

# School of InfoComm Technology

**Deep Learning Assignment**

Diploma in CSF / FI / IT

Apr 2022 Semester

**ASSIGNMENT 2**

(40% of DL Module)

4th Jul 2022 – 12th Aug 2022

**Submission Deadline:**

**Presentation: 12th Aug 2022 (Week 17),**

**Report: 12th Aug 2022 (Friday), 11:59PM**

|  |  |  |
| --- | --- | --- |
| **Tutorial Group** | **:** | **P01 / P02 / P03 / P04** |
| **Student Name** | **:** | Suen Jeren |
| **Student Number** | **:** | S10204092 |

**Penalty for late submission:**

10% of the marks will be deducted every calendar day after the deadline.

**NO** submission will be accepted after 21st Aug 2022 (Sunday), 11:59PM.

Contents

[Overview 3](#_Toc111047109)

[Data Loading and Processing 4](#_Toc111047110)

[Loading and cleansing data 4](#_Toc111047111)

[Preparing data and one-hot encoding 8](#_Toc111047112)

[Developing the sequence generator models 10](#_Toc111047113)

[Model with GRU 10](#_Toc111047114)

[First model with GRU 10](#_Toc111047115)

[Adding regularizer to first model 13](#_Toc111047116)

[Second model with GRU 15](#_Toc111047117)

[Model with Conv1D 17](#_Toc111047118)

[First model with Conv1D 17](#_Toc111047119)

[Adding regularizers to model 19](#_Toc111047120)

[Second model with Conv1D 21](#_Toc111047121)

[Model with LSTM 23](#_Toc111047122)

[Final Selected Model 25](#_Toc111047123)

[Text generation with developed model 26](#_Toc111047124)

[Observations with different temperatures 26](#_Toc111047125)

[Overall observations from text generation 30](#_Toc111047126)

[Summary and further improvements 31](#_Toc111047127)

[Summary 31](#_Toc111047128)

[Further improvements 32](#_Toc111047129)

# Overview

In part 2 of our assignment, we are to implement a Recurrent Neural Network to create an English language character generator capable of building semi-coherent English sentences from scratch, by building them up character-by-character. We will use J. K. Rowling’s book – Harry Potter and the Philosophers Stone to train our model.

We will train a deep learning model to generate text automatically, character-by-character by showing the model many training examples so that it can learn a pattern between text inputs and potential character outputs.

In this report, we will be going through the data loading process – loading our data into the Jupyter notebook, data processing – removing unnecessary characters from the data, splitting the training text and labels, and performing one-hot encoding, and developing the sequence generator model. We will then try writing words into an input and see if our model can correctly write valid words out. I will be trying and testing out different model architectures, as well as different RNN layers such as GRU, LSTM and Conv1D to find out which of the different RNN layers are best for our model in terms of accuracy and performance.

I will be following the universal machine learning workflow to develop the models, by:

* Starting with a baseline model
* Scaling up the model until it overfits
* Regularize the model accordingly

After applying the universal machine learning workflow to each of our different models, I will then compile the accuracy together and decide on the best model to use for our text generation.

# Data Loading and Processing

## Loading and cleansing data

We first start off by loading our data into the Jupyter notebook.



*Figure 1, loading data into text*

We specified the name of the book into a parameter path and opened the text with the in-built python method “.read()” to read the text file. There was some issue with the file not being able to load in without specifying which encoding type it was, thus we specified that our book is in utf8 when loading the dataset in.

Text

Description automatically generated

*Figure 2, Seeing our data for the first time*

After loading our data in, I loaded it up to see what kind of data cleansing we must do. One of the key steps in NLP pre-processing is removing irregular expressions. This is due to the fact that although punctuation marks are commonly employed in text, irregular expressions like full stops or exclamation marks have an impact on the outcome of any text processing strategy, especially those that depend on the frequency of occurrences of words and phrases.

In our dataset, we can see that there is a lot of “\n”, as well as punctuations in the text. Since this was from a book, we can also see that there is “page | ” inside the text file to show where each page ends.

