using the collection of Sherlock Holmes short stories [8] in the first layer of our story generator. The foundation of our Word2Vec implementation was inspired by the work from *Adventures in Machine Learning* blogger, Andy Thomas [19].

A. Architecture

To build our story generator, we chose to use an LSTM RNN which we implemented with a simple structure that has an input layer, hidden layer, and output layer. The first layer is the embedded layer, where we use our pre-trained Word2Vec model. The hidden layer is a bidirectional LSTM layer. And, our final layer is a fully connected dense layer, where we use softmax as the activation function. Figure 4 shows a high-level perspective of the architecture.

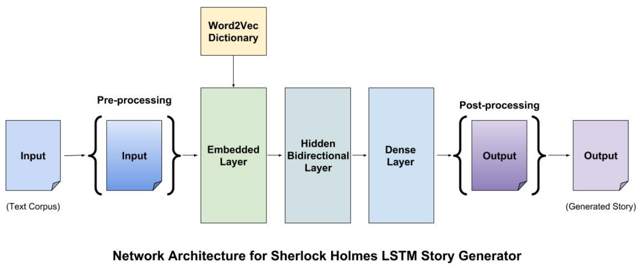


Figure 4. The architecture of our Sherlock Holmes LSTM model.

To construct our embedded layer, we implemented a Word2Vec model, which we trained on a data set consisting of all Sherlock Holmes short stories. This allows our model to select from a custom vocabulary of words that fits within our

Sherlock Holmes theme. Our Word2Vec model uses the Skip-gram approach, where it predicts a word that fits contextually with the prior word in a sequence. In addition, because using a softmax activation function can be computationally expensive with this approach [19], we used a ”negative sampling” method that keeps track of truly contextual words using binary classification by assigning a 1 to probable word choices, and a 0 to the improbable word choices, or ”negative samples” [14]. A flow chart of this process can be seen in Figure 5.

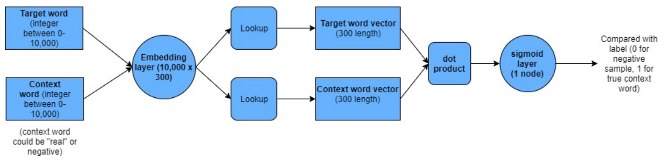

Target word (integer between 0-10,000)
Context word (integer between 0-10,000)
(context word could be "real" or negative)
Embedding layer (10,000 x 300)
Lookup
Lookup
Target word vector (300 length)
Context word vector (300 length)
dot product
sigmoid layer (1 node)
Compared with label (0 for negative sample, 1 for true context word)

Figure 5. Classifying contextual words vs. non-contextual words using the ”negative sampling” method [19].

For our hidden layer, we use a bidirectional LSTM layer with a dropout of 0.2 probability. A bidirectional LSTM allows our network to consider future states when predicting the next word in a sequence. This helps us overcome ambiguity when the prior words in a sequence could unfold into several different, yet meaningful, sequences [15]. By considering future states, we are better able to predict context, an important feature when implementing a story generator. This is accomplished by using a combination of forward and backward propagation before predicting a final output value. A flow of this process can be seen in Figure 6. Our dropout value of 0.2 means that we occasionally (20% of the time) drop some learned patterns from our network to prevent over-fitting during the training process [3]. Doing this also increases our training speed.

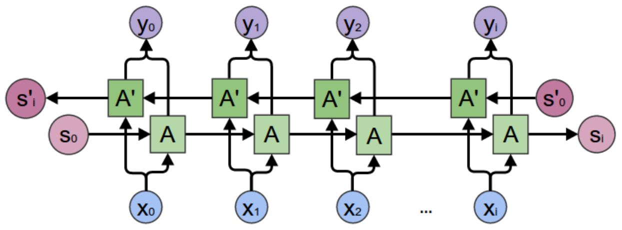


Figure 6. A bidirectional LSTM RNN where the A's represent neural networks, the s's represent the input and output sequences, and the x's and y's represent the intermediate input and output predictions, respectively [15].

