A black text on a white background

Description automatically generated

**Deep Learning Assignment**

Diploma in DS / IT

Oct 2023 Semester

**ASSIGNMENT 2 PROBLEM 2**

(40% of ASSIGNMENT 2)

8th Jan 2023 – 7th Feb 2024

**Submission Deadline:**

**Presentation: 7th Feb 2024 (Week 17),**

**Report: 7th Feb 2024 (Wednesday), 11:59PM**

|  |  |  |
| --- | --- | --- |
| **Tutorial Group** | **:** | **P01 / P02** |
| **Student Name** | **:** | Markell Wong |
| **Student Number** | **:** | S10242300 |

**Penalty for late submission:**

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

**NO** submission will be accepted after 17th Feb 2023 (Saturday), 11:59PM.

Table of Contents

[Overview 3](#_Toc158219500)

[Data Loading and Preprocessing 4](#_Toc158219501)

[Data Loading 4](#_Toc158219502)

[Data Preprocessing 5](#_Toc158219503)

[Develop the Sequence Generator Model(s) 7](#_Toc158219504)

[Metrics 7](#_Toc158219505)

[Model 1: LSTM Base Model 9](#_Toc158219506)

[Model 2: Data Augmentation 9](#_Toc158219507)

[Model 3: LSTM RMSProp Learning Rate 12](#_Toc158219508)

[Model 4: LSTM Batch Sizes 12](#_Toc158219509)

[Model 5: LSTM Layer Sizes 13](#_Toc158219510)

[Model 6: GRU Layers 15](#_Toc158219511)

[Model 7: GRU RMSProp Learning Rate 17](#_Toc158219512)

[Model 8: GRU Batch Sizes 18](#_Toc158219513)

[Model 9: GRU Layer Sizes 18](#_Toc158219514)

[Choosing Between LSTM and GRU 20](#_Toc158219515)

[Model 10: Adam Optimizer 21](#_Toc158219516)

[Model 11: Activation Functions 23](#_Toc158219517)

[Model 12: Kernel Initializers 26](#_Toc158219518)

[Model 13: Dropout Layers 28](#_Toc158219519)

[Final Model 29](#_Toc158219520)

[Use the developed Model to Generate Texts 30](#_Toc158219521)

[Sampling Techniques 30](#_Toc158219522)

[Stochastic Sampling 30](#_Toc158219523)

[Top K Sampling 31](#_Toc158219524)

[User Input 32](#_Toc158219525)

[Summary 33](#_Toc158219526)

# Overview

Objective: Implement a Recurrent Neural Network (RNN) to create an English language character generator capable of building semi-coherent English sentences from scratch, by building them up character by character.

For this problem, we will be using a complete version of J. K. Rowling’s famous book Harry Potter and the Prisoner of Azkaban as the model input. We look to create a model that can best mimic J.K Rowling’s writing style.

For model training, we will input sequences and the following characters so that the model can learn patterns between text inputs and potential character outputs. In data preprocessing, we will define the sequences in the train data and the following characters in the test.

As we want to improve our text generation, we will tune the model hyperparameters. The tuning process will follow the universal machine learning workflow where we start with a baseline model, scale up the model until overfitting and then regularize the model by introducing implicit and explicit regularization as well as tune model hyperparameters.

For my model tuning process, I did as follows:

A graph paper with text and arrows

Description automatically generated

Figure 1: Tuning Process

For the tuning process, I built up two different kinds of model layers, LSTM (Long-Short Term Memory) and GRU (Gated Recurrent Units). I trained the models separately as both have different architectures (explained in below sections) and optimal hyperparameter values will be different for both. For both models, I will first scale it up, tune the training hyperparameters, namely learning rate and batch size, the model complexity, and if the model is overfitting, experiment with data augmentation.

Afterwards, I will choose the best model out of the 2. I will then experiment with optimizers, activation functions, and kernel initializers and if the model is still overfitting, introduce explicit regularisation.

After tuning, I will try different sampling techniques to see which gives me the best quality of generated texts. I will also enter a user input to see how my model works on real-life data, where I will analyse the output.

# Data Loading and Preprocessing

## Data Loading

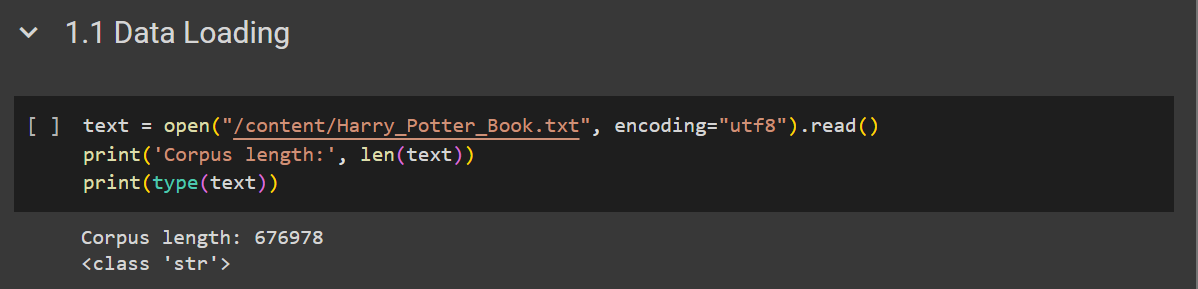


Figure 2: Data Loading

I first called and loaded the Harry\_Potter\_Book.txt provided for this assignment. The encoding used to read the file is “utf8” which is a widely used character encoding that represents most of the characters in the Unicode character set. It is a flexible encoding that can encode a diverse range of characters such as non-ASCII characters.

I also decided not to .lower() the text to keep upper case letters. Upper case letters are used to denote the start of sentences, proper nouns and abbreviations as well as display and highlight emotion in text (eg: Happiness: I LOVE THIS!). By keeping them, we can better mimic and train the model on the intricacies of which JK Rowling (the author) wrote the book.

