# 



**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** |
| **Student Name** | **:** | 1. Lim Kai Chong S10226797   (PROBLEM 2) |

**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.

Table of Contents

[1](#_Toc111212841)

[1. Overview 3](#_Toc111212842)

[Problem and Objective 3](#_Toc111212843)

[Approach 3](#_Toc111212844)

[2. Data Loading and Processing 5](#_Toc111212845)

[Data Loading 5](#_Toc111212846)

[Data Processing 6](#_Toc111212847)

[Reducing the Vocabulary of the Text Corpus 6](#_Toc111212848)

[Data Preparation and Labelling 8](#_Toc111212849)

[3. Develop the Sequence Generator Models 10](#_Toc111212850)

[Temperature Sampling in Text Generation 10](#_Toc111212851)

[Choosing the architectures and how the models will be built 11](#_Toc111212852)

[Building the LSTM model 13](#_Toc111212853)

[Step 1: Determining the optimal weights 14](#_Toc111212854)

[Step 2: Determining the optimal batch size and optimizer 15](#_Toc111212855)

[Step 3: Reducing Overfitting 17](#_Toc111212856)

[Rerunning LSTM Model 20 until it overfits 18](#_Toc111212857)

[Building the GRU model 20](#_Toc111212858)

[Step 1: Determining the optimal weights 20](#_Toc111212859)

[Step 2: Determining the optimal batch size and optimizer 21](#_Toc111212860)

[Step 3: Reducing Overfitting 22](#_Toc111212861)

[Rerunning GRU Model 20 until it overfits 22](#_Toc111212862)

[Experimenting and building Conv1D models 23](#_Toc111212863)

[Developing the Best Model and Analyzing Temperature 24](#_Toc111212864)

[Bidirectional LSTMs 25](#_Toc111212865)

[Temperature Analysis 26](#_Toc111212866)

[4. Use the Developed Model to Generate Texts 27](#_Toc111212867)

[Recording Input 27](#_Toc111212868)

[Generating Text 29](#_Toc111212869)

[5. Summary 30](#_Toc111212870)

[Conclusion 30](#_Toc111212871)

[Further Improvements 30](#_Toc111212872)

# Overview

## Problem and Objective

We encounter different types of data in our day-to-day lives, and some of this data has temporal dependencies while some does not. Sometimes, in the case of unstructured natural language data, networks may need to store and retrieve useful information from the past in order to analyse current input, or even make a prediction for future inputs. Unfortunately, traditional neural networks or feed-forward neural networks do not have this capability to process data sequentially. As a result, for processing data with temporal dependencies, architectures such as Recurrent Neural Networks (RNNs) are used instead. With the capability to process sequential data, RNNs can not only be used as classification models to classify sentiments given textual data, but RNNs can also be used as generative models to learn the sequences of a problem and then generate entirely new plausible sequences for the problem domain. In this assignment, RNNs will be used to train deep learning models for character-by-character text generation using the complete version of J.K. Rowling’s book Harry Potter and the Philosopher’s Stone as the text corpus. By implementing the functionalities of an RNN, the objective of this assignment is for the developed RNN model to be able to create an English language character generator capable of building semi-coherent English sentences from scratch.

## Approach

In order to build a deep learning model that can perform character-by-character text generation, a methodological approach following the universal workflow of machine learning would need to be leveraged. For any machine learning problem, the first step would always be to load and process the data. In this step, the textual data will first be read, and basic data understanding will be performed by identifying the number of characters in the entire text corpus. Thereafter, basic data cleansing will be applied on the textual data by removing any special characters and/or unnecessary spaces from the text corpus. Once the textual data is cleansed, data will then be parsed into more easy-to-interpret forms by sampling the textual data into sequences and target characters, both of which to be stored as the X and y variables respectively using the “sliding window” method, which will be discussed in greater depth later in this report. After the X and y training and test samples are obtained, one-hot encoding will be performed on the samples to convert and store the results in a 3D NumPy array, where each character is stored as a sparse vector representation of the text sequences and the target labels. After vectorization, the sequence generator models will then be able to be properly trained and developed. As mentioned, because of their ability to analyse sequential data, RNNs, along with its variants, will be the main architecture explored in this step. This will include exploring advanced RNN architectures such as the Gated Recurrent Unit (GRU) network and the Long Short-Term Memory (LSTM) network. In addition, 1D Convolutional Neural Networks will be considered as well. In this step, experimentation is key. The process of developing and determining the best sequence generator model is one that involves a lot of experimentation and patience. As such, experimentations and comparisons will be performed across multiple architectures, and for each architecture, the universal machine learning workflow will be closely followed, that is, starting with a baseline model, scaling up the model until it overfits and regularizing and tuning the model hyperparameters accordingly. It can be noted that in the interest of time, most models was only trained up to epoch 20 or 30, making the overall evaluation not the most precise estimate of each model’s performance, but rather an approximation, especially for models that have a slower but smoother learning process. Across the trained models, new texts will be generated as well by applying proper sampling strategies, leveraging probability distributions and reweighting the distributions based on sampling function parameters such as temperature accordingly. Finally, using the developed model, text will be generated by recording input, encoding the recorded input, feeding the encoded input into the model to let the model generate 400 characters, and lastly analysing the overall coherence of the generated text. To better visualise the approach taken, a visual diagram was created:

Diagram

Description automatically generated with low confidence

Figure .1 – The 3-part approach used to develop the sequence generator model

Figure 1.1 depicts the 3-part approach used to tackle this assignment. As mentioned above, Part 1 will involve loading and processing the data, Part 2 will involve developing the sequence generator models, and Part 3 will involve using the developed model to generate texts.

# Data Loading and Processing

## Data Loading

The first step to developing any sequence generator model is to first load the text corpus to be used to train the generative model.



Figure .1 – Code used to load the text corpus

As seen from Figure 2.1, the text file containing the full text of Harry Potter and the Philosopher’s Stone was first loaded as the text corpus. After loading the text corpus, the corpus can be seen to have a length of 474429 characters.

Text

Description automatically generated

Figure 2.2 – First 1000 characters of the text corpus

With reference to Figure 2.2, the first 1000 characters of the text corpus was printed to get a better idea of the text corpus in its raw form. As seen from the Figure, there are a lot of redundant spaces and special characters such as ‘/’ that do not contribute to the contextual semantics of the document. As such, in the next step, data will be processed with the aim of removing these redundant elements from the text corpus, to remove as much noise as possible in the textual data that could potentially affect the accuracy of the model.

## Data Processing

### Reducing the Vocabulary of the Text Corpus

As mentioned, the first aim for data processing is to remove any noise in the data by removing redundant elements that do not add to the contextual semantics of the text corpus.

Chart, text, scatter chart

Description automatically generated

Figure 2.3 – Function to cleanse the text corpus

Figure 2.3 shows the function used to cleanse the text corpus. In the figure, the function ‘clean\_text’ was used to filter out special characters by defining an allowable vocabulary list. In the function, a punctuation list specifying all the acceptable punctuation types, and a string specifying all 26 alphabetical characters were created to serve as the allowable vocabulary list of the text corpus. Thereafter, every character in the text parameter passed into the function will be read, and only if the character is within the allowable vocabulary list will the character be appended to the cleansed text corpus variable.

Text

Description automatically generated

Figure 2.4 – Cleansed text corpus

With reference to Figure 2.4, after the text corpus is passed into the ‘clean\_text’ function, there are no longer any special characters or redundant spaces in the text, reducing the vocabulary that the network must learn.

### Data Preparation and Labelling

The next step of data processing will involve generating sequences and target labels from the cleansed text corpus to serve as the X and y labels for the network. In this step, this is done using the **Sliding Window Method**. The Sliding Window Method is a computational method used analyse a long chain of data by incorporating the use of fixed Window sizes and steps. For example, consider a sequence 5, 6, 4, 3, 8. Imagine if we wanted to obtain the largest sum of three consecutive integers in the sequence, how would we do it? We would first have to analyse the entire sequence in groups of three, where the first three integers would be the subsequence 5, 6, 4 of the sequence of integers, to obtain the sum of three consecutive integers. In such cases, the Sliding Window Method can be used whereby each subsequence can be defined with a window size of 3 and a step of 1 to loop through each combination of 3 consecutive integers in the entire sequence with a one-index-incremental. The figure below shows a simple depiction of how the Sliding Window method can be used to extract sequences from a chain of data.

A picture containing table

Description automatically generated

Figure 2.5 – Simple depiction of the Sliding Window method

As such, from this example, it can be seen that the use of the Sliding Window method is an efficient means to extract **partially overlapping sequences** and can be incorporated to generate and retrieve sequences of text, as well as target characters, to serve as the X and y labels from the text corpus.

Text

Description automatically generated

Figure 2.6 – Extracting sequences and corresponding target characters

Figure 2.6 shows the process of extracting the partially overlapping sequences and the corresponding target characters. From the figure, the window sizes and step were first defined at values of 60 and 3 respectively, suggesting that each extracted sequence will be 60 characters long, and each sequence will be sampled every 3 characters. After defining the window size and step values, a loop is run through the entire text corpus at an interval of 3 characters and at each state, a 60-character sequence is appended to the list ‘sentences’ that stores all the sampled sequences. At the end of every sequence, the target character is retrieved and appended to a list ‘next\_chars’, as seen from the output where the next character of the first sequence that ends with ‘pri’ is shown to be ‘v’. After the sequences and target labels are extracted, a list of all the unique characters that exist in the text is then obtained and a dictionary ‘char\_indices’ is created to map each unique character to their index. This is so that one-hot encoding can be performed on our textual data to make the data more interpretable and easier for the model to learn.

