Generating Tv-Script using

Recurrent Neural Network

INSE 6421

Project

Concordia University

Submitted By:

Meghaksh Brahmbhatt | 40166917

Binit Kumar | 40172005

April, 14th 2022

Table of Contents

[What is TV Script Generation Project? 3](#_Toc100754036)

[Expected Input Script and output Script 3](#_Toc100754037)

[Why using Recurrent Neural Network? 3](#_Toc100754038)

[Problem with RNN Architecture 5](#_Toc100754039)

[Vanishing or Exploding Gradients 6](#_Toc100754040)

[RNN Cell 6](#_Toc100754041)

[LSTM Cell 7](#_Toc100754042)

[Forget Gate 7](#_Toc100754043)

[Update Gate 8](#_Toc100754044)

[Cell state to hidden output 8](#_Toc100754045)

[Putting it all together 8](#_Toc100754046)

[How to fix gradient problem? 9](#_Toc100754047)

[Framework/Library Used 9](#_Toc100754048)

[Pre-processing of Input data 10](#_Toc100754049)

[Creating batches of data 11](#_Toc100754050)

[How data-loader looks like 12](#_Toc100754051)

[Neural Network Architecture 12](#_Toc100754052)

[Constructor 12](#_Toc100754053)

[Forward Propagation 13](#_Toc100754054)

[Initializing the hidden state of LSTM 13](#_Toc100754055)

[Backpropagation 13](#_Toc100754056)

[Training loop 15](#_Toc100754057)

[Hyperparameters 16](#_Toc100754058)

[Training Result 17](#_Toc100754059)

[Script Generation 18](#_Toc100754060)

[References 18](#_Toc100754061)

# What is TV Script Generation Project?

A project done by us to generate a fake Tv-script using a RNN model. We are using a Tv script from a popular Tv show named Seinfeld. The neural network takes the following mentioned tv-script, learns the text sequences from it to generate a new fake tv-script.

* Our code repository location: <https://github.com/binit92/INSE-6421>
* Seinfeld dataset from season 9. Dataset source: [https://www.kaggle.com/thec03u5/seinfeld-chronicles#scripts.csv](https://www.kaggle.com/thec03u5/seinfeld-chronicles)
* Seinfeld: <https://en.wikipedia.org/wiki/Seinfeld>

## Expected Input Script and output Script

|  |  |
| --- | --- |
| Input Sample | Output Sample |
| george: so, i was in the contrary.  george: so, i guess i was a woman, and the defendants was a good boy.  elaine: i don't know where it is.  jerry: so i was thinking about this one?  hoyt: i thought i was a little adjustment.  jerry: what is that?  hoyt: yes, yes. yes. i got a pee on this. you were in the middle of a plane, and i have a little adjustment.  george: what happened to him? | george: so you want to see how you could do that?  hoyt: i don't want to see you in a hotel.  elaine: you want to get the car on the street and a wheelchair?  hoyt: what do you think?  george: yeah, i guess i was wondering about it.  jerry: what are you doing with this girl?  elaine: no, i got to tell him.  george: i can't believe this is the most exciting thing. |

# Why using Recurrent Neural Network?

Our script generations take a word and determine the next word in a sentence. This requires keeping a sequence or order of words in the neural network. However, In Feed-Forward Network, there is no sense of order in the input. So, the question is how to build the concept of memory in the neural network so that it can learning for the sequence.

Recurrence relations need to apply at each time step t, such that model learns the concepts of memory by updating the hidden state.

|  |  |
| --- | --- |
| Diagram  Description automatically generated | Text  Description automatically generated with medium confidence |
| Visualizing Character-wise RNN | |
| Our goal here is to predict the next character in word “steep”   * When we pass “s”, desired output is “t”. * When we pass “t”, desired output is “e”. * When we pass “e”, desired output could be “e” or “p”. The network doesn’t have enough information to determine which character to predict!   To solve this problem, we need to include information about the sequence of characters. |  |
| We can solve this problem by routing the hidden layer output from the previous step back into the hidden layer.   * The box in the diagram means the value from the previous sequence, or time step. * Now the network sees an “e”, it knows it saw an “s” and a “t” before, so the next character should be another “e”   This architecture is known as Recurrent Neural Network or RNN   * Now the total input in the hidden layer is the sum of the layered combinations from the input layer and previous hidden layer |  |
| We can view our recurrent network as one big graph by unrolling it.   * Now, we have a feed-forward network for each character but connected through the hidden layers. * Each hidden nodes receives inputs from input node and hidden node from the previous step |  |
| Let’s visualize by adding some numbers here   * Here, we’re **one hot encoding** the input characters. * 1000 = “s”, 0100 = “t”, “0010” = “e”, 0010 = “e” * There are three units in the hidden layer and the output layer is showing the logits * We pass the logits into Softmax function to get prediction and to train with a cross entropy loss.   This is basic architecture for Character-Wise RNN. |  |
|  |  |
|  |  |