A screenshot of a computer

Description automatically generated with medium confidence

*Figure 3, Viewing data in .txt*

I have also noticed that after every page it will show “Harry Potter and the Philosophers Stone – J.K. Rowling” and removed this line physically from our text file. Personally, I found removing this line relatively important, since before removing it doing text generation will just end up with many of these words being present in the text.

Text

Description automatically generated with medium confidence

*Figure 4, Items that was removed*

After analysing our text, above are the items that we have removed – we removed punctuations, “\n”, as well as other irregular expressions such as hyphens, colons, and the “page |” for after the end of each page. After scrolling through the dataset, I discovered more irregular expressions such as “■” as well as “•”, and removed them as well, since they serve no purpose in our dataset.

Text

Description automatically generated

*Figure 5, Tokenizing text*

Finishing off removing the irregular expressions, we then proceed to tokenize our text. Tokenization is used to separate the text into smaller units, to allow stop words to be removed and text to be lemmatized.

Text, letter

Description automatically generated

*Figure 6, Removing stop words*

Stop words are words that are generally filtered out before our processing. These words are usually the most common words in the dataset but also render to be the most redundant. By removing these stop words, we can remove the low-level information from our text and give more focus to the important information. In the above code, we are importing the stop words from NLTK. Since NLTK caters to multiple different languages, we would have to specify that our stop words are in English. We would then remove the stop words from our text.

Text, letter

Description automatically generated

*Figure 7, Lemmatizing text*

After removing the stop words from our data, we move on to lemmatize the text. Lemmatization is grouping together the different forms of the same word. We did not use stemming in this dataset as it chops off the word without taking in the context of the word. Although it is proven to be faster, it usually provides lesser accuracy compared to lemmatization. Lemmatization is used in NPL and text analysis as it can reduce the different forms of words to a common base form. In our code, we are first downloading the lemmatization package from NLTK. We then apply lemmatization by running a loop to replace all the words with the lemmatized words from the dictionary.

Text

Description automatically generated

*Figure 8, Text after lemmatizing*

After our text has been processed and cleaned, we can see that compared to *Figure 2*, there is quite a significant amount of change – all our irregular expressions as well as stop words has been removed, and we have also lemmatized the data.

## Preparing data and one-hot encoding

To prepare our data for one-hot encoding, we must first have data for it. Since our data is an entire book and not multiple instances of information, we must change them into many lines of sentences for our machine to learn it properly.

Text

Description automatically generated

*Figure 9, Splitting our text into training data*

The code above is used to split our text into multiple instances of training data. What is happening in the code is we are specifying the maximum length of character that should be in each text, as well as the difference (steps) in characters from the first and second data. After making our data with a step of 3, we can see that the total number of data we have is around 104000.

One-hot encoding is a process of converting data to prepare it for an algorithm. We do this by converting each value into a new column and assign a binary value of 1 or 0 (True or false) onto these columns. By using one-hot encoding, we can ensure that our models do not assume that higher numbers are more important.

Text

Description automatically generated

*Figure 10, One-hot encoding our data*

From our code, we are one-hot encoding the data into a 3D Numpy array, with the shape of sequences, maximum length, and characters. Additionally, we create an array y containing the one-hot encoded characters that come right after each extracted sequence.

# Developing the sequence generator models

For generating the models, we are going to follow the universal machine learning workflow to develop and improve the model accuracy. That means to first build a baseline model, scale it up till overfitting is occurred and regularize the model accordingly. For each of the models, we have used validation\_split in the model fit to split the data into training and validation samples.

I have used the sample function from practical 8b for generating new text.

A screenshot of a computer

Description automatically generated with medium confidence

*Figure 11, Sample function*

From the code, we will be able to reweight the original probability distribution coming out of the model, and draw a character index from it. We will be building models with different RNN layers, such as GRU, LSTM and Conv1D. We will then compare the accuracy of the best models to make the final model.

## Model with GRU

### First model with GRU

For our first model, we started off with a 1-layer GRU model. We started off with a simple model so we can later scale up and regularize the model. We follow the single layer with a dense classifier and SoftMax over all possible characters.