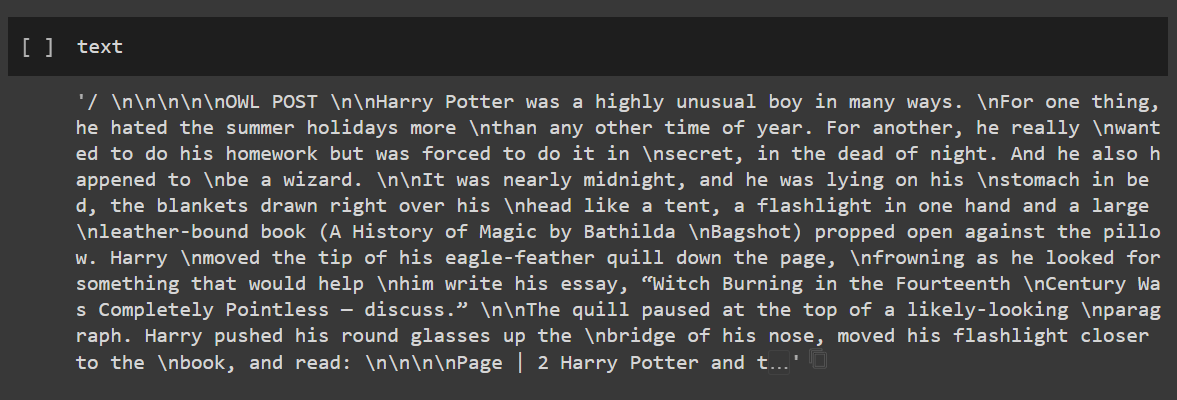


Figure 3: Example of text

As we can see, “\n” is used to denote when a new line is used,

## Data Preprocessing

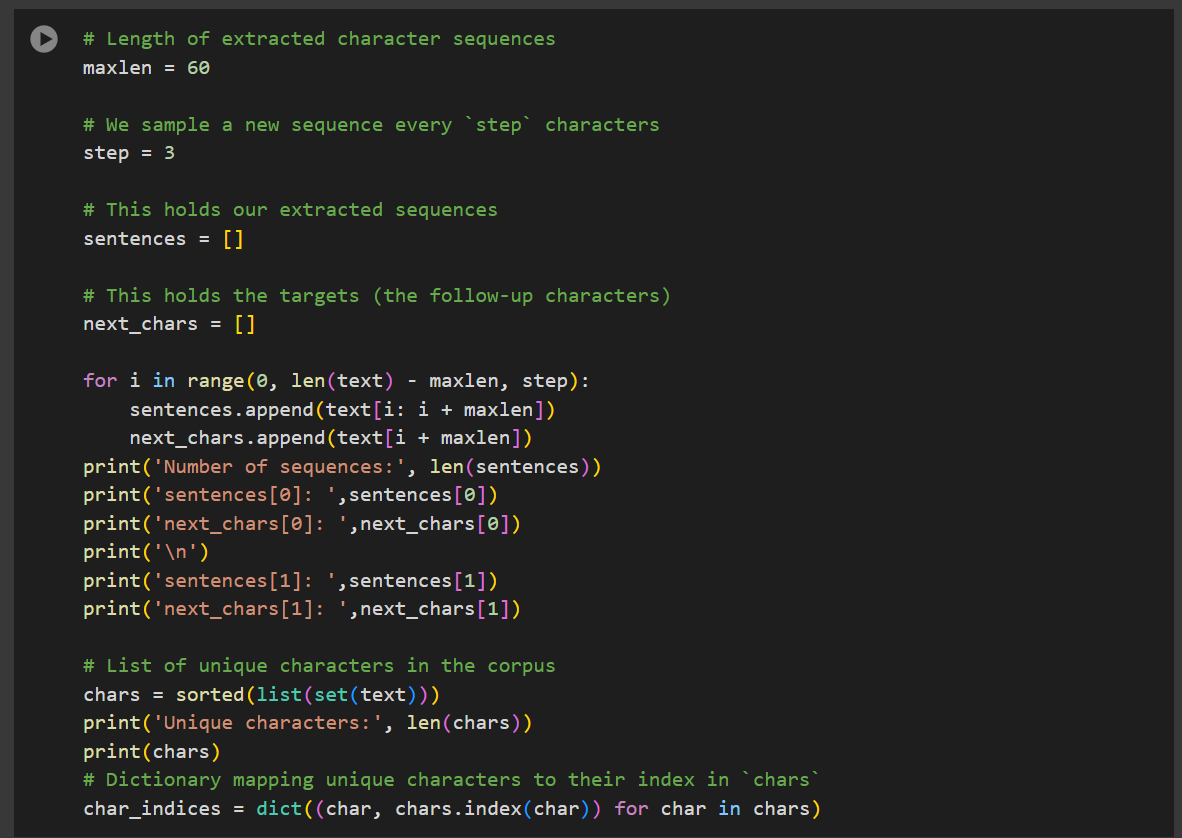


Figure 4: Getting sequences

Next, I converted the text into fixed-length sequences. Essentially, from the entire text, I want to get a segment of the text to create input data for the model. Here, I defined the maximum length of the sequences by setting maxlen, hence the maximum length of the sequences is 60. I also defined the step size as 3, which refers to the interval I slide a window forward over the input sequences for subsequent sequence extraction.

Let’s say my text is “I Love Bak Kut Teh” and I set my sequence length as 5 and step size as 2. When I extract sequences, my first sequence will be “I Lov”, my second sequence will be “Love ” and my third sequence will be “ve Ba”.

Basically, for every sequence, I use the starting letter that is 2 indexes ahead of the previous starting letter (“I” to “L” to “v”), sliding the window 2 indexes ahead. This allows us to generate multiple sequences from a base sequence, increasing the amount of input data for the model.

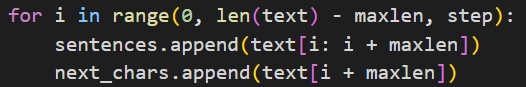


Figure 5: Creating sequences and target values

Here, I extract sequences from the text with maxlen and step size using the technique mentioned in the example. Each sequence is stored in a list and the next character following the end of the sequence is stored in the next\_chars list, this is our target value for predictions. For example, let's take the sequence “I Lov”, the next character is the character afterwards, which will be “e”.

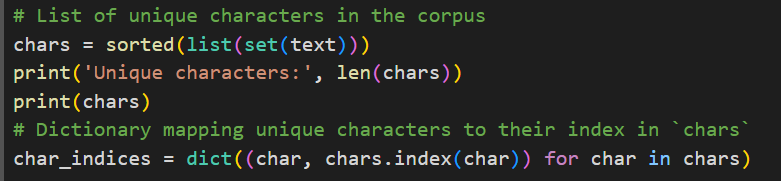


Figure 6: Unique characters

Next, I find the unique characters and the number of them. After looking through the unique character dictionary, I noted that there were characters such as asterisks (\*), vertical bars (|) and double slashes (\\). While they are unusual, I similarly decided not to remove any of them to preserve the writing style. I had 86 unique characters.

A computer screen with text and images

Description automatically generated

Figure 7: Vectorisation and One hot encoding

Next, I vectorised the sequences and one hot encoded the characters in the sequences. Firstly, I created two arrays x and y to store the vectorized representations, I initiated the arrays with False values first. The x array has 3 dimensions as we need to represent each character in each sequence, the y array only has 2 dimensions as there is only one character prediction for each sequence and we only need to encode the character.