Graphical user interface, text, application

Description automatically generated

Figure 2.7 – One-hot encoding process to convert text into sparse vectors for better representation

As seen in Figure 2.7, each character is mapped to their index and one-hot encoding is performed to convert the textual sequences and target characters into sparse vectors. After the encoding process, the outputs are stored into the x and y variables respectively, which are represented by 3D NumPy arrays. In the 3D NumPy array for x, the shape (149603, 60, 35) represents the total number of extracted sequences (149603), the character-length of each sequence (60), and the number of unique characters (35). With this, the text has finally been loaded, cleansed and vectorized into two separate 3D NumPy arrays x and y.

# Develop the Sequence Generator Models

## Temperature Sampling in Text Generation

Before the architectures and approaches used to develop the best performing language model are explored, it is important to first understand how the sampling process of text generation works, or more specifically how temperature sampling in text generation works, as a means to observe and extract patterns from language models.

Temperature Sampling is inspired by statistical thermodynamics, where high temperature means low energy states are more likely encountered. In language models, **logits** play the role of energy and temperature sampling can be implemented by dividing logits by the temperature before feeding them into a softmax activation function. Thereafter, the process of generating text is as follows:

1. Drawing from the model a probability distribution over the next character given the text available so far
2. Reweighting the distribution to a certain “temperature”
3. Sampling the next character at random according to the reweighted distribution
4. Adding the sampled character to the end of the available text

Text

Description automatically generated

Figure 3.1 – Code used to create the sampling function

Figure 3.1 shows the sampling function ‘sample’ used to perform temperature sampling. As seen in the figure, ‘preds’ exist as the **logits**, that are divided by the temperature so that the probability distribution can be reweighted for the next character to be sampled. It is good practice to repeatedly train and generate text when developing language models as this provides insight as to how the generated text evolves as the model starts converging, as well as the impact of temperature in the sampling strategy. In the case of this assignment, text will be generated using temperature sampling for the baseline model and for the final developed model to make comparisons between the quality of the generated text both of the models produce.

## Choosing the architectures and how the models will be built

As mentioned, the process of developing the best performing sequence generator model requires lots of experimentation. In the interest of time, hypothetical considerations would need to be made to weigh the architectures that have the highest likelihood of developing the best language model. With this consideration in mind, looking at SimpleRNN, the simplest implementation of an RNN provided by Keras, it can be assumed that the network would not have a high likelihood of developing the best model, because the network suffers from a **Vanishing Gradient.** During the training process, it is common for the gradient of na RNN to eventually become too small, because of its multiplicative nature, and because gradients usually have a value of between 0 and 1. With a small gradient, the learning rate of the model will always decrease when multiplied, causing a geometric decay of information, resulting in the network being unable to learn properly, or what is referred to as **underfitting**. As such, for RNN experimentation, exploring other more advanced RNN variants such as the LSTM network and the GRU network would be a more optimal approach. In addition to these RNN variants, 1D Convnets should be something to consider as well, as they have proven to be successful in recent times as well.

For the approach, each architecture will be explored systematically in 3 steps. Firstly, the optimal weights for the model will be determined by identifying the best hyperparameters for the recurrent units and layers. The best combination of weights would be the model that achieves the highest validation accuracy before the model overfits. Secondly, after the optimal weights are determined, the optimal batch sizes and optimization algorithm for each architecture will be determined. Lastly, techniques to reduce overfitting will be considered as well, namely different combinations involving L2 regularization, recurrent dropout and regular dropout. Finally, after the developed models are built for each architecture, comparisons will be made to see which architecture produced the best model, and once the best model is achieved, the best model will be repeatedly trained and used to generate text at each epoch to observe the learning process of the best model.

## Building the LSTM model

First, the baseline LSTM model is defined:

Table

Description automatically generated

Figure 3.2 – Baseline LSTM model

With reference to Figure 10, the baseline LSTM model is a single LSTM layer with 8 recurrent units, followed by a Dense classifier and softmax over all possible characters. The model utilizes the RMSprop optimizer at a learning rate of 0.01, trains at a batch size of 128 and is compiled with the categorical\_crossentropy loss function because the targets are one-hot encoded.

A picture containing text

Description automatically generated

A picture containing text

Description automatically generated

Figure 3.3 – Analysing generated text from baseline model

Figure 3.3 shows the text generated by the last epoch of the baseline model. From the figure, it can be seen that the text generated at a temperature of 0.2 is very repetitive but mostly coherent and actual English words. Since the accuracy of the baseline model is around 40%, the generated samples seem very underwhelming and uninteresting. After developing the best sequence generator model, the model will be used to generate text and then hopefully there will be more interesting and coherent sentences, so that more analysis can be made.