Graphical user interface, text, application

Description automatically generated

*Figure 12, Model 1 with GRU*

Since our data are one-hot encoded, we will use categorical crossentropy as the loss to train the model. We are also using Adam as our optimizer since it generally performs better compared to other algorithms and require lesser parameter for tuning.

Text

Description automatically generated

*Figure 13, Training, and text generation*

For our text generation, we used a loop and set different temperatures to them to see how our text generation will evolve as the model starts converging, as well as the impact of temperature in the sampling.

A picture containing diagram

Description automatically generatedChart

Description automatically generated with medium confidence

*Figure 14, Model 1 Accuracy and Loss*

From our first model, we can see that our model’s training accuracy continuously rises till 70%, and probably will continue further if we ran for more epochs, while our validation accuracy followed closely to the 5th epoch, and starts to separate out from the training accuracy, maintaining at around 0.47 accuracy as it slowly decreases after the 10th epoch. We can also see from our validation loss that the model started to overfit exponentially after around the 10th epoch, while the training loss continues to decrease. Text, letter

Description automatically generated

*Figure 15, Model 1 Generated text*

From our generated text, we can see that with temperature set to 0.2, the text generation is very erratic, with many short 1–2-character words. For the higher temperatures, we can see that our text generation after “bedt” ended up with a long word “bedtaterunddinvoothats”, which tells us that in higher temperatures our structure starts to break down, and most words looks like semi-random strings of characters.

### Adding regularizer to first model

Following the universal machine learning workflow, we go on to regularize our first model.

Text

Description automatically generated

*Figure 16, Tuning model 1*

After we see overfitting on the first model, we move on to tune our model. We added a dropout of 0.2, and ReLu activation into our model.

A picture containing shape

Description automatically generatedA picture containing shape

Description automatically generated

*Figure 17, Accuracy and Loss after tuning*

After adding a dropout of 0.2, we can see that our model’s validation accuracy was higher than that of the training accuracy for a good 50 epochs, before slowly coming close together and training accuracy overtaking validation accuracy. There is no overfitting observed in this model. Our model’s validation accuracy came out to be around 0.5039.

Text, letter

Description automatically generated

*Figure 18, Generated text with tuned model*

We then generated text from our tuned model. When our temperature is set to 0.2, we can see that there is a lot of repeated words, such as “harry”. There is also not a lot of interesting content present. With higher temperatures, we can see that the words get more dense, but many of the words are not English – more of random strings of character. The most interesting temperature is 0.5, where we can see decently generated English words although there is some non-English words.

### Second model with GRU

After trying out a single layer GRU model, I decided to try out more layers for GRU model. However, since we are dealing with text generation and there is not as many features to capture as CNN, I would try to limit the number of layers used to at most 3. For my second model, I decided to use 2 layers.

A screenshot of a computer

Description automatically generated with medium confidence

*Figure 19, Second model with GRU*

From our second model, we added a dropout of 0.2 just to keep it similar with our first model, with the only difference being an additional dropout layer. We have set return sequences to True in our first layer, since in RNN it is defaulted as false. Return sequences will return the last output in the output sequence, in this case the previous layer.

A picture containing shape

Description automatically generated A picture containing graphical user interface

Description automatically generated

*Figure 20, Accuracy and loss for second model*

In our second model, after training it for 100 epochs, we can see how the 0.2 dropout is really keeping the overfitting in check. Although our validation loss is slowly getting higher, we can see that it is not as drastic spike as our model 1 without any dropout. By adding an additional layer in our model, we barely see any inprovement in our accuracy which is now at 0.5059 (compared to model 1 – 0.5039), but we see the computational time per epoch rise from 45 seconds in model 1 to 140 seconds in this current model.

Text

Description automatically generated

*Figure 21, Text generation with model 2*

When running our model 2 text generation, we can see how in the lower temperatures the text is very repetitive, with words such as “harry” and “hermione” coming up very frequently. With higher temperatures, we can see that the text generation is very messy, with lots of unreadable strings of words. The more interesting temperature would be around 0.5 and 1.0, since the words there are majority english, and with temperatures of 1.0 even generating English like words such as “bearnizeve” or “bristed”.