Next, I loop over each sequence and one hot encode each character in the sequence for x. Essentially, the index corresponding to the character in char\_indies is set to 1 for each position in the sequence.

For example, I have the text “abc”, I will then map the unique characters to an index in a dictionary, char\_indices = {a: 0, b:1, c:2}. Then, I one hot encode the characters by replacing the False value of the index of the character with 1 or True. Let's say my sequence is “ab”, my encoding will then be:

[ [True False False] # a is the first index

[False True False] ] #b is the second index

We do the same thing for y, but instead of each character of the sequence, we only need to encode one character which is the predicted character of the sequence. We one hot encode as we want each character to have a unique vector with a length equal to the total number of unique characters and to convert the text into a format the model can understand.

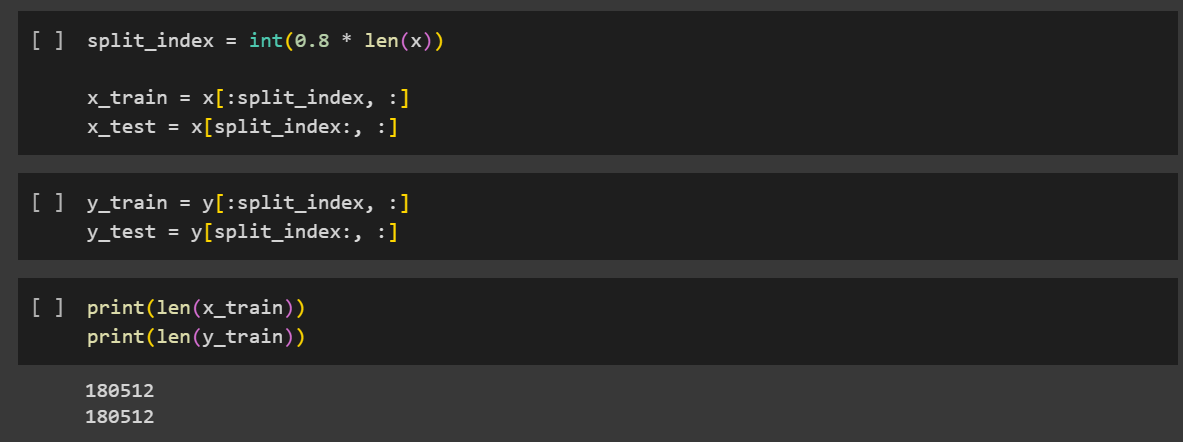


Figure 8: Train test split

Here, I did a train test split in case I wanted to use a test dataset for final evaluation separate from the training. I split the data sequentially so the first 80% of sequences will be used as my train and the last 20% is the test. As we are doing character generation, the context of previous characters is crucial, and the model needs to learn the patterns of characters that follow each other in a sequence hence I did a sequential split.

During model training, a further validation split is done, this is used to see how well the model generalises to different sequences and provide some insight into how I should tune the model.

# Develop the Sequence Generator Model(s)

## Metrics

For training:

Accuracy: While accuracy may not be the best metric for language modelling tasks as we have imbalanced frequencies of each character, I used it just to get a sense of how many characters the model was able to predict correctly and to measure the overall performance of the model.

Loss: Cross-categorical entropy, evaluates how well the model can predict the characters by measuring the error between predicted and actual characters.

For generated texts:

BLEU Score: Evaluates the quality of generated texts, score is between 0 and 1 where the higher the score the better.

BLEU uses n-grams, which in this context, is a set of n consecutive characters in a word/sentence. For example, for the word “Sun”, the unigram (1-gram) would be “S”, the 2-gram would be “Su” and so on. We then measure the number of characters in the generated text that also occurred in the original text using clipped precision.

Original text: Walter

Generated text: worry

Here, we compare each character of the original and generated text. If the character matches, it is considered correct. We also limit the count of each correct character to the maximum frequency the character occurs in the original text.

|  |  |  |
| --- | --- | --- |
| Character | Matched predicted count | Clipped Count |
| w | 1 | 1 |
| o | 0 | 0 |
| r | 2 | 1 |
| y | 0 | 0 |
| Total | 3 | 2 |

The character “r” appeared twice in the generated text but since only appears once in the original, the count is reduced to one. Hence, the clipped precision = ( Clipped num of correctly predicted characters / total predicted characters ), which will be 2/5 for the above example.

BLEU then finds the clipped precision for n-grams and combines them using geometric average precision.

A black text on a white background

Description automatically generated

Lastly, we compute a brevity penalty to penalize generated texts that are too short which can be misleading as it encourages the model to output fewer characters to get a high score. This shouldn’t be a problem as we will define the number of characters to be generated.

BLEU is then calculated by multiplying the Brevity Penalty with the Geometric Average Precision. While BLEU provides an objective way of evaluating texts, it does not consider the meaning of the text generated or the order of characters.

Hence, generated text evaluation will also use human assessment (me😊) to look at whether the text formed is coherent and meaningful as well as evaluate the creation of words.

## Model 1: LSTM Base Model

A screenshot of a computer program

Description automatically generated

Figure 9: Base Model

My base model was relatively simple, creating a baseline for performance of which I can compare my subsequent models.

The model only had 1 layer with 128 nodes. It used RMSProp with a learning rate of 0.01, kernel initializer was the default glorot uniform and activation function was the default TanH (hyperbolic tangent). I ran the model for 20 epochs at a batch size of 64.

A graph of a line and a line

Description automatically generated with medium confidence

Figure 10: Base LSTM

The validation accuracy and loss hovers at around 0.575 and 1.55 respectively. We can see some overfitting in the model as seen by the deviation of train and validation curves and a slight increase in validation loss at epoch 17 while train loss continues to decrease.

## Model 2: Data Augmentation

As our base model is experiencing a bit of overfitting, I experimented with some data augmentation of the input text.