### Step 1: Determining the optimal weights

As mentioned, the first step will involve determining the optimal weights for the LSTM model, with the aim of not only increasing the model accuracy, but also to spot patterns in overfitting across different weight magnitudes. For this step, the optimal number of recurrent units will first be determined.

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Weights** | **Epoch where model overfits** | **Validation accuracy** |
| 1 | 8 recurrent units (Baseline) | Nil | 41.5% |
| 2 | 16 recurrent units | Nil | 48.0% |
| 3 | 32 recurrent units | Nil | 52.6% |
| 4 | 64 recurrent units | Nil | 55.8% |
| 5 | 128 recurrent units | Nil | 57.3% |

Table 3.1 – Tuning the optimal number of recurrent units

With reference to Table 3.1, 5 different models were tested each incrementing from 8 recurrent units in Model 1 to 128 recurrent units in Model 5. Despite training the models for 30 epochs, none of the models displayed indications of overfitting. As seen from the Table, there is an increasing trend in the number of recurrent units, where the larger the weights, the higher the accuracy. As mentioned earlier, due to time constraints, each model was only trained for 30 epochs. In the table, the validation accuracy recorded for each model was either the validation accuracy of the last epoch the model was trained till, or the epoch in which the model overfits, whichever that comes first. Since Model 5, with 128 recurrent units managed to achieve the highest validation accuracy of 57.2% at epoch 30, 128 recurrent units for the LSTM model will be used moving forward.

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Weights** | **Epoch where model overfits** | **Validation accuracy** |
| 6 | 1 LSTM layer (128 recurrent units) | 15 | 57.3% |
| 7 | 2 LSTM layers (128 recurrent units) | 10 | 57.4% |
| 8 | 3 LSTM layers (128 recurrent units) | 10 | 56.4% |

Table 3.2 – Tuning the optimal number of recurrent layers

Table 3.2 shows the results obtained from experimenting with different number of LSTM layers. As seen from the table, Model 7 performed the best, obtaining a validation accuracy of 57.4%. However, Model 6, which only has a single 128-node LSTM layer, is able to achieve similar results, at a difference of only 0.01% validation accuracy. In addition, Model 6 can also be seen to overfit later at epoch 15 instead of epoch 10. As such, by weighing in other factors such as performance cost and the learning process, Model 6 can be said to be a slightly better model than Model 7. Therefore, moving forward, the optimal weights the LSTM model will use will be a single LSTM layer with 128 recurrent units.

### Step 2: Determining the optimal batch size and optimizer

In any deep learning problem, determining the optimal batch size and optimization algorithm is a crucial step in order to develop the best performing deep learning model. Despite some sources claiming that RMSprop “is just the Adam optimizer with momentum”, these descriptions are oftentimes shorthanded, and should not be taken at face value. This section serves as a stark reminder that in most cases, the optimal batch size and optimization algorithms usually vary from dataset to dataset and should be important hyperparameters to be tuned in every deep learning problem.

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Batch Size** | **Epoch where model overfits** | **Validation accuracy** |
| 9 | 32 | 8 | 55.1% |
| 10 | 64 | Nil | 56.8% |
| 11 | 128 | 15 | 57.3% |
| 12 | 256 | Nil | 56.8% |
| 13 | 512 | 8 | 56.7% |

Table 3.3 - Tuning the optimal batch size

Table 3.3 shows the results after experimenting with different batch sizes. In this case, 5 models were tested ranging from batch sizes of 32 in Model 9 to 512 in Model 13. From the table, Model 11, with a batch size of 128 seems to be the best performing model, with the highest validation accuracy of 57.3%. As such, a batch size of 128 will be optimized moving forward.

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Optimizer** | **Epoch where model overfits** | **Validation accuracy** |
| 14 | Adam | 15 | 57.6% |
| 15 | RMSprop | 15 | 57.3% |
| 16 | Rectified Adam | Nil | 52.5% |

Table 3.4 – Determining the best optimizer

Table 3.4 shows the results after experimenting with different optimizers. In this case, 3 optimization algorithms, namely Adam, RMSprop and Rectified Adam were tested. To elaborate, Rectified Adam is a variant of the Adam optimizer whose adaptive learning rate is rectified. This provides a consistent variance with the aim of resolving the generalization and variance issues faced by adaptive learning rate optimizers like Adam

Chart

Description automatically generated with low confidenceChart

Description automatically generated

Figure 3.1 – Left: Rectified Adam; Right: Adam

From Figure 3.1, it can be seen that the Rectified Adam optimizer indeed performs better in regard to variance and generalization, as the curves observed on the left depict a much smoother learning process with no indications of overfitting. Whereas on the right, indications of overfitting can be seen to develop at epoch 15, where the validation loss starts to gradually increase. However, to compensate, the standard Adam optimizer is still able to achieve better performance. In fact, with reference to Table 4, the Adam optimizer managed to achieve the highest validation accuracy out of the 3 optimizers, at a validation accuracy of 57.6% before overfitting. As such, the Adam optimizer will be used moving forward.