## Model with Conv1D

### First model with Conv1D

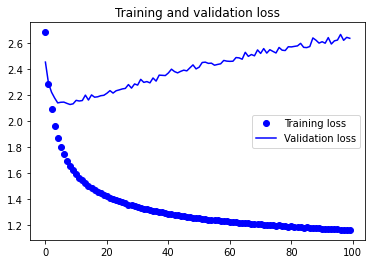
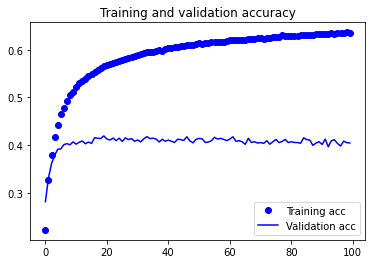
In Conv2D layers, each channel in the input and filter is 2 dimensional, whereas Conv1D means each channel in the input and filter is 1 dimensional. Since we are doing sequences for assignment 2, it is possible to use Conv1D for text generation. Conv1D’s model architecture is very similar to that of Conv2D, where there is stacks of layers and max-pooling, ending in a global pooling or flattening operation.

A screenshot of a computer

Description automatically generated with medium confidence

*Figure 22, Model with Conv1D*

Following the universal machine learning workflow, we first started off by building a baseline model of our Conv1D by giving it 2 layers of a kernel size of 3, and added max pooling as well as a flatten operation before compiling the model. For our batch sizes as well as epochs, we try to keep the parameters the same to have a better comparison based on RNN layers to see which models perform better.



*Figure 23, Accuracy and Loss for Conv1D model 1*

Immediately, we can see from our validation loss that the model overfitted really early on at around the 10th epoch, while the model’s accuracy maintains at a low 40%. The gap between our training and validation accuracy is also relatively big, with an almost 20% gap between them.

Text, letter

Description automatically generated

*Figure 24, Text generation with Conv1D model 1*

From our first Conv1D generated text, we can see that majority of the words throughout the temperatures are just strings and barely readable, with temperatures of 0.2 being the most interesting so fat with a decent number of English words inside. As the temperature goes on, the words get messier and becomes just random text generation, with only some words being readable or English.

### Adding regularizers to model

To stop the overfitting from happening so early on, we added dropout to each of our layers.

Text, letter

Description automatically generated

*Figure 25, Adding regularizer to Conv1D model*

We implemented a dropout of 0.2 to each of our layers to try and balance out the training and validation accuracy’s gap, as well as to prevent the model from overfitting so early on.

A picture containing graphical user interface

Description automatically generatedA picture containing graphical user interface

Description automatically generated

*Figure 26, Adding dropout to Conv1D layer*

After adding the dropout to our model, we can see that there is no overfitting observed, and our training and validation gap is not as big as it was previously. Assumingly, since we are doing text generation and are feeding words to our model, the model will learn way too much with the training data such that it follows way too closely to it, and thus when validating it will have a big difference. By adding our dropout layers, it prevents the model from learning too much of the training information and overfitting.

Text

Description automatically generated

*Figure 27, Text generation with Conv1D layer*

After tuning our model, we can see the text generation is better compared to before. However, what is interesting is how we see even at temperatures of 0.5, our generated text is still very repetitive, with “harry” coming up very often. With higher temperatures, it is like previously where the words are just strings of text and not readable. Our accuracy score is 0.4390.

### Second model with Conv1D

From our first model, some improvements such as increasing the filter size will help in increasing the general accuracy of our model. Thus, we decided to build a model with size of 128 and 2 layers.

Text

Description automatically generated

*Figure 28, Second model with Conv1D*

Notice we already added the dropout for the mode, since we know that it will overfit and that our filters are higher it should overfit quicker compared to the first model.

A picture containing graphical user interface

Description automatically generatedA picture containing graphical user interface

Description automatically generated

*Figure 29, Accuracy and Loss of model 2*

From our model 2 when we are generating a model with higher filters, we can see that our accuracy and loss is similar to that of the first model when we did not implement any dropout, where the training and validation accuracy has a significant difference of 20% between them.