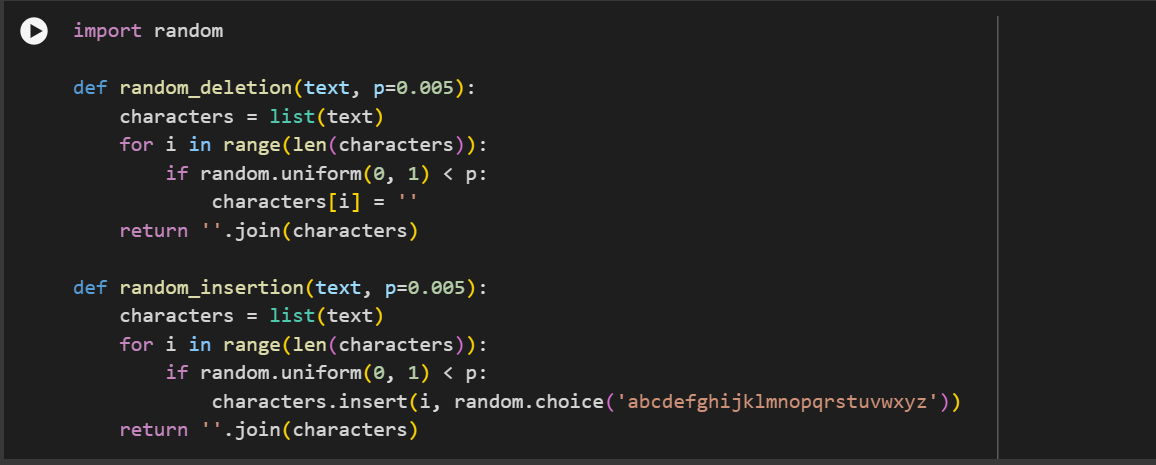


Figure 11: Text augmentation

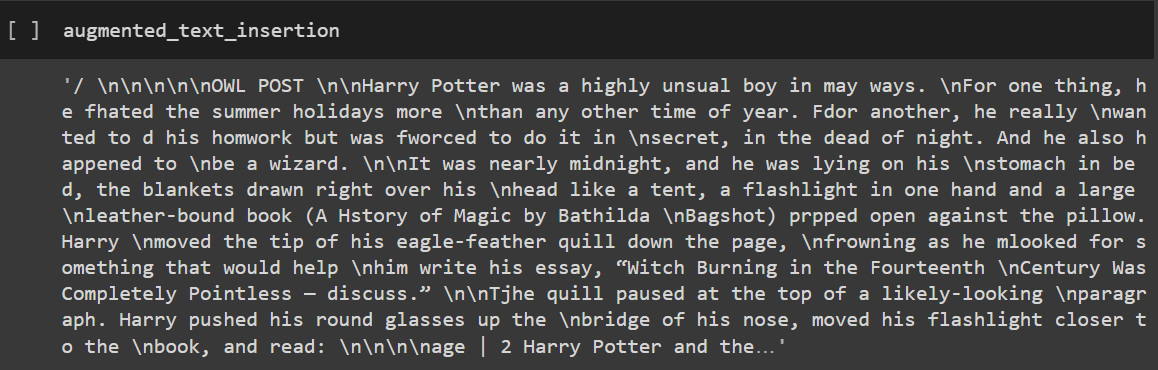


Figure 12: Text augmentation output

Here, I tried randomly deleting and inserting characters into the input text. In Figure 12, we can see the effect of the insertion and deletion (“hated” -> “fhated”, “History” -> “Hstory”). By introducing text augmentation, I looked to introduce random typos and spelling errors to simulate noisy data. This would hopefully make the model more robust to the input variations and when unseen data is introduced, be able to generalise better to the new data, reducing overfitting.

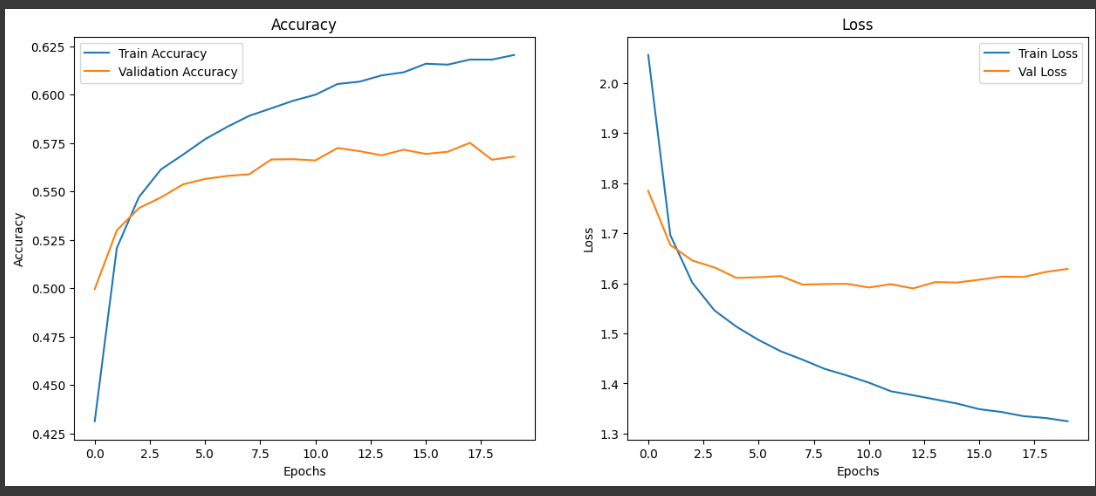


Figure 13: Text augmentation (p = 0.01)

Unfortunately, the model saw a decrease in validation performance (hovering around val\_accuracy 0.568 and val\_loss 1.62) without a substantial decrease in deviation of train and validation curves. This could be due to the strength of text augmentation being too great, introducing too much noise that negatively impacted learning.

A graph of different colored lines

Description automatically generated with medium confidence

Figure 14: Text augmentation (p = 0.005)

Here, I halved the augmentation strength with p-value of 0.005, to slightly reduce the noise introduced. While the deviation between train and validation was slightly reduced, so was the validation performance (around val\_accuracy 0.55 and val\_loss 1.67).

In both p-values, the deviation was barely reduced while validation performance suffered showing that the model was not learning the variations well. This could mean that introducing typos were not suitable for more precise generation tasks such as characters. Hence, I did not use text augmentation.

## Model 3: LSTM RMSProp Learning Rate

Next, I tuned the learning rate, a crucial training hyperparameter. Learning rates determine the “size” of steps taken during optimization. If the learning rate is too high, the model will overshoot the minimum and fail to converge. If it is too small, the model will take very long to converge.

A group of graphs showing the value of a number of percent

Description automatically generated with medium confidence

Figure 15: Learning rates

Here, I decided to continue using LR = 0.01. LR = 0.1 is too large a learning rate hence we see large fluctuations in train and validation curves showing that weight updates are too drastic. Between the remaining 3, LR = 0.01 had the highest val\_accuracy without having the largest variance between train and validation curves.

## Model 4: LSTM Batch Sizes

Next, I tuned the batch size. Batch size determines the number of sequences utilized in one iteration which affects the model’s convergence speed and stability during training. Batch sizes also provide implicit regularization.