# Problem with RNN Architecture

We can include information from a sequence of data using a recurrent connection on the hidden layer. This connection goes through these weights, *Whh* After enrolling the network, we say the hidden layer at step t is a function of the previous hidden state multiplied by those weights. The output of that layer is again multiplied by *Whh*. For every step we have in the network, we are multiplying by the weights again and again.

And when we do backpropagation, that’s even more multiplication. These leads to problem where gradients going through network either get really small and vanish or get really large and explode.

Diagram

Description automatically generated

# Vanishing or Exploding Gradients

If we multiply by some number a bunch of times, we will get two results except a couple of special cases. If that number is less than 1, we will end up at 0. If it greater than 1, we will head towards infinity. This happens to gradient in normal RNN, where they either vanish or explode.

Resulting in making it difficult for RNNs to learn long range interactions.

Diagram

Description automatically generated

# RNN Cell

We can think of RNNs as a bunch of cells with inputs and outputs. Inside the cell, we have network layers, such as the sigmoid layer labelled with a sigma here. To solve the problem of the vanishing gradients, we can use more complicated cells called long short-term memory or LSTM

Diagram, schematic

Description automatically generated

# LSTM Cell

Let’s break down LSTM to understand:

The key addition here is the cell state labelled C. In this cell, there are four network layers shown as yellow boxes. Each of them with their own weights. The layers labelled with sigma are sigmoid and tanh is the hyperbolic tangent function. Tanh is similar to a sigmoid in that it squashes input, but the output is between -1 to 1 instead of 0 and 1

The red circles are point-wise or element-wise operations i.e. they operate on matrices element by element. The main improvement here is through the cell state. The cell state goes through LSTM cell with little interaction allowing information to flow easily through the cells. The cell state is modified only through element-wise operation which functions as gates.

And the hidden state is now calculated through cell state, then passed on

Diagram

Description automatically generated

## Forget Gate

The first gate is the forget gate. The values coming out of sigmoid layer are between 0 and 1. Then they are multiplied element-wise with the cell state. So the values from this layer close to 0 will shut off certain elements in the cell state. Effectively, forgetting that information going forward. Conversely, values close to 1 will allow information to pass through unchanged. This is helpful, because the network can learn to forget information that causes incorrect predictions. On the other hand, long range information that are helpful is allowed to flow through freely.

A screenshot of a computer

Description automatically generated with medium confidence

## Update Gate

The next gate updates the cell state from the input and previous hidden state. The tanh layer output is added to the cell state and again gated by a sigmoid layer. In this way, the cell state can be updated in the step and passed along to the next cell

Diagram

Description automatically generated

## Cell state to hidden output

Here, the cell state is used to produce the hidden state which is sent to the next hidden cell as well as to higher layers. It’s the arrow pointing up here

The cell state is passed through another tanh then gated again with another sigmoid layer. All these sigmoid gates let the network learn which information to keep and which information to get rid of.

A screenshot of a computer

Description automatically generated with low confidence

## Putting it all together

Putting all this together, the LSTM cell consists of a cell state with a bunch of gates used to update it, and leak it out to the hidden state.

This is just the basic LSTM. There are multiple variations and lot of ongoing experimentation into improving these. They are also stacked into deeper layer. We just send the output from one cell to the input of another

Diagram

Description automatically generated

# How to fix gradient problem?

Since the cell state is allowed to flow through the hidden layers with only this linear sum operation. Gradient can easily move through the network without being diminished. We can also get gradients added into the network through the LSTM cells but they are just added to the gradients flowing through

* Math behind LSTMs <https://www.youtube.com/watch?v=iX5V1WpxxkY>
* Understanding LSTM Networks <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>

LSTMs are basic unit of RNNs in many applications

Diagram

Description automatically generated

# Framework/Library Used

* **Pytorch** – An open-source machine learning framework that accelerates the path from research prototyping to production deployment. (<https://pytorch.org/>)
* **Numpy** – A library that offers comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms and more. (<https://numpy.org/>)
* **Pickle** – A library to convert Python object hierarchy into byte stream called “pickling”. “unpickling” is the inverse operation where a byte stream is converted back into an object hierarchy. (<https://docs.python.org/3/library/pickle.html>)