Our final layer is a typical fully connected dense layer, requiring that its input vector is the same size as its output vector. Here we use a softmax activation function, allowing us to make predictions based on probability values. This means that our model will predict the next word in a sequence that has the highest probability. For example, if we predict that the next word in the sequence ”I'm” + ”driving” is going to be either ”around” with 60% probability or ”away” with 40% probability, then our model will output ”around”.

B. Data

We acquired our data set from Kaggle.com [8]. The data consists of the entire collection of Sherlock Holmes short stories by Sir Arthur Conan Doyle. The data was originally generated by being digitized by hand in the 1980's and is now available to the public domain. Due to the manual digitization process the data includes some misspellings, which artificially increases the amount of unique words in the vocabulary.

Pre-processing:

We pre-processed the data with two goals in mind: (1) we wanted to include what we thought were the most important elements that reflected Doyle's writing style, and (2) any pre-processing we did could be undone to the generated text in post-processing. Our pre-processing included the following steps:

1) Transform the input text into lowercase. This helps us avoid counting the same word as two separate words (e.g. ”the” and ”The”).
2) Bring all words separated across lines back together. This also accounts for multiple new lines and indents between when the word on the first line ends and the rest of the word is later on (e.g. ”exagg-[new line][new line]erated” to be: ”exaggerated[new line][new line]”).
3) Split the text by [new line] followed by at least one whitespace character; this allows us to keep the [new line]'s and subsequent whitespace characters. Essentially, we are splitting the text into paragraph strings.
4) Split the list of words (i.e. paragraphs) by any whitespaces (e.g. [new line], space, etc.) that exist between non-whitespace characters and remove them. Essentially, we are splitting the paragraphs into word strings.
5) Split the list of words by any remaining whitespace characters. This helps to preserve any notion of indentations or other special formatting found in the original text (i.e. some whitespace will be treated as word strings).
6) Split the list of words by anything that is not a letter, a number, an apostrophe or a hyphen. Through that we aim to preserve hyphenated words or contractions. Otherwise, we are splitting by any form of punctuation.
7) Separate single-quotes into their own words if they appear at the beginning of an existing word. And, we separate single-quotes into their own words if they appear at the end of an existing word. This preserves the use of quotes within quotes.

Finally, we returned the pre-processed list of words from the input text, as can be seen in Figure 7.

Post-processing:

When post-processing the output text generated by our story generator, we wanted to undo any of the pre-processing

Input text from the short story, "Silver Blaze":

```
  "I am afraid, Watson that I shall have to go," said Holmes as
we sat down together to our breakfast one morning.

  "Go! Where to?"

  "To Dartmoor; to King's Pyland."

  I was not surprised. Indeed, my only wonder was that he had
not already been mixed up in this extraordinary case, which was
the one topic of conversation through the length and breadth of
England. For a whole day my companion had rambled about the
room with his chin upon his chest and his brows knitted, charg-
ing and recharging his pipe with the strongest black tobacco, and
absolutely deaf to any of my questions or remarks. Fresh editions
of every paper had been sent up by our news agent, only to be
glanced over and tossed down into a corner.
```