Smaller batch sizes can result in the model having noisier gradient estimates as the batch sizes do not fully represent the gradients of the entire dataset. This noise introduces variability and reduces the model’s tendency to fit training data too closely. On the other hand, larger batch sizes are more representative of the dataset’s gradients and smaller differences between iterations can lead to smoother training processes.

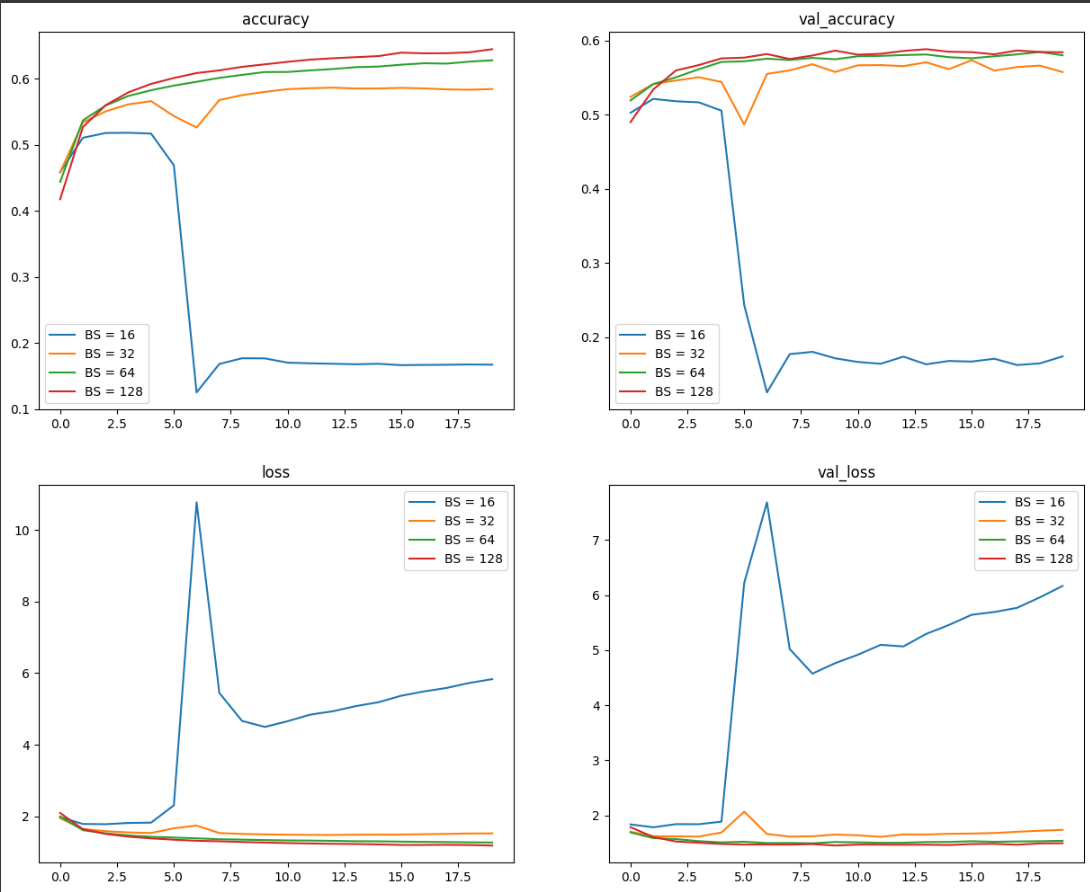


Figure 16: Batch Sizes

After testing multiple batch sizes, I decided to continue using batch size 64. BS = 16 seemed to have introduced too much noise between batches, causing training and validation to fluctuate greatly. Similarly, BS = 32 also sees more fluctuations and poorer model performance. Between BS = 128 and BS = 64, BS = 64 had lesser deviation between train and validation curves while maintaining good validation scores.

## Model 5: LSTM Layer Sizes

Next, I tuned the number of nodes within the LSTM layer. The number of nodes influences the model’s capacity to learn and represent complex patterns in sequential data. While increasing the number of nodes can result in better model generalisation to new sequences, it also increases the risk of overfitting as the model learns and adapts too closely to the training data. Hence, we will need to tune the layer size to see what is most optimal for our model.

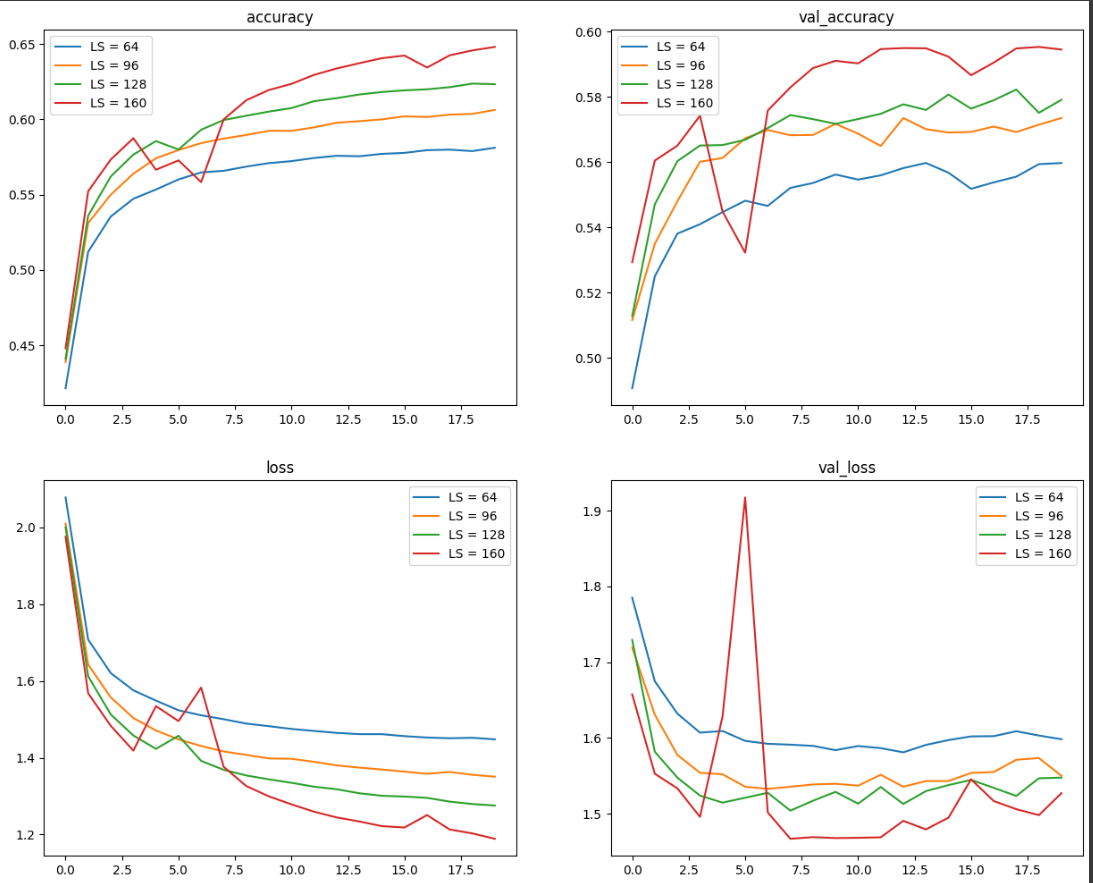


Figure 17: Layer Size

Here, I noticed that LS = 160 had quite a significant improvement of validation performance compared to the other layer sizes (hovering around val\_accuracy 0.595, val\_loss 1.46). However, I also noted that there are increased fluctuations of train and validation curves for LS = 160, showing that the training process was more unstable. This could just be due to the weight initialization for that specific run which resulted in larger weight updates.