Text, letter

Description automatically generated

*Figure 30, Text generation with second Conv1D model*

From our second model, we can see that when in lower temperatures of 0.2, the text generated is incoherent and messy, with very short, repeated words like “wa”. With higher temperatures, we can see the words does not make sense and are just string, and many of the words looks unreadable. However, in the temperature of 0.5, a lot of the words generated are interesting ,although some of them are also just strings and unreadable. Our accuracy however seems to be a bit higher than our first model, at around 0.4401.

## Model with LSTM

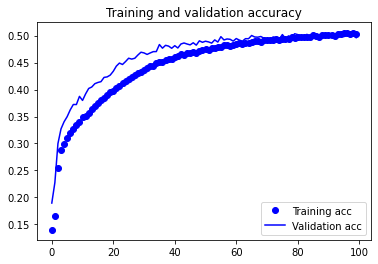
We only developed one LSTM model, mainly for comparisons from the GRU models. This is because we want to see how the different RNN layers would affect our overall accuracy.

Text

Description automatically generated

*Figure 31, Model with LSTM*

Thus, we created a single layer LSTM model and fitted it with the same parameters that was used in the GRU model to see how well it does. We have also included a dropout like that of our GRU model to prevent overfitting.

A picture containing shape

Description automatically generated

*Figure 32, Accuracy and Loss of LSTM model*

Surprisingly, with our LSTM model, we can see that our training as well as validation is following very closely with each other. There is no overfitting observed in our LSTM model, and our accuracy score is relatively high at about 0.4962.

Text, letter

Description automatically generated

*Figure 33, Text generation with LSTM*

From our generated text, we can see that in lower temperatures there is a lot of repetition in the words “harry” and moving up the temperatures to 1.0 and 1.2 we can see that the words starts getting pretty messy generation and become more string like compared to English words. In the temperatures of 0.5, we can see that the text generated still is quite repetitive, but there is many English words generated as well.

## Final Selected Model

Our final model is selected based on accuracy as well as performance. The model with the highest accuracy is our model with GRU, which has an accuracy of above 50%.

Text, letter

Description automatically generated

*Figure 34, Final Model used*

For the final model, we used 2 layers of GRU as it has proven to have the highest accuracy amongst all our tested models. Since the model was still overfitting after adding 1 layer of dropout, we implemented 2 layers of dropout and tuned it a little so that it does not underfit.

Shape

Description automatically generated with medium confidenceA picture containing graphical user interface

Description automatically generated

*Figure 35, Training, and validation of final model*

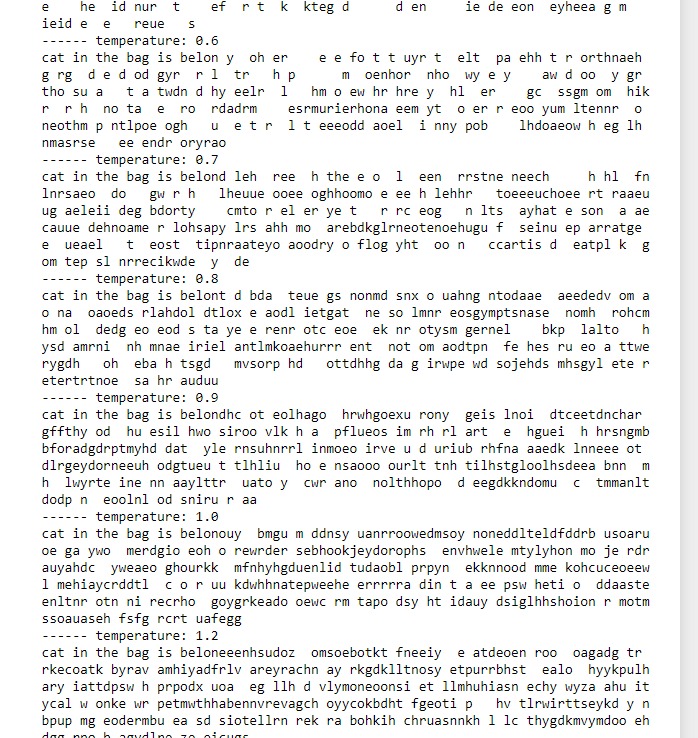
From the final model, we can see that the training and validation follows each other relatively closely with a 5% difference, and there is no overfitting observed.