Pre-processed text from the short story, "Silver Blaze":

```
[… '"', 'i', 'am', 'afraid', ',', 'watson', 'that', 'i', 'shall', 'have', 'to', 'go',
',', '"', 'said', 'holmes', 'as', 'we', 'sat', 'down', 'together', 'to', 'our',
'breakfast', 'one', 'morning', '.', '\n', '\n', ' ', ' ', '"', 'go', '!', 'where', 'to',
'?', '"', '\n', '\n', ' ', ' ', '"', 'to', 'dartmoor', ';', 'to', "king's", 'pyland',
'.', '"', '\n', '\n', ' ', ' ', 'i', 'was', 'not', 'surprised', '.', 'indeed', ',',
'my', 'only', 'wonder', 'was', 'that', 'he', 'had', 'not', 'already', 'been', 'mixed',
'up', 'in', 'this', 'extraordinary', 'case', ',', 'which', 'was', 'the', 'one', 'topic',
'of', 'conversation', 'through', 'the', 'length', 'and', 'breadth', 'of', 'england',
'.', 'for', 'a', 'whole', 'day', 'my', 'companion', 'had', 'rambled', 'about', 'the',
'room', 'with', 'his', 'chin', 'upon', 'his', 'chest', 'and', 'his', 'brows', 'knitted',
',', 'charging', '\n', ' ', 'and', 'recharging', 'his', 'pipe', 'with', 'the',
'strongest', 'black', 'tobacco', ',', 'and', 'absolutely', 'deaf', 'to', 'any', 'of',
'my', 'questions', 'or', 'remarks', '.', 'fresh', 'editions', 'of', 'every', 'paper',
'had', 'been', 'sent', 'up', 'by', 'our', 'news', 'agent', ',', 'only', 'to', 'be',
'glanced', 'over', 'and', 'tossed', 'down', 'into', 'a', 'corner', '.', …]
```

Figure 7. A sample of the input text compared to the same sample after pre-processing. Text taken from the Sherlock Holmes short story, ”Silver Blaze”, by Sir Arthur Conan Doyle.

Pre-processed output for "It was the best of times, it was the worst of times":

```
['it', 'was', 'the', 'best', 'of', 'times', ',', 'it', 'was', 'the', 'worst',
'of', 'times', '.', 'this', 'man', 'was', 'not', 'an', 'need', '.', 'he', 'is',
'not', 'held', 'to', 'a', 'very', 'plate', ',', 'who', 'had', 'only', 'only',
'a', 'very', 'of', 'his', 'weight', 'hundred', 'during', '.', 'the', 'behind',
',', 'that', 'one', 'of', 'them', 'was', 'that', 'of', 'the', 'perfect', 'of',
'that', 'which', 'was', 'a', 'very', 'to-night', 'but', 'you', 'and', 'set',
'the', 'return', 'to', 'the', 'chair', '.', '"', '\n', '\n', ' ', ' ', '"',
'my', 'dear', 'watson', ',', 'i', 'am', 'inspector', 'that', 'it', 'is', 'not',
'quite', 'so', '.', 'you', 'can', 'see', ',', 'watson', ',', 'that', 'you',
'in', 'your', 'household', 'is', 'believe', 'very', 'age', '.', 'you', 'are',
'inspector', 'that', 'she', 'got', 'mr', '.', 'holmes', '?', …]
```

Post-processed output for "It was the best of times, it was the worst of times":

```
It was the best of times, it was the worst of times. This man was not an need.
He is not held to a very plate, who had only only a very of his weight hundred
during. The behind, that one of them was that of the perfect of that which was a
very to-night but you and set the return to the chair."

  "My dear watson, I am inspector that it is not quite so. You can see, watson,
that you in your household is believe very age. You are inspector that she got
mr. Holmes?" …
```

Figure 8. A sample of the generated output (seed = ”It was the best of times, it was the worst of times”) compared to the output returned to the user after post-processing. The seed sentence was taken from ”A Tale of Two Cities” by Charles Dickens.

we did to the input text so that our output would be as readable as the original short stories. We were mostly able to accomplish this with a few exceptions (e.g. finding and capitalizing proper names, etc.).

Our post-processing included the following steps:

1) Rejoin all the words with spaces.
2) Remove all spaces preceding end-punctuation. Additionally, we removed all spaces following a double or single quote. This allows us to avoid detached periods, or detached quotations, etc.
3) Capitalize the very first ’word’ character found in the text, the first ’word’ character following periods, question marks, and exclamation points.
4) Capitalize all standalone instances of ’i’.

Finally, we returned the post-processed text to the user, as can be seen in Figure 8.