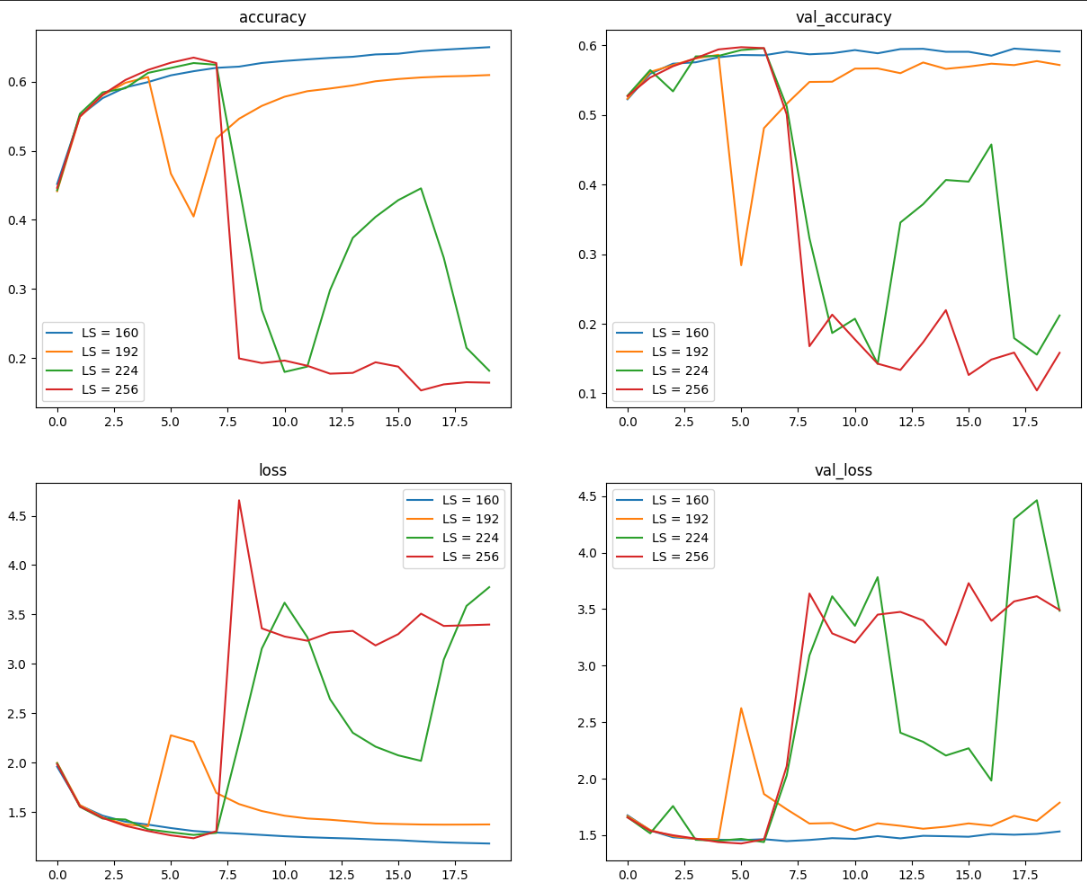


Figure 18: Even larger layer sizes

Next, I tried even larger layer sizes to see if model generalisation can still be improved. However, the larger layer sizes showed extreme fluctuations in training and validation curves. Furthermore, LS = 224 and 256 saw significant deterioration of accuracy and loss after epoch 8. LS = 192 saw worse performance for both training and validation compared to LS =160. It is possible that further increasing the layer sizes led to model learning data noise too well, causing very poor generalisation of the model overall.

Hence, I only increased the layer size from 128 to 160 due to a large validation performance improvement.

## Model 6: Base GRU Model

LSTMs VS GRUs

A diagram of a gate

Description automatically generated

Figure 19: LSTM Architecture

[<https://towardsdatascience.com/a-brief-introduction-to-recurrent-neural-networks-638f64a61ff4> ]

LSTMs use cell states which allow information to flow through LSTM cells with only minor linear actions through the three gates. Cell states serve as the long-term memory of the LSTM. The three gates are forget, input and output and they work as filters to determine which information is kept or forgotten.

The forget gate decides how much long-term memory is kept, where a sigmoid function is used to determine the importance of the cell state between 0 and 1. The input gate decides which information is added to the cell state and the output gate decides which part of the cell state builds the output. The hidden state of the current time step is determined by the output gate and a tanh function.

A diagram of a gate

Description automatically generated

Figure 20: GRU Architecture

[<https://towardsdatascience.com/a-brief-introduction-to-recurrent-neural-networks-638f64a61ff4> ]

GRUs get rid of the cell state in LSTMs and use the hidden state to transfer information. Compared to LSTMs, it only has two gates, the reset gate and update gate. The update gate performs similarly to the forget and input gate of LSTMs, deciding what information to throw away and what new information to add. The reset gate is used to decide how much past information to forget.

GRUs have lesser tensor operations than LSTMs and are faster to train. As both are able to capture long-term dependencies, the difference between the two may not be much. However, GRU’s simpler structure may make it more suitable for the task and data, hence we can test it out.

A screenshot of a computer program

Description automatically generated

Figure 21: Base GRU Configuration

Here, I swapped the Base LSTM model for GRU layers. The hyperparameter values are the same: 1 layer, 128 nodes, RMSProp with 0.01 learning rate, default glorot uniform kernel initializer, default TanH activation function, ran for 20 epochs and 64 batch size.

