Generating Sherlock Holmes Passages Using Character-Level Recurrent Neural Networks

Yash Potdar

23 March 2023

Abstract

In this project, a character-level recurrent neural network (RNN) is used to predict text based on a corpus of Sherlock Holmes. This is an example of a many-to-many RNN application, as there is a sequential input that is treated like multiple inputs and the outputs are generated sequentially. In the project, hyperparameters such as *n_epochs*, *hidden_size*, *n_layers*, *lr*, *temperature*, and *chunk_len* are tuned in order to make an RNN that performs better than the Vanilla RNN. From the results, it is apparent that the hyperparameters that improve the generation of sequential data the most may be the hidden nodes and layers within the GRU, the temperature, and the size of the chunks.

1 Introduction

This project investigates the use of a character-level RNN in order to generate a passage of Sherlock Holmes that is 250 characters long when trained on a small passage of text from a corpus of Sherlock Holmes stories. An RNN is a neural network architecture that is specialized to handle sequential data and generate an output sequence that represents the input well. One key in the RNN architecture is the hidden state, which is where the output of each step will update and feed information to. As the RNN works with the input one step, or iteration, at a time, the hidden state is updated in a feedback loop sort of way after each step, hence the name "recurrent". The hidden state essentially serves as the RNN's memory and is how the network's understanding of the syntax and rules grows.

The overarching goal of this project is to understand the hyperparameters and structure within an RNN and attempt to generate a passage of Sherlock Holmes that is logical. This project can potentially have a strong impact because an RNN that is well-trained and understands the context behind an input text can allow us to have an assisted writing tool that could take a passage we have written and augment it.

2 Data Description

The dataset is a text file containing the complete canon of Sherlock Holmes by Arthur Conan Doyle [1]. One consideration worth pointing out is the corpus consists of 56 short stories and four novels.

This means that the text is not fully connected as there are multiple stories. However, since we will be training on a relatively small number of characters than compared to the total number of characters in the text file (3,381,982 characters), the impact of having multiple stories, not just one, is negligible. If a randomly selected chunk spans multiple stories, there is still value gained from this passage. The RNN can learn from the syntactic structure of the works, as well as the grammar and mannerisms of the author, although the text generated may not make sense in context. Since this is a large dataset, it would be impossible to train on it as a whole, so I will use chunking in this project.

3 Methods

3.1 RNN Overview

RNNs differ from traditional feedforward neural networks such as convolutional neural networks (CNNs) in that RNNs have a hidden state, which allows them to gradually gain a context of the sequential data fed into the network. CNNs do not possess the ability to have a memory of previous inputs and thus are not effective in identifying trends in a sequence, like a series of words, images, or even time series data.

There are many applications of RNNs and varieties depending on the inputs and outputs [2].

- **One to one**: image classification
- **One to many**: image captioning - creating a caption that describes the scene in an image
- **Many to one**: sentiment analysis - summarizing the feelings present in a passage of text
- **Many to many**: video captioning - creating a caption that describes the events in a video

One common thread among all these real-world applications of RNNs is they all need to contextualize the scene. In cases where there are multiple sequences of input ("many"), there is an additional challenge of not only understanding each individual portion, but understanding how the sequences relate to one another.

There are a few limitations of RNNs, as explained in Andrej Karpathy's article *The Unreasonable Effectiveness of Recurrrent Neural Networks*. Vanilla, or basic RNNs without many changes to the boxed architecture, do not generalize in the right way. Although there are promising cases like an RNN generating Linux code and understanding syntax, code flow, and indentation when given the corpus of Linux code, there may not always be correct generalizations in terms of context [3]. Another limitation of RNNs is their subpar performance when generating output sequences that are very long [4]. The memory, or hidden states, may get progressively worse throughout the training process since training occurs one step at a time. There is an approach called Long Short-Term Memory (LSTM), which is a variation of the RNN architecture that controls vanishing gradients, resulting in better outputs for longer passages. In this paper, I did not implement LSTM because I was generating a passage of 250 characters. If I were to generate passages of a few thousand characters, I would experiment with LSTM.