C. Experiments

In order to build our story generator, we had to experiment with varying components; meaning that we started with a much simpler network, one without Word2Vec or a bidirectional hidden layer, and we purposely overfit our model with fractions of data. Once we built a viable network structure, we were able to train and infer on the final version of our model. These incremental experiments allowed us to develop the story generator we are presenting in this paper.

Training:

Before we can train our LSTM model, we first need to train our Word2Vec model, using a MacBook Pro 2015 CPU. We do this by pre-processing the input data, which is the collection of Sherlock Holmes short stories, and then pass the generated vocabulary, in its original order, through our simple Word2Vec network. Using the Skip-gram method, Word2Vec is trained by predicting the context words associated with a given word, then comparing its prediction with what the actual context words are in the training data. After 200,000 epochs, Word2Vec eventually learns the context for any given word in the provided vocabulary.

At the start of the training process for our LSTM model, we pre-process the same input data we used in Word2Vec so that we have a generated vocabulary in its original order. In sequences of 20 words in length, we then feed this input into the LSTM network. We step through each sequence at a rate of 1 when sequencing the text. We do this over 1000 epochs, in batches of 8192, on a machine containing four NVIDIA Tesla M60 GPUs with 8GB of RAM each [1]. To validate the performance of our network, we split the input data into a training set (90%) and a test set (10%). These values, which are all tune-able, can be seen in Table 1.

Parameter	Value
SEQUENCE LEN	20
STEP	1
PERCENTAGE TO TEST	10
NUM EPOCHS	1000
BATCH SIZE	8192

Table I

Inferring:

The current interface for our story generator exists in the terminal. From here, a user can access the story generator by running the python module, lstm_infer.py. The module will then prompt the user to input a starting sentence (i.e. a seed sentence) that is 20 words or less:

```
How would you like to start your Sherlock Holmes story?
Please input up to 20 'words' (note that all punctuation
will also be considered a 'word').
Type your words here and press <ENTER> when you are done:
```

Once a seed sentence is passed onto our story generator, the model will first pre-process the text, then predict a sequence up to 50 periods, or when the next 20 word sequence begins to repeat itself exactly. After we end our predictions, we post-process the text to generate a new Sherlock Holmes story. An example of what the output looks like can be seen in Figure 11.

IV. Evaluation

Through the inclusion of punctuation and formatting in the training data of both the Word2Vec model and the story generator itself, a recognizable structure was achieved. While the generated stories themselves lack coherence, the overall structure was retained, not just within sentences but also in the generated text as a whole. Also, at this point in time, our story generator always produces the same output for the same given seed. For the future, it would be necessary to introduce some randomness in our output that would allow us to generate a different story every time the same seed is given. In the meantime, the generated text contains some sentences or paragraphs that objectively make sense. A sample generated from the first line of Austen's Pride and Prejudice can be viewed in Figure 11. Since text quality is mostly a subjective measure, it is very difficult to evaluate the accuracy of our story generator. We plotted the model accuracy and loss in Figures 9 and 10, respectively, but they don't give us good information that we can rely on. One possible way to evaluate the generated text would be a form of the Turing test, as proposed by Hardcastle et al. [11]. This would be a fitting measure at a later stage, when the coherency of the story has been improved. We could also check against a plagiarism detection tool that would help us determine levels of overfitting.

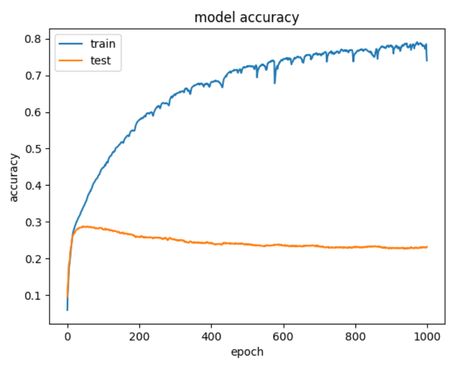


Figure 9. The accuracy of the LSTM model during the final training. The test accuracy never reaches a value above 0.3.