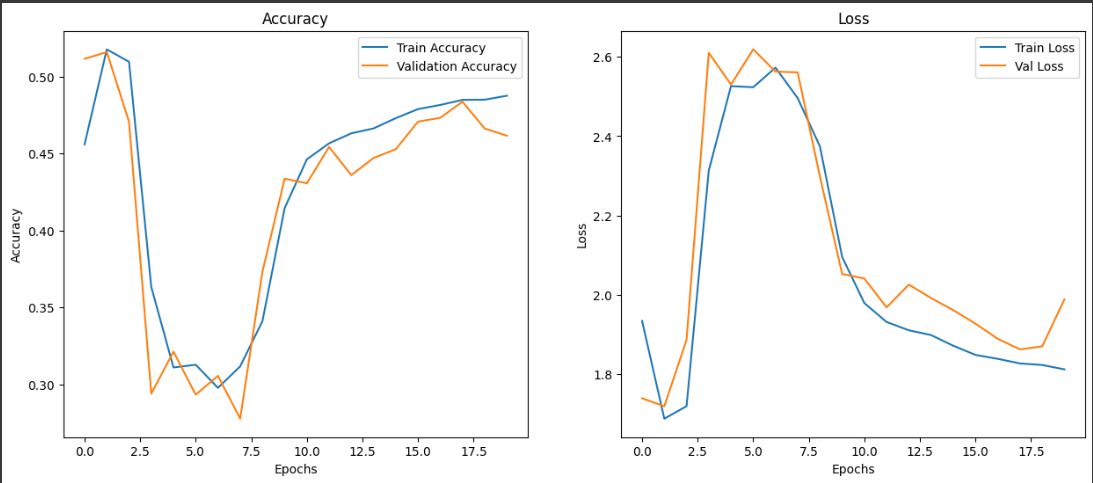


Figure 22: GRU Base Model

We can see that our GRU training and validation curves exhibit very unusual behaviour. The train and validation curves seem to fluctuate wildly, with the validation curve sticking close to the train curve at every epoch.

This could be a result of extremely high learning rates as gradient steps in the gradient descent are too large, causing weight updates to be drastic hence the extreme fluctuations seen in Figure 22.

## Model 7: GRU RMSProp Learning Rate

Hence, I decided to tune the learning rate first.

A group of graphs with different colored lines

Description automatically generated

Figure 23: GRU RMSProp Learning Rates

From Figure 25, we can see that all the learning rates other than the smallest learning rate LR = 0.001 show a high degree of fluctuations in both training and validation. LR = 0.001 showed a much more stable training process and the best model performance amongst all the learning rates (hovering around 0.566 val\_accuracy and 1.55 val\_loss). Hence, I decreased the learning rate from 0.01 to 0.001.

However, I also noted that there is some overfitting in the model as seen from the deviation between train and validation curves.

## Model 8: GRU Batch Sizes

From Figure 23, we could see there was some overfitting, hence we can tune batch sizes to see if we can reduce it and improve model generalisation on unseen data.

A group of graphs with numbers

Description automatically generated

Figure 24: Batch Sizes

From Figure 24, we can see BS = 64 managed to hit the highest validation accuracy and lowest validation loss among the batch sizes at around epoch 12 (val\_acc 0.5747, val\_loss 1.5161). However, BS 128’s highest val\_accuracy and lowest val\_loss was quite close to BS 64 (val\_acc 0.513, val\_loss = 1.526) at epoch 20 with lesser deviation of train and validation curves.

Hence, I decided to adopt a larger batch size from 64 to 128. Although we sacrifice a bit of validation performance, there is less overfitting.

## Model 9: GRU Layer Sizes

Next, I tuned layer size to see if I can further improve model generalization. Hopefully, by increasing layer size, we can increase model complexity, allowing the model to generalize better.

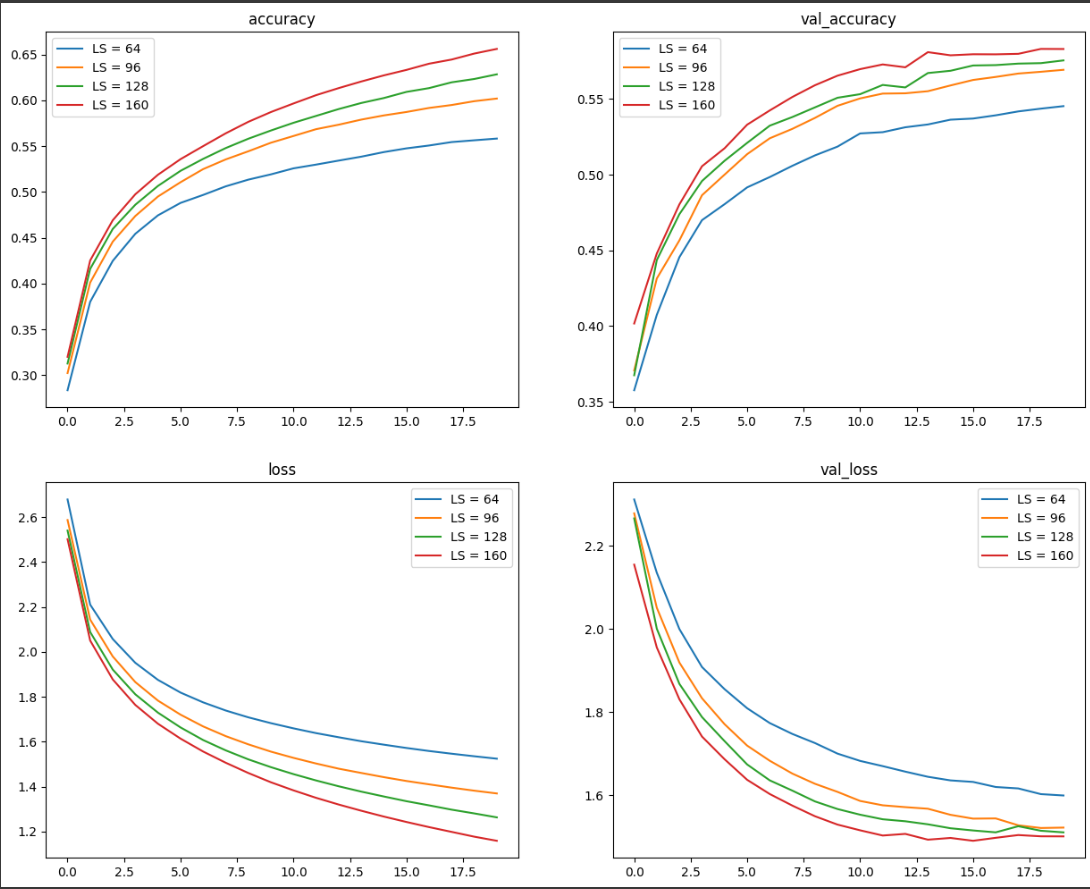


Figure 25: Layer Size

In Figure 25, LS = 160,128 and 96 had close peak validation accuracy and loss scores.