3.2 RNN Implementation Explanation

This section will provide an explanation of the RNN implementation, which was made using a Practical PyTorch tutorial [5]. The accompanying code contains an adapted version of the tutorial since this project focuses on the Sherlock Holmes dataset, and I will be experimenting with several architectures. The RNN will be generating a string one character at a time, so *all_characters* is a list of all 100 possible characters that can be predicted. The *char_tensor* function takes a string and tokenizes it based on the indices the characters have in *all_characters*. Therefore, if a sentence has 15 characters, *char_tensor* will return a Tensor of length 15 that has the indices of the characters within the sentence.

The *random_chunk* function serves to randomly select an individual chunk of text of length *chunk_len*. It does this by selecting a random starting index and slicing a user-defined length of text. The *random_training_set* function serves to take a random chunk of text and return the input and target for a randomly selected chunk. The input is all the characters except the last, while the target is all the characters except the first, since the RNN will be predicting the next character given the current character. The RNN class contains three layers: one linear embedding layer that encodes the inputs, one gated recurrent unit (GRU) layer which has a user-defined number of hidden layers, and one decoder linear layer that outputs the probability distribution which will influence the next character generated. The forward method takes the input and the current hidden layer as parameters and puts them through the aforementioned layers, and then returns the output and the updated hidden layer. The *init_hidden* method will initialize the hidden layer with a user-defined number of layers.

The train function takes the input and target outputted from the *random_training_set* function. It initializes the hidden layer, clears old gradients from the last step, and feeds the input one character at a time into the RNN. At each step, the forward method is called, and the output is generated along with the updated hidden, or memory, layer. The cross-entropy loss between the output and the true target value is also calculated and accumulates throughout the training chunk. Next, the derivative is computed using backpropagation and Adam optimizer with a user-defined learning rate improves the parameter optimization based on these gradients from the backpropagation.

The evaluate function takes the trained model and uses a priming string to build up a hidden state. It then iterates the number of characters to predict and repeatedly follows this process: (1) use the forward method to get an output distribution and updated the hidden layer, (2) divide the distribution by the temperature hyperparameter and take the exponential, (3) selects one character from the distribution, and (4) updates the predicted string, which is eventually returned.

Finally, the model is initialized and trained for a user-defined number of epochs. The losses are stored and a sample output is printed every hundredth epoch.

3.3 Hyperparameter Tuning

The hyperparameters of interest in this experiment were n_epochs, $hidden\_size$, n_layers, lr, $temperature$, and $chunk\_len$. These hyperparameters are explained below:

- **n_epochs**: This is the number of epochs the model is trained for. A larger number will lead to longer training times but potentially lower training loss, until it hits a plateau.
- **hidden_size**: This is the number of hidden nodes on each hidden layer within the GRU. A higher number results in higher model complexity.
- **n_layers**: This is the number of hidden layers within the GRU. A higher number results in higher model complexity.
- **lr**: This is the learning rate, or the pace at which the model performs its gradient updates during the gradient descent algorithm.
- **temperature**: This is the number that the output distribution is divided by. A number greater than one makes outcomes more equally likely, introducing more randomness. A smaller number will introduce less randomness and weight higher probabilities in the output distribution.
- **chunk_len**: This determines the size of the training set. A higher number will make the training process longer since it adds steps for the training process.

3.4 Architecture Comparison

Table 1 summarizes the hyperparameters selected for each model. Deviations from the Vanilla model are bolded.

	Vanilla	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6	Iteration 7
n_epochs	2000	2000	**4000**	**3000**	2000	2000	2000	2000
hidden_size	100	100	100	100	**200**	100	100	100
n_layers	1	**3**	**3**	**3**	**3**	**2**	**2**	**3**
lr	0.005	**0.001**	**0.001**	0.005	0.005	0.005	0.005	0.005
temperature	0.8	0.8	0.8	0.8	0.8	**0.5**	**1.5**	**0.5**
chunk_len	200	200	200	200	200	200	200	**500**

Table 1: Hyperparameter Selections for Each Model

4 Results

The training loss curves of the various models are displayed in Figure 1:

Table 2 displays the last training loss achieved by each model.