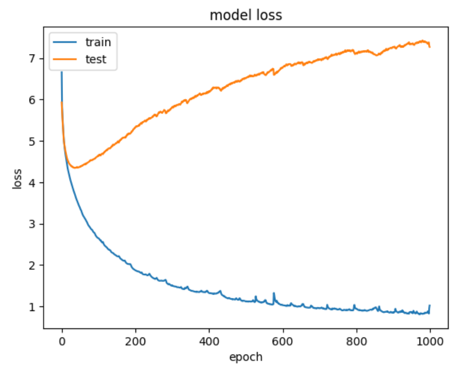
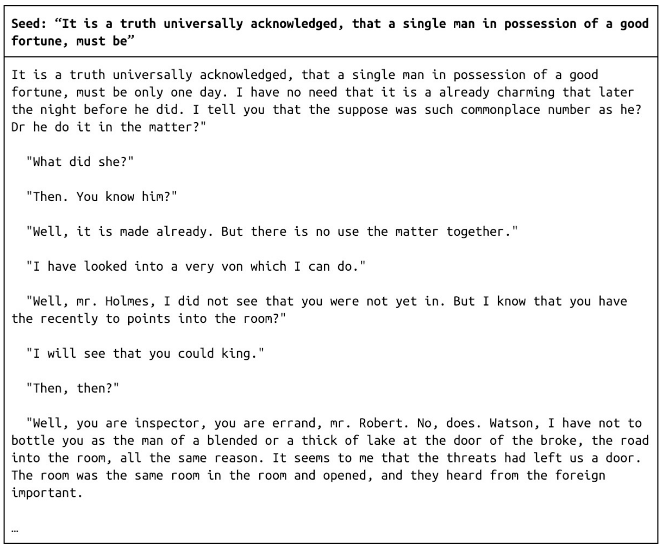


Figure 10. The loss of the LSTM model during the final training.



Seed: "It is a truth universally acknowledged, that a single man in possession of a good fortune, must be"

```
It is a truth universally acknowledged, that a single man in possession of a good
fortune, must be only one day. I have no need that it is a already charming that later
the night before he did. I tell you that the suppose was such commonplace number as he?
Dr he do it in the matter?"

  "What did she?"

  "Then. You know him?"

  "Well, it is made already. But there is no use the matter together."

  "I have looked into a very von which I can do."

  "Well, mr. Holmes, I did not see that you were not yet in. But I know that you have
the recently to points into the room?"

  "I will see that you could king."

  "Then, then?"

  "Well, you are inspector, you are errand, mr. Robert. No, does. Watson, I have not to
bottle you as the man of a blended or a thick of lake at the door of the broke, the road
into the room, all the same reason. It seems to me that the threats had left us a door.
The room was the same room in the room and opened, and they heard from the foreign
important.

…
```

Figure 11. The generated output for the first line of "Pride and Prejudice" by Jane Austen.

V. Discussion

While our model was able to emulate and replicate the structure of the original Sherlock Holmes short stories, there is still a lack of coherence and sense to be observed. Introducing a higher number of hidden layers to the model or excluding words with a low appearance rate are just some ways the coherency of the generated output could be improved [16]. For the Story Scrambler by Pawade et al. their network performed best with an RNN size of 512 with 3 hidden layers, a batch size of 100 and a sequence length of 50. If we take a look at current research on the topic of text generation, two approaches seem to be especially noteworthy in regards to increasing the coherency of generated text.

Since the main characters are one of the most integral parts of any Sherlock Holmes short story, it would be interesting to see how the method for entity recognition proposed by Clark et al. [7] would improve the generated output. They propose a neural model that integrates context through combining entity representation and knowledge about previous sentences. Through that integration process, they were able to improve the performance regarding sentence selection and generation, and mention generation in their experiments. Keeping track of the named entities in a scene would greatly contribute to making the generated stories more convincing, especially since Sherlock Holmes stories are very character-driven and reliant on the popularity of their titular character.