### Step 3: Reducing Overfitting

In addition to the previous steps, tackling the issue of overfitting is also an important step in the process of developing a good deep learning model. This is because any good deep learning model would need to have a smooth learning process to capture as much contextual information as possible. In this section, combinations of various techniques known to solve overfitting will be used, namely L2 regularization, regular dropouts, and recurrent dropouts. To clarify, recurrent dropouts apply dropout within recurrent units between different states to account for overfitting that may occur in recurrent layers.

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Techniques Applied** | **Epoch where model overfits** | **Validation accuracy** |
| 17 | None | 15 | 57.6% |
| 18 | Only L2 regularization (0.001) | Nil | 52.6% |
| 19 | Only Regular Dropout (0.1) | 8 | 56.1% |
| 20 | Only Recurrent Dropout (0.1) | Nil | 58.3% |
| 21 | Recurrent Dropout + Regular Dropout | Nil | 57.1% |
| 22 | Recurrent Dropout + L2 Regularization | Nil | 52.5% |
| 23 | Regular Dropout + L2 Regularization | Nil | 52.0% |
| 24 | Recurrent Dropout + Regular Dropout + L2 Regularization | Nil | 51.1% |

Table 3.5 – Determining best combination of techniques to reduce overfitting

Table 3.5 shows the results obtained from testing 8 different LSTM Models. From the table, it can be seen that apart from utilising only regular dropout, which caused the model to overfit earlier at epoch 8, every other combination of techniques managed to alleviate the issue of overfitting. In addition, Model 20, which utilised a recurrent dropout of 0.1, also managed to increase the accuracy of the Model to 58.3%. Therefore, at the end of these 3 steps, Model 20, which has achieved a validation accuracy of 58.3% at the 20th epoch will be the developed LSTM model, since it is the best performing LSTM model thus far.

### Rerunning LSTM Model 20 until it overfits

Using the same set of hyperparameters, LSTM Model 20 was trained again, this time using Early Stopping to determine when the model stops improving. To elaborate, Early Stopping is a cross-validation strategy that stops a model’s training phase at the point where it has learnt to extract all meaningful relationships from the data. This is done by monitoring a validation metric such that the training phase ends when this metric stops improving after a specified number of epochs.

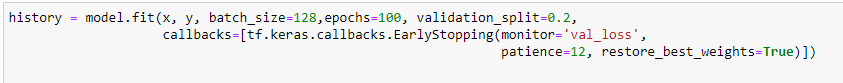


Figure 3.2 – Early Stopping implementation in Keras

As seen from Figure 3.2, the validation metric used in this case is validation loss, and a patience of 12 is specified to stop the training phase if the validation loss does not decrease within 12 epochs.

Chart

Description automatically generatedChart

Description automatically generated

Figure 3.3 – Model performance of LSTM Model 20

As seen from Figure 3.3, the training process of LSTM Model 20 was stopped at epoch 35. Despite no overfitting occurring, it can be seen that from epoch 15-20 onwards, the model stagnates at a constant validation loss/accuracy, suggesting that the model is unlikely to improve its performance as it has already learnt to extract all meaningful relationships from the data. With that being said, LSTM Model 20 was able to hit a validation accuracy of 58.9% after training for 35 epochs.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Model** | **Number of layers** | **Number of recurrent units** | **Batch Size** | **Optimizer** | **Technique to reduce overfitting** | **Validation Accuracy** |
| Baseline | 1 | 8 | 128 | RMSprop | None | 41.5% |
| Final | 1 | 128 | 128 | Adam | Recurrent Dropout | 58.9% |

Table 3.6 – Developed LSTM Model

At a validation accuracy of 58.6%, LSTM Model 17 has a notably larger validation accuracy compared to the baseline model, at a validation accuracy of 41.5%. As such, by following a methodological 3-step approach and by tuning hyperparameters, a LSTM model that performs significantly better than the baseline model was produced.

## Building the GRU model

In this section, the best performing GRU model will be developed. Similar to the LSTM baseline model, the baseline GRU model also has a single GRU layer with 8 recurrent units, a Dense classifier and softmax over all possible characters. The model also utilizes the RMSprop optimizer at a learning rate of 0.01 and a batch size of 128.

### Step 1: Determining the optimal weights

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Weights** | **Epoch where model overfits** | **Validation accuracy** |
| 1 | 8 recurrent units (Baseline) | Nil | 40.0% |
| 2 | 16 recurrent units | Nil | 46.7% |
| 3 | 32 recurrent units | Nil | 51.2% |
| 4 | 64 recurrent units | Nil | 53.3% |
| 5 | 128 recurrent units | Nil | 53.2%% |

Table 3.7 – Tuning the optimal number of recurrent units

With reference to Table 3.7, Model 4, which has 64 recurrent units, achieved the highest validation accuracy of 53.3%. As such, for subsequent GRU models, 64 recurrent units will be used.