	Vanilla	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6	Iteration 7
Last training loss	1.593	1.646	**1.528**	1.621	1.795	1.634	1.686	1.507

Table 2: Last Training Losses for Each Model

Table 3 displays the generated output of length 250 characters for each model.

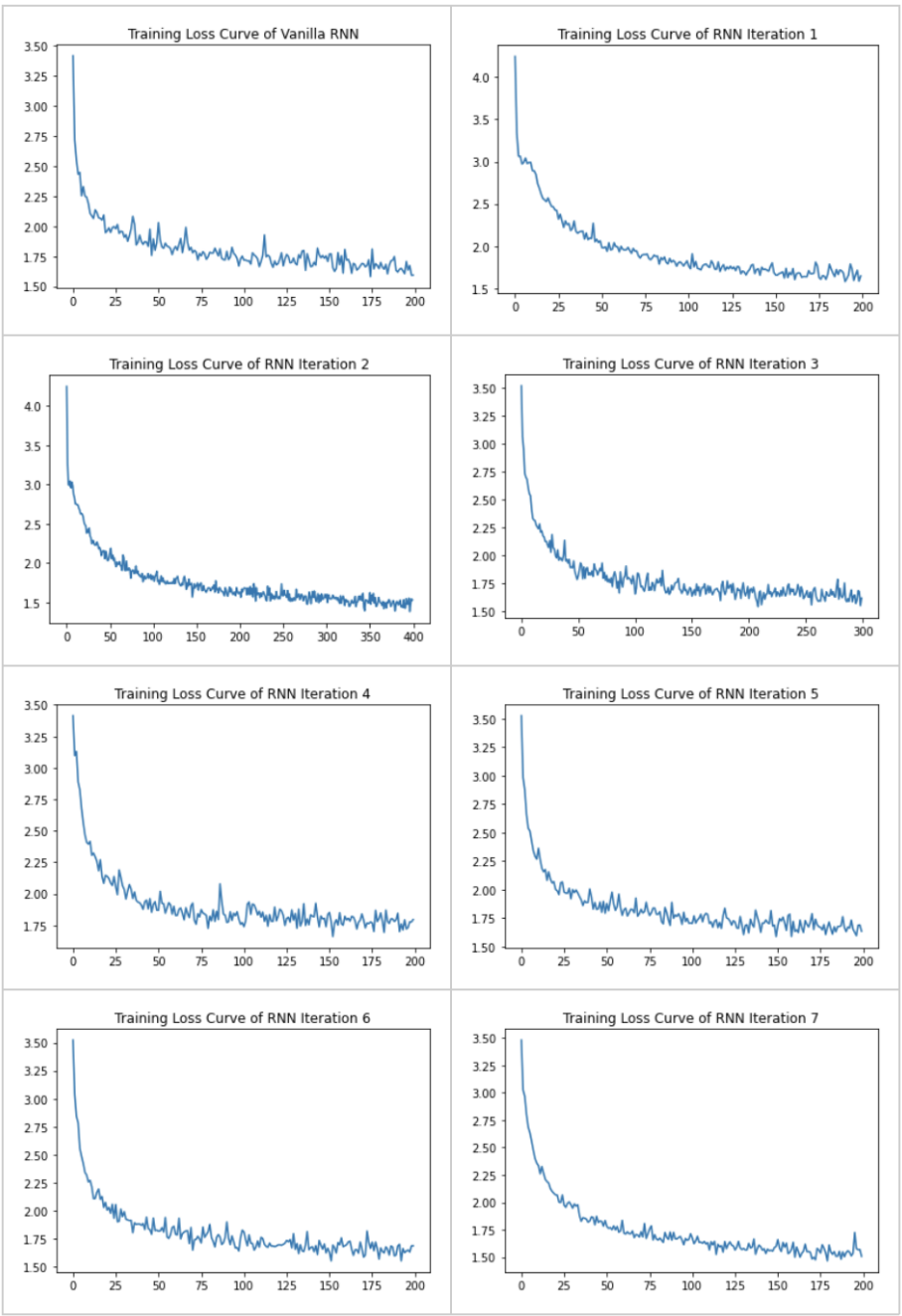

Figure 1: Loss Curves for Each Model

5 Discussion

To evaluate the models, I used a combination of their loss achieved at the end of training and how natural the sentence sounded. It is important to note that most of the models generated outputs that sounded more natural than the Vanilla model. This reinforces that the Vanilla model is not