# Pre-processing of Input data

Pre-processing is a step before feeding the input to the neural network where the input data is pre-processed to make it suitable for training. This includes

* Changing the entire data set into lowercase
* Splitting the sentences to get all the words
* Creating lookup table to generate word embeddings i.e. transforms the word to integer ids
* *vocab\_to\_int*: dictionary to go from a word to id
* *int\_to\_vocab*: dictionary to go from the id to word

Text

Description automatically generated

The above function takes the tv script as input and return two dictionaries as tuple containing word to id and id to word entries.

**Tokenize Punctuation**

Punctuations like periods and exclamation marks can create multiple ids for the same word. For e.g. bye, bye!

This dictionary will be used to tokenize the symbols and add the delimiter (space) around it. This separates each symbol as its own word, making it easier for the neural network to predict the next word.

Text

Description automatically generated

# Creating batches of data

**batch\_data** function to batch words data into chunks of size **batch\_size** using Pytorch’s DataLoader classes. DataLoader class will help to create **feature\_tensors** and **target\_tensors** of correct size and content of given **sequence\_length** .

E.g. words = [1, 2, 3, 4, 5, 6, 7]

* sequence\_length = 4
* First feature\_tensor would be: [1, 2, 3, 4]
* The corresponding target\_tensor would be: 5
* Second feature\_tensor would be: [2, 3, 4, 5]
* And the second target\_tensor would be: 6

Graphical user interface, text, application, email

Description automatically generated

The **sample\_x** should be of size (batch\_size, sequence\_length) or (10, 5) in this case and **sample\_y** should just have one dimension: batch\_size (10). We can also notice that the target **sample\_y** are the next value in the ordered test\_text data.

# How data-loader looks like

Sample batch of inputs **sample\_x** and targets **sample\_y** from the data loader. We are shuffling the data in the data loader to get random batches.

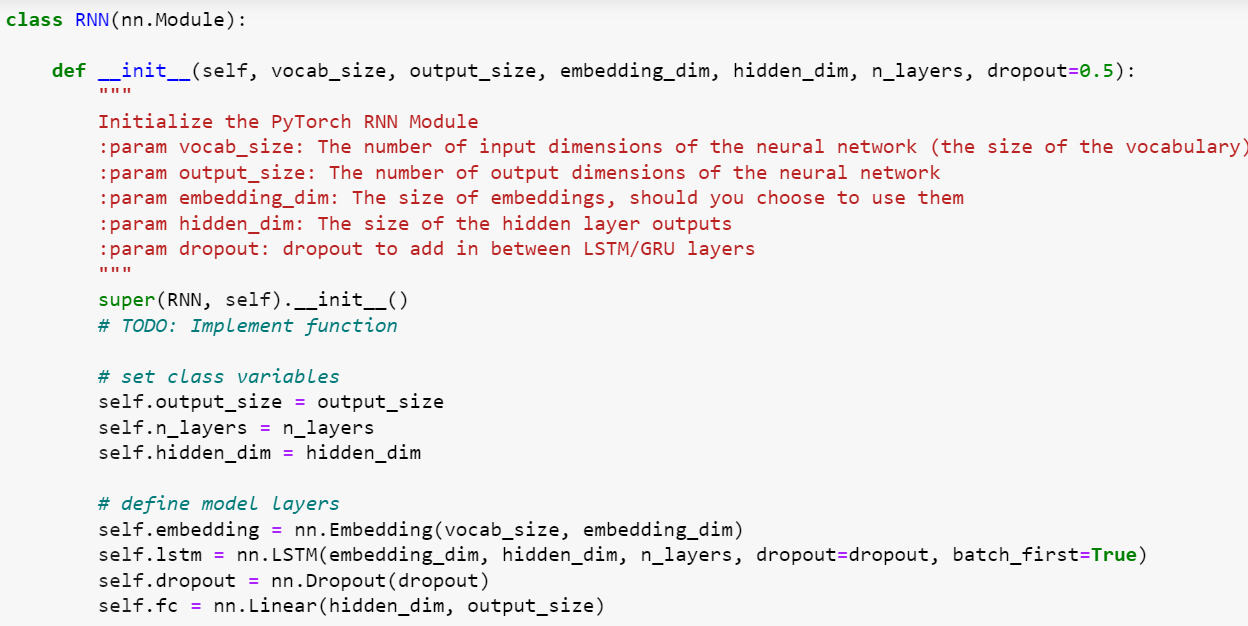
Table

Description automatically generated

# Neural Network Architecture

The below mentioned class RNN inherit the nn.Module class from Pytorch library. \_\_init\_\_ method is the constructor of the class to initialize the Pytorch RNN module with parameters like vocab\_size, output\_size, embedding\_dim, hidden\_dim and dropout.