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Weights** | **Epoch where model overfits** | **Validation accuracy** |
| 6 | 1 GRU layer (64 recurrent units) | Nil | 53.3% |
| 7 | 2 GRU layers (64 recurrent units) | 10 | 51.2% |
| 8 | 3 GRU layers (32 recurrent units) | 5 | 49.4% |

Table 3.8 – Tuning the optimal number of GRU layers

Table 3.8 shows the results obtained from experimenting with different numbers of GRU layers. From the table, Model 6 with 1 GRU layer performed the best, obtaining a validation accuracy of 53.3%. Therefore, subsequent GRU models will utilize 1 recurrent layer with 64 hidden units.

### Step 2: Determining the optimal batch size and optimizer

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Batch Size** | **Epoch where model overfits** | **Validation accuracy** |
| 9 | 32 | 10 | 49.8% |
| 10 | 64 | Nil | 51.8% |
| 11 | 128 | Nil | 53.3% |
| 12 | 256 | Nil | 54.3% |
| 13 | 512 | Nil | 54.9% |

Table 3.9 - Tuning the optimal batch size

From Table 3.9, Model 9 with a batch size of 512 was able to produce the highest validation accuracy of 54.9% by the 20th epoch. In this case, it can be seen that the higher the batch size, the better the accuracy. In addition, only Model 9 with a batch size of 32 overfitted at epoch 10, suggesting that larger batch sizes alleviates the issue of overfitting as well. Moving forward, a batch size of 512 will be used.

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Optimizer** | **Epoch where model overfits** | **Validation accuracy** |
| 14 | Adam | Nil | 55.1% |
| 15 | RMSprop | Nil | 54.9% |
| 16 | Rectified Adam | 2 | 50.5% |

Table 3.10 – Determining the best optimizer

From Table 3.10, similar to LSTM, the Adam, RMSprop and Rectified Adam optimizers were tested and the Adam optimizer ended up performing the best for the GRU models as well, achieving the highest validation accuracy of 55.1% as seen in Model 14. As such, the Adam optimizer will be used moving forward.

### Step 3: Reducing Overfitting

|  |  |  |  |
| --- | --- | --- | --- |
| **Model** | **Techniques Applied** | **Epoch where model overfits** | **Validation accuracy** |
| 17 | None | Nil | 55.6% |
| 18 | Only L2 regularization (0.001) | Nil | 52.8% |
| 19 | Only Regular Dropout (0.1) | Nil | 54.7% |
| 20 | Only Recurrent Dropout (0.1) | Nil | 55.8% |
| 21 | Recurrent Dropout + Regular Dropout | Nil | 53.8% |
| 22 | Regular Dropout + L2 Regularization | Nil | 51.7% |
| 23 | Regular Dropout + L2 Regularization | Nil | 52.1% |
| 24 | Recurrent Dropout + Regular Dropout + L2 Regularization | Nil | 50.5% |

Table 3.11 – Determining best combination of techniques to reduce overfitting

Table 3.11 shows the results obtained from testing 8 different GRU Models. Since Model 17, the model with no overfitting techniques applied has not shown signs of overfitting yet, the models in this step were trained for 40 epochs to see if the techniques applied had any effect in reducing the overfitting issue by extending the training process. However, even after running the models for 40 epochs, none of the GRU models seemed to overfit, but rather stagnate at the same validation accuracy after a certain point, suggesting that the GRU models take very long to converge and the accuracy is unlikely to increase. With that being said, by using recurrent dropout at a value of 0.1, a validation accuracy of 55.8% is obtained, which is the highest yet for the GRU models. Therefore, Model 20 with recurrent dropout will be used as the final GRU model.

### Rerunning GRU Model 20 until it overfits

Similar to the developed LSTM model, the Early Stopping callback feature is used for GRU Model 20 to train the model until it overfits.

Chart

Description automatically generatedGraphical user interface

Description automatically generated with medium confidence

Figure 3.4 – Performance of the developed GRU model

As seen from the Figure, similar to the LSTM model, the GRU model stagnates at around epoch 40, and remains at a constant accuracy all the way to epoch 100 without overfitting. In this case, after training for 100 epochs, the developed GRU model managed to obtain the highest validation accuracy of 56.3% in comparison to other GRU models thus far.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Model** | **Number of layers** | **Number of recurrent units** | **Batch Size** | **Optimizer** | **Technique to reduce overfitting** | **Validation Accuracy** |
| Baseline | 1 | 8 | 128 | RMSprop | None | 40.0% |
| Final | 1 | 64 | 512 | Adam | Recurrent Dropout | 56.3% |

Table 3.12 – Best combination of hyperparameters

With reference to Table 3.12, at a final validation accuracy of 56.3%, GRU Model 17 was able to improve by a sizeable amount compared to the baseline GRU model.