# Text generation with developed model

## Observations with different temperatures

We used our best model for the generation of our texts. To find the best temperature which our text can be generated in, it is most suitable for us to generate text throughout all the temperatures and see which temperature makes the most sense to us.

Our text generation will allow the user to have an input on what they want to type, and the words will be generated accordingly afterwards. For our input, it has a caveat – it must have 60 characters for the text to be generated correctly.



*Figure 36, Incorrect text generation due to wrong number of characters*

From the figure above, we can see that the text is incorrectly generated as it is not the specified correct amount of text that was generated. This is because the length of the input has to be consistent during training phase and testing phase.

Graphical user interface, text

Description automatically generated

*Figure 37, Asking for user input*

From our text generator, we are first loading back our previously saved model. After that, we ask the user for an input and convert it to a string. This way, our generator model can read off it and add more text into it. After specifying our input of 60 characters, in this example “whos joe a distant voice asks instantly everyone nearby hear”, we start generating our text.

Text

Description automatically generated

*Figure 38, Text generating with input text*

From our text generation code, we are setting the initial text to our text input, as well as specifying a range of temperatures from 0.2 to 1.2 to see how well our text generates for each of the temperatures. We then generate 400 characters from our best model.

Text, letter

Description automatically generated

*Figure 39, Text generation on lower temperatures*

For our text generation with lower temperatures, we can see that the words are very repetitive. For example, in temperatures of 0.2, we can see that the word “harry” repeatedly comes up numerous times. However, we can note that the word structure is highly realistic – most of the text generated are real English words, or words from our text data. In temperatures of 0.3, we can also see a similar trend, where words such as “harry” is repeated over again, while there is no other interesting data.

Text

Description automatically generated

*Figure 40, Text generation on middle temperatures*

For our text generation in the temperature range of 0.4 to 0.7, we start to see more interesting words. In temperatures of 0.4, we can still see quite an amount of repetition for the words “harry”. In the higher up temperatures of 0.5 to 0.7, we can see that the text generated is really good, with words from English and sometimes even words that seems English but isn’t, such as “fresty” and “petunia” in temperatures of 0.7. These range of text is no doubt the most interesting in our model.

Text, letter

Description automatically generated

*Figure 41, Text generation with higher temperatures*

With higher temperatures, we can see that the words generated started getting more messy but interesting. We can see that there is a lot of longer words being generated, and although it does not seem to make sense, a lot of the words are still pronounceable. The words being generated also gotten messier as the temperatures go on, in temperatures of 0.8 we can still see decent English structure but by temperatures of 1.2 it is more of string – like words.

## Overall observations from text generation

From our generated text, we see a general trend. In lower temperatures, the text generated is often extremely repetitive and predictable, but have a highly realistic local structure. This means all words generated are mostly English, and repetitive words are often words that comes up the most in our training data. In this case, “harry” repeatedly comes up in lower temperatures as it is mentioned a lot in our training dataset. With higher temperatures, the generated text becomes more interesting, surprising, and even creative. It can even sometimes create completely new words that sounds somewhat plausible, such as “silverder” and “gruptied”. With higher temperatures, we can also see that the local structure starts breaking down, with most words looking like semi-random strings of characters. Without a doubt from our dataset, temperatures in the range of 0.5 to 0.7 boasts the most interesting find.