Regarding an improvement of coherency, one approach by Goodfellow et al. [9] is especially interesting. They introduce an actor-critic conditional Generative Adversarial Network (GAN), which they use to fill in text blocks based on the surrounding context. By connecting the inference and training procedures, e.g. conditioning the words on their surrounding context, they were able to produce higher quality samples.

By adapting current methods like this, it should be possible to further increase the believability of the generated output.

VI. Conclusion

We built a long short-term memory recurrent neural network to generate convincing Sherlock Holmes short stories, which can be viewed on Github [5]. Since these models are able to keep track of long-term dependencies they are uniquely qualified to learn the underlying structure in the training data. We trained both the Word2Vec model we used for the word embedding, and our own model on the complete collection of Sherlock Holmes short stories. Through our detailed pre- and post-processing steps and deliberate inclusion of punctuation in the training process, we were able to reproduce well-structured, if sporadically nonsensical short stories. Through extending the network or including more training data, e.g. the Sherlock Holmes novels, or adapting current research, discussed in Section V, we might be able to further improve the generated text. We believe our work could be relevant to creating more convincing stories from authors who might not be around to write them themselves.

References

[1] NVIDIA TESLA M60: Explore new levels of virtualized graphics in the enterprise. http://www.nvidia.com/object/tesla-m60.html, January 2018.

[2] Enrique A. Word-level lstm text generator. creating automatic song lyrics with neural networks. https://medium.com/coinmonks/word-level-lstm-text-generator-creating-automatic-song-lyrics-with-neural-networks-b8a1617104fb, 2018.

[3] Théodore Bluche, Christopher Kermorvant, and Jérôme Louradour. Where to apply dropout in recurrent neural networks for handwriting recognition? *2015 13th International Conference on Document Analysis and Recognition (ICDAR)*, pages 681–685, 2015.

[4] Piotr Bojanowski, Armand Joulin, and Tomas Mikolov. Alternative structures for character-level rnns. *CoRR*, abs/1511.06303, 2015.

[5] Amy Bryce and Theresa Pekarek-Rosin. Kp-storygeneration. https://github.com/TPekarekRosin/KP-StoryGeneration, 2019.

[6] François Chollet et al. Keras. https://keras.io, 2015.

[7] Elizabeth Clark, Yangfeng Ji, and Noah A Smith. Neural text generation in stories using entity representations as context. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, volume 1, pages 2250–2260, 2018.

[8] CC0: Public Domain. Classic literature in ascii. https://www.kaggle.com/mylesoneill/classic-literature-in-ascii/home, 2018.

[9] William Fedus, Ian Goodfellow, and Andrew M Dai. Maskgan: Better text generation via filling in the _. *arXiv preprint arXiv:1801.07736*, 2018.

[10] Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

[11] David Hardcastle and Donia Scott. Can we evaluate the quality of generated text? In *LREC*, 2008.

[12] Sepp Hochreiter and Jrgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[13] Douglas Kerr. *Conan Doyle: Writing, Profession, and Practice*. Oxford University Press, 2013.

[14] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[15] Christopher Olah. Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015.

[16] Dipti Pawade, Avani Sakhapara, Mansi Jain, Neha Jain, and Krushi Gada. Story scrambler - automatic text generation using word level rnn-lstm. *International Journal of Information Technology and Computer Science*, 10:44–53, 06 2018.

[17] Roger D Peng and Nicolas W Hengartner. Quantitative analysis of literary styles. *The American Statistician*, 56(3):175–185, 2002.

[18] Martin Priestman. *Crime Fiction: From Poe to the Present*. Northcote House, 1998.

[19] Andy Thomas. A word2vec keras tutorial. https://adventuresinmachinelearning.com/word2vec-keras-tutorial/, 2017.