## Experimenting and building Conv1D models

A 1D Convolution layer, or what is known as a temporal convolution layer, is a layer that creates a convolution kernel that is convolved with the layer input over a single temporal dimension to produce a tensor of outputs. Just like how 2D Convolution layers are used for computer vision, 1D Convolution layers are able to process sequences by treating time as a spatial dimension, like how the height or width of an image is treated in a 2D convolution process. While Conv1D layers have generally been successful in other natural language processing tasks such as timeseries forecasting or text classification, a simple experimentation across a few Conv1D models will be tested in this section, to see if the Conv1D models are able to offer a fast alternative to RNN variants for sequence generation problems.

Text

Description automatically generated

Figure 3.5 – Code to instantiate a Conv1D model

Figure 3.5 shows the code to instantiate a Conv1D model. In this example, two Conv1D layers were used. In each layer, ‘filters’ was used to specify the dimensionality of the output space (i.e. the number of filters in the convolution). In addition, ‘kernel\_size’ was used to specify the length of each 1D convolution window. In between each layer, a 1D Max Pooling layer is used. After the convolution process, a 3D tensor is returned. Subsequently, in order for the dense output classifier to work, the Flatten layer is used to convert the 3D tensor to a 2D output.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Model** | **Number of layers** | **Number of recurrent units** | **Batch Size** | **Optimizer** | **Technique to reduce overfitting** | **Validation Accuracy** |
| 1 | 2 | 32 | 512 | RMSprop | None | 22.6% |
| 2 | 2 | 64 | 128 | RMSprop | None | 20.2% |
| 3 | 3 | 64 | 32 | Adam | None | 21.8% |

Table 3.13 – Best combination of hyperparameters

The process is repeated 2 more times and 3 Conv1D models were built to experiment and see the results. As seen from Table 3.13, the results were not promising at all, with an average of around 21% validation accuracy across 3 models, suggesting that in the case of text generation, 1D Convnets are generally not a suitable architecture to be used.

## Developing the Best Model and Analyzing Temperature

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Model** | **Number of layers** | **Number of recurrent units** | **Batch Size** | **Optimizer** | **Technique to reduce overfitting** | **Validation Accuracy** |
| LSTM | 1 | 128 | 128 | Adam | Recurrent Dropout | 58.9% |
| GRU | 1 | 64 | 512 | Adam | Recurrent Dropout | 56.3% |

Table 3.14 – Comparing the LSTM and GRU models

After experimentations ranging over 40 models and 3 architectures, the best LSTM and GRU models were obtained. Table 3.14 shows the results of both the best LSTM and GRU models. From the table, it is clear that the LSTM model is the better performing model of the two, making it the best model and architecture for text generation problems. To justify, this could largely be a result of **gates** that LSTM networks utilize. In LSTM networks, gates help to decide what data to be ignored and what to be fed-forward for the training. Using the three gates: forget gate, input gate and output gate, LSTMs are able to only allow important inputs to modify the existing information, making the network a robust network for text generation problems by storing useful contextual information from the past to aid the model’s learning process.

### Bidirectional LSTMs

As a last means of experimentation, the LSTM layer in the LSTM model was replaced with a Bidirectional LSTM layer to see if the implementation of Bidirectional traversal in the network would result in the model learning better. For clarification, a **Bidirectional RNN (BRNN)** is an RNN that has an additional hidden layer to accommodate backward training process. In other words, at any given time, both the forward and backward hidden states are updated simultaneously, which could be useful in text generation problems, where the model might need to understand the contextual semantics of the words that are before and that precede a sequence in order to learn better. In fact, one example of a popular Pretrained Language Model (PLM) that includes bidirectionality is BERT, which stands for Bidirectional Encoder-Representations from Transformers.

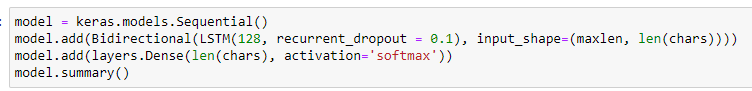


Figure 3.6 – Code to instantiate a bi-LSTM model

Chart, scatter chart

Description automatically generatedChart, line chart

Description automatically generated

Figure 3.7 – Performance of the bi-LSTM model

Figure 3.7 shows the performance after replacing the LSTM layer with a Bidirectional layer. In this case, it can be seen that the Bidirectional LSTM for this set of hyperparameters did not manage to perform better than the final LSTM model, only achieving a validation accuracy of 56.7%. This could be either because this set of hyperparameters in combination with a Bidirectional LSTM simply did not work well together, or because adding Bidirectionality in this case could be an unnecessarily element that overcomplicates the problem.