# Summary and further improvements

## Summary and findings

Below are the different model accuracies for my models

|  |  |  |
| --- | --- | --- |
| Model | Validation Accuracy | Validation Loss |
| GRU model 1 | 0.4497 | 2.314 |
| GRU model 1 (After Tuning) | 0.5039 | 1.6287 |
| GRU model 2 | 0.5059 | 1.6778 |
| GRU model 2 (After Tuning)  Final Model | 0.5102 | 1.6256 |
| Conv1D model 1 | 0.4044 | 2.6348 |
| Conv1D model 1 (After Tuning) | 0.4390 | 1.9668 |
| Tuned Conv1D model 2 | 0.4401 | 2.0554 |
| Tuned LSTM model | 0.4962 | 1.6551 |

*Figure 42, Table of different models built*

From our report, the model that was selected as our final model for the text generation is GRU model 2, which showcased the highest accuracy amongst our training, as well as one for having one of the lowest validation losses.

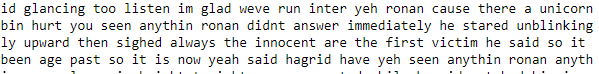
There different models performed an average of 45% in terms of accuracy, and some interesting findings noted is the speed of training when it comes to the different models. With our GRU models, the span of our training time is from 60 seconds to over 200 seconds per epoch. We have different GRU models with the different densities, and it is no surprise that the models with bigger parameters will have to take a longer time to train. What is interesting is also the speed that our Conv1D model takes to train. Even with multiple layers, our Conv1D models are training an average of 10 to 30 seconds per epoch, but still coming in with a decent accuracy of about 44% for my best model. That is almost as good as our base model for GRU, which although had lesser parameters is training at almost double the time it takes to train the Conv1D model.

With the LSTM model, LSTM can keep up well with our GRU model which has the same parameters, as my GRU model 1 (After tuning) is identical to the LSTM model in terms of parameters. The difference between GRU and LSTM is how GRU’s bag has two gates that are reset and updated, while LSTM has three gates that are input, output and forget. GRU is less complicated than LSTM as it has lesser number of gates and is generally seen to perform better with smaller datasets. If our dataset was set to have 20 or 30 characters in one sequence instead of 60, we might potentially see LSTM outperform GRU.

## Further improvements

From our best model, there is still further improvements that can be done. Our best model does not have the “best” model layers, as we did not try out many different amounts of layers as well as parameters to find a model that can perform the best. Our final model also still can be tuned more, as there is still a gap between the training and validation accuracy.

Text cleansing can be further improved as well. Even in our “cleansed” data that is has stop words removed, as well as text lemmatized, there is still potential dirty data inside our text. For example with our current data:



*Figure 43, Cleansed text*

With our cleansed data, we can see there is still weird words inside, which might be caused by our NLTK stop words and lemmatization. Words such as “weve”, “yeh”, “anything”, are not words from English, yet is in our text data after lemmatization. Thus, we could have performed our lemmatization and text cleansing better.

We can train bigger models with more epochs and implement bigger quantities of text to achieve better generated sampling that can look more coherent and realistic as well. Having a more coherent and realistic looking text doesn’t necessarily mean that the text generated will be meaningful, since text generation is based more on if words are formed than the structure of the sentence. All we do is to sample data from a statistical model to find which characters comes after which.

We can also try out more RNN types that is not mentioned in our current report, even if it may not be the most optimal. We can also add embedding layers, which initialize the embedding vector at random and uses network optimizer to update it. This can help make it easier to do machine learning on large inputs like sparse vectors representing words. On the topic on trying out other things, we could have used functional API as well to manipulate tensors and use layers as functions that takes and returns tensors. By implementing functional API, we could create models that are more flexible than the sequential API, as functional APIs can handle models with non-linear topology, shared layers and multiple inputs or outputs.

# References

oneThousandHertz. (2017, November). Keras LSTM for Text Generation keeps repeating a line or a sequence. Stack overflow.

<https://stackoverflow.com/questions/47125723/keras-lstm-for-text-generation-keeps-repeating-a-line-or-a-sequence>

TensorFlow. (2022, May). Text generation with an RNN

<https://www.tensorflow.org/text/tutorials/text_generation>

Rohan, R. (2020, July). Text Generation With RNN + TensorFlow

<https://www.section.io/engineering-education/text-generation-nn/>

Vadapalli, P. (2021, Feb). CNN vs RNN: Difference Between CNN and RNN

<https://www.upgrad.com/blog/cnn-vs-rnn/>