Model	Output
Vanilla	When his towl face which has bys we was thide it?" "There my lookered, sharl Serlact the experspy ocrelushinalon us this trange which a coulled we used for from in non accust victle a face tach as wetter said couch as a face at is accare wo
Iteration 1	Where to the notblut furel my should, when the dearom and the fain syrustleet. "Which crise hy midds have disy his some. The fertion diver my the that our the questioned amred that the cornescasent have my wist not the Mader with reshest of
Iteration 2	Whis nturned made. I have at me of the trout nercy her that and had seemed the firning a me a mibch as low is been changable to amention. The despersation. "I efcear Gery behind too the very trace which seeme, and had over the curned
Iteration 3	Whe the poim tell have had to a sev in it sever had procame Hirried, it as path lear, the pass of state my ponistles some a very my reemipaon which he with however cally on aldectel as I have smatietions. And sever well be officied your
Iteration 4	Whe no denty, I ward the reat in which which May you leady, I cant was facter, wortone face not a dund whore his a whis yardemer, intiso I pan cailf wake never his have to was it word, you no was you and goor, sor. I mugented frong, and "pant
Iteration 5	Whixt. We the Carning that I said so pressernied frous well wan gurst facight–a moat of the and the was. Then this is shad a clided when I had that it it is some that the do band the had away bet-of the was after what eardy slecth a face of you
Iteration 6	Whif, and voight a gave, Mr you that, who cauld for for the would the wexcudland but of this caul is the proped. The reabe to can ladgle for a scards you, and the sallock was and than mown all to the let that be back more pow and howe s
Iteration 7	What I have cost and in the moxed you come with oblesky clinaly, lye firned, "we could you mony of is Colution, and by to the scamed. "The thill could looking that cuchook. It a may somp head the friar trose to street, and at hunds a great i

Table 3: Generated Outputs of 250 Characters for Each Model

very effective, especially for RNNs and should be tuned. Iteration 2, which trained on 4000 epochs, added 3 hidden layers, and used a small learning rate of 0.001 had the smallest last loss. However, this did not yield the most coherent message. There is some structure visible with the dialogue and the indents indicating new paragraphs, however, there is not much grammatical sense.

Models 4 and 7 generated the most coherent outputs. Model 4 had a higher complexity as it used 200 hidden nodes per layer and 3 hidden layers. Model 7 also had 3 hidden layers, and although it had only 100 hidden nodes per layer, the difference most likely came because it used a chunk length of 500. This meant that it was training on a larger train set. Its *temperature* was also 0.5, which means the higher probabilities from the output distribution have a higher chance of being selected. This could be a key player in creating structure for the output.

6 Conclusion

These results showed that the Vanilla model is not effective at generating text. The default hyperparameters may not be sufficient for contextualizing the input and understanding the structure within the input. From our results, we can conclude that the hyperparameters that improve the generation of sequential data the most may be the hidden nodes and layers within the GRU, the temperature, and the size of the chunks. It is definitely important to note that these are initial findings, but they are promising because the potential of creating an RNN that generates text accurately may help increase productivity in the future.

7 References

[1] *The Complete Sherlock Holmes.* The complete Sherlock Holmes. (n.d.). Retrieved March 23, 2023, from https://sherlock-holm.es/stories/plain-text/cnus.txt

[2] Li, F.-F. (n.d.). *Lecture 10 — Recurrent Neural Networks. Stanford University CS 231N.*

[3] Karpathy, A. (n.d.). *The Unreasonable Effectiveness of Recurrent Neural Networks.* The unreasonable effectiveness of recurrent neural networks. Retrieved March 23, 2023, from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

[4] Tu, Z. (n.d.). *Recurrent Neural Networks. UC San Diego COGS 181.*

[5] *Practical PyTorch: Generating Shakespeare with a character-level RNN.* — notebook.community. (n.d.). Retrieved March 23, 2023, from https://notebook.community/spro/practical-pytorch/char-rnn-generation/char-rnn-generation