### Temperature Analysis

As mentioned, the final LSTM model is the best model developed thus far. In this last step, temperature analysis will be performed on the LSTM model through text generation temperature sampling, to observe the patterns of the model and how the model progressively learns.

Text

Description automatically generated with medium confidence A picture containing text

Description automatically generated

Figure 3.8 – Temperature observations from epoch 1 of final model

Figure 3.8 shows the text generated from epoch 1 of the final LSTM model. From the figure, it can be seen that lower temperature values usually result in more structured and coherent sentences; albeit the generated text is highly repetitive and predictable. On the other hand, a higher temperature value, for example 1.0 and 1.2, generates more unstructured but interesting text sequences. As seen from the text, the punctuations seem to be randomly slotted in the middle of each sentence, and although words like “waifed” and “exaltly” do not particularly exist in the vernacular, they seem like somewhat plausible words that could be invented by the model.

Text

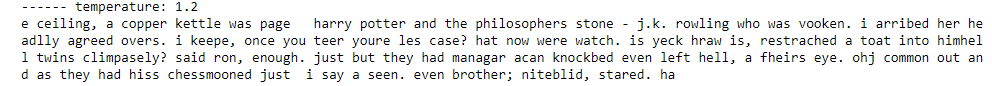
Description automatically generated with medium confidence 

Figure 3.9 – Temperature observations from epoch 20 of final model

Just for comparison, Figure 3.9 shows the temperature observations from epoch 20 of the final model. In the figure, the accuracy is shown to have increased to 61%, and in terms of the observations, most of the logic is around the same – higher temperature = more interesting. However, there are generally a larger percentage of words in each text sequence to be coherent English words, as a result of the higher accuracy in this instance.

# Use the Developed Model to Generate Texts

## Recording Input

Now that the developed sequence generator model is obtained, the model can finally be used to generate text. In this section, text will be generated by recording user input, encoding the input and feeding the input for the model to continue the sequence by generating new text.

Graphical user interface, text, application

Description automatically generated

Figure 4.1 – Recording user input

First and foremost, user input is recorded as seen in the Figure above. Input validation is added such that if the user enters an input longer than 60 characters, the user will be prompted to enter the input again.

Text

Description automatically generated

Figure 4.2 – Function to cleanse input

Thereafter, the input is cleansed using a function ‘cleanse’. The function first checks if the characters in the input exist in the vocabulary list ‘chars’ (i.e. no special characters), thereafter, the input string is manually padded by adding empty spaces at the start of the string until the input is 60 characters long.

## Generating Text

Text

Description automatically generated

Figure 4.3 – Function to generate text

Once the input is cleansed, the function ‘generate\_text’ was created to generate the text for the given input. Since after analysing temperature, the temperature value of 0.5 produced the most interesting results, the text was generated using a temperature value of 0.5. In the function, a loop with 400 iterations is ran, and at each state, the 60-character input is first encoded and fed to the model to make a character prediction. After a character prediction is made, the character will be appended to the end of the input string, and the first character in the input string will be removed to retain the length of 60-characters, and this process repeats until 400 characters (i.e. 400 iterations of the loop) is generated from the given input.

Text

Description automatically generated with low confidence

Figure 4.4 – Generating text

As seen from Figure 4.4, upon running the input text over the two functions, a 400-character sequence is generated with a temperature of 0.5. From the text, it can be seen that the context of the text is heavily related to Harry Potter, which is the text corpus the model was trained with. Despite not being meaningful and fully structured sentences, the text generated was still semi-coherent with a large percentage of words being actual English words.

# Summary

## Conclusion

In conclusion, by following the 3-step approach, that is, processing the data, developing the models and generating new text, this report demonstrated the process and methodology used for text generation problems. Firstly, as shown in the data processing section, this report showed how textual data was able to be processed by leveraging techniques such as the sliding window method and vectorization. In addition, following the universal machine learning workflow, this report also encapsulated the process of developing sequence generator models, and how RNN variants such as LSTM and GRU are robust networks for sequence processing, while architectures like 1D Convnets are not as efficient for this problem. Lastly, by using sampling strategies like temperature sampling in accordance with the developed language model, this report shows how user input can be ‘fed’ into a language model to generate text as well as the effect of temperature on the generated text sequences.

## Further Improvements

That being said, many things could have been considered or improved upon for Problem 2 of this assignment. This includes but is not limited to:

1. Training each model for more epochs to get a better visualization of the learning process for each model
2. Training more extensive combinations of hyperparameters and tuning other hyperparameters such as learning rate or the dropout percentages
3. Exploring more into other techniques and algorithms such as Weight/Batch normalization, which are normalization techniques, and Ranger, a synergistic optimization algorithm
4. Model checkpointing could be used as well since advanced RNN variants like the LSTM network is known for a slow training process
5. Trying out other text generation sampling strategies such as Greedy Search or Top-K Sampling
6. Experimenting with different window sizes to see if different sequence lengths helped the model to learn better