## Constructor



## Forward Propagation

Forward method in the RNN class defines the forward propagation of the neural network.

Graphical user interface, text, application, email

Description automatically generated

## Initializing the hidden state of LSTM

Graphical user interface, text, email

Description automatically generated

Note: Alternatively, we can also use GRU (i.e. Gated Recurrent Units) as well in place of LSTM. GRU is related to LSTM as both are utilizing different way if gating information to prevent vanishing gradient problem.

* The GRU controls the flow of information like the LSTM unit, but without having to use a **memory unit**. It just exposes the full hidden content without any control.
* GRU is relatively new and performance is similar to LSTM but computationally **more efficient** because of less complex structure

Graphical user interface, text, application

Description automatically generated

# Backpropagation

Use the above RNN class to apply forward and back propagation

This function will be called iteratively, in the training loop

Text

Description automatically generated

# Training loop

This function train the network over all the batches for the number of epochs given.

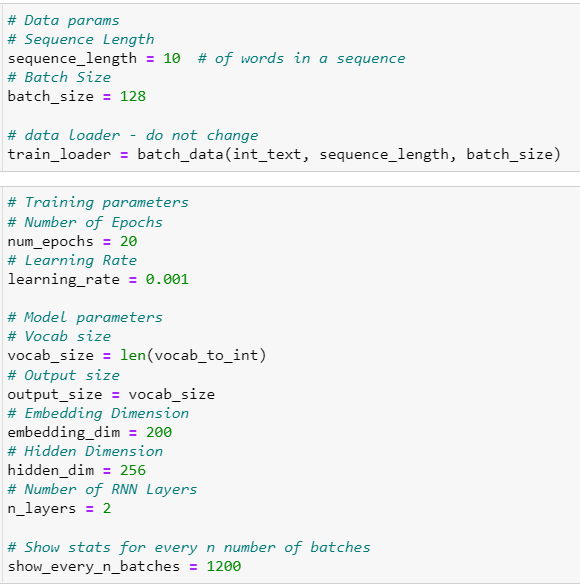
The model progress is printed every number of batches

Graphical user interface, text, application

Description automatically generated

# Hyperparameters

* **Sequence Length**: to set the length of the sequence
* **Batch Size** : to set the size of batches
* **Number of epochs** : to set the number of epochs to train for
* **Learning Rate**: to set the learning rate of Adam Optimizer
* **Vocab Size** : to set the number of unique tokens in vocabulary
* **Output Size** : to set the desired size of output
* **Embedding Dim** : to set the embedding dimension, smaller than vocab\_size
* **Hidden Dim** : to set the hidden dimension of RNN
* **N Layer**: to set the number of layers in the RNN
* **Show N Batches**: to set the number of batches at which neural network should print progress



# Training Result

Before starting training, we are initializing:

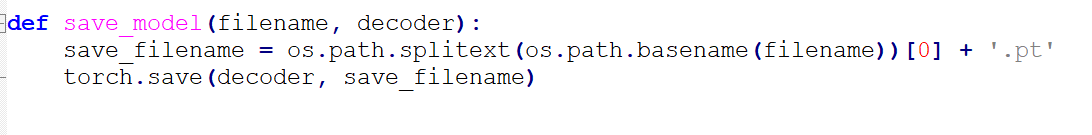
* The optimizer as Adam using Pytorch’s library. Details could be found here: ([https://pytorch.org/docs/stable/generated/torch.optim.Adam.html#torch.optim.Adam](https://pytorch.org/docs/stable/generated/torch.optim.Adam.html%23torch.optim.Adam))
* loss function as Cross Entropy loss function. Details could be found here:

( <https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html>)

Graphical user interface, text, application, email

Description automatically generated

Saving the model as a physical checkpoint file after training is complete.



Graphical user interface, table, Excel

Description automatically generated

Observations

# Script Generation

Graphical user interface, text, application, email

Description automatically generated

# References

https://arxiv.org/pdf/1412.3555v1.pdf