LS = 128, Peak: Epoch 20, val\_acc = 0.5755, val\_loss = 1.5101

LS = 160, Peak: Epoch 20, val\_acc = 0.583, val\_loss = 1.5005

LS = 96, Peak: Epoch 20, val\_acc = 0.5693, val\_loss = 1.5216

As we can see, all three layer sizes have small differences between validation scores but with varying scales of overfitting. While, LS = 160 had the best performance, the distance between train and validation curves was the largest. Similarly, LS = 96 has the worst performance of the three, but the least variance.

Hence, I continued to use LS 128 as it balances the performance and variance tradeoff and we do not sacrifice as much as performance for a smaller variance.

## Choosing Between LSTM and GRU

A graph of a line and a line

Description automatically generated with medium confidence

Figure 26: Final LSTM Model

A comparison of a graph

Description automatically generated with medium confidence

Figure 27: Final GRU Model

When we compare the two models, LSTM was slightly better in terms of performance. LSTM peaked at epoch 11, with a val\_acc 0.5973 and a val\_loss 1.4600. GRU peaked at epoch 20, with a val\_acc 0.5805 and val\_loss 1.5006. However, LSTM seems to have a less stable training process than GRU as seen by the fluctuations in train and validation curves.

In the end, due to the better performance, I chose to use LSTM. To increase stability in LSTM, we can try different optimizers, kernel initializers etc.

## Model 10: Adam Optimizer

Adam is another adaptive optimization algorithm that adjusts learning rates during training, similar to RMSProp.

Adam adapts its learning rates based on the mean and uncentered variance of the gradients, using momentum to accelerate optimization. Adam utilizes momentum by using the moving average of the gradient and its squared values. RMSProp, on the other hand, only considers the uncentered variance of the gradients and does not use momentum.

While it is difficult to outright state that using Adam optimizer will improve model performance, it is generally considered to be more robust due to it considering mean and uncentered variance.

A group of graphs with different colored lines

Description automatically generated

Figure 28: Adam Learning Rates

From Figure 28, we can see that only LR = 0.001 has a smooth training process. The other learning rates see large fluctuations in both train and validation curves. The larger learning rates are too high, causing the weight updates to be extremely large, hence large fluctuations.

I also noticed that LR = 0.1 stops having a loss curve from epoch 11 onwards. This is due to the loss score being null and is also known as exploding gradient problem.

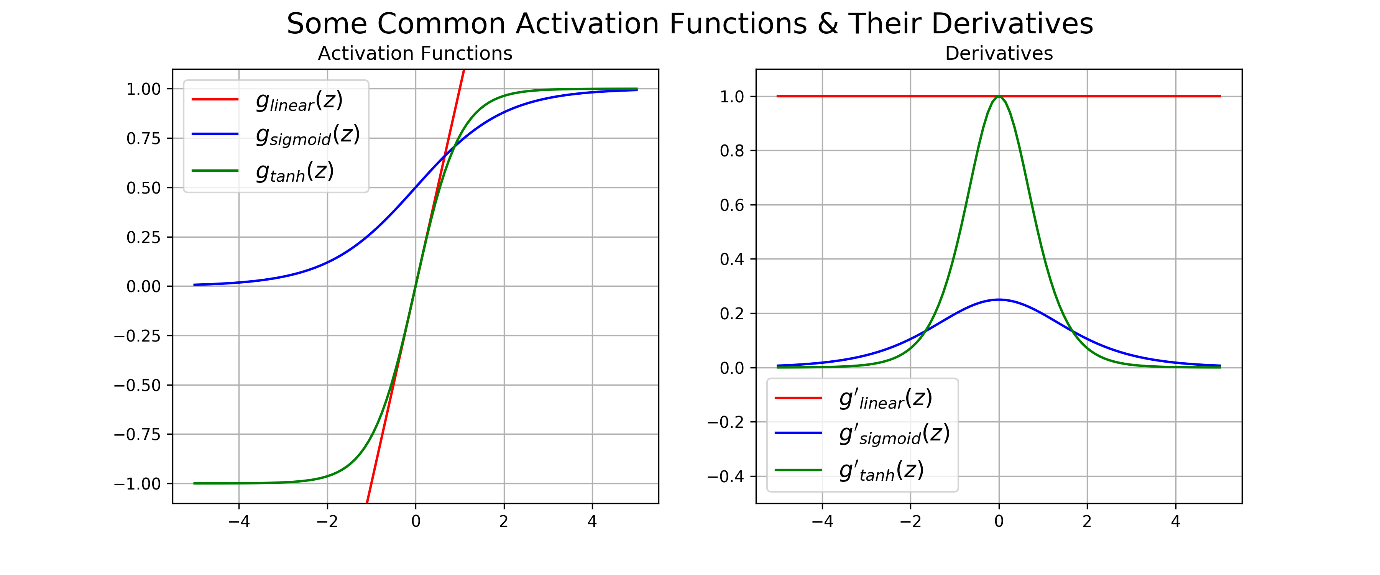


Figure 29: Activation functions and their derivatives

<https://dustinstansbury.github.io/theclevermachine/derivation-common-neural-network-activation-functions>

Essentially, in Figure 29, we can see that for TanH (the activation function we are currently using), more extreme inputs (eg x = 4 and -4), it saturates at -1 or 1 with a derivative very close to zero. Exploding gradients happen when activation functions are saturated (eg: x = 4), and gradient values increase dramatically, making updates to weights unstable. Eventually, the model parameters can get so large they diverge to infinity, causing the learning process to fail and resulting in the loss becoming null.

This shows that LR 0.01 was too high a learning rate and gradient values became too large and unstable, eventually causing exploding gradients.

Overall, Adam LR = 0.001 had a peak val\_acc 0.5660 and val\_loss 1.5204 at epoch 20, however, this is a poorer performance compared to RMSProp. This showed that RMSProp’s simpler calculations were more suited for the data. Hence, I continued to use RMSProp.

## Model 11: Activation Functions

Next, I experimented with activation functions. Activation functions are mathematical functions applied to the output of a node where they introduce non-linearity (for non-linear functions) to the network which allows the network to learn complex relationships better.

A graph of a function

Description automatically generated with medium confidence

Figure 30: Non-linearity

Non-linear functions allow the model to fit into non-linear relationships as seen in Figure 30, improving model generalization and its ability to adapt to new data. Activation functions are crucial as they influence the flow of gradients during backpropagation and convergence speed of the learning process. By default, LSTM layers use hyperbolic tangent, which is also a non-linear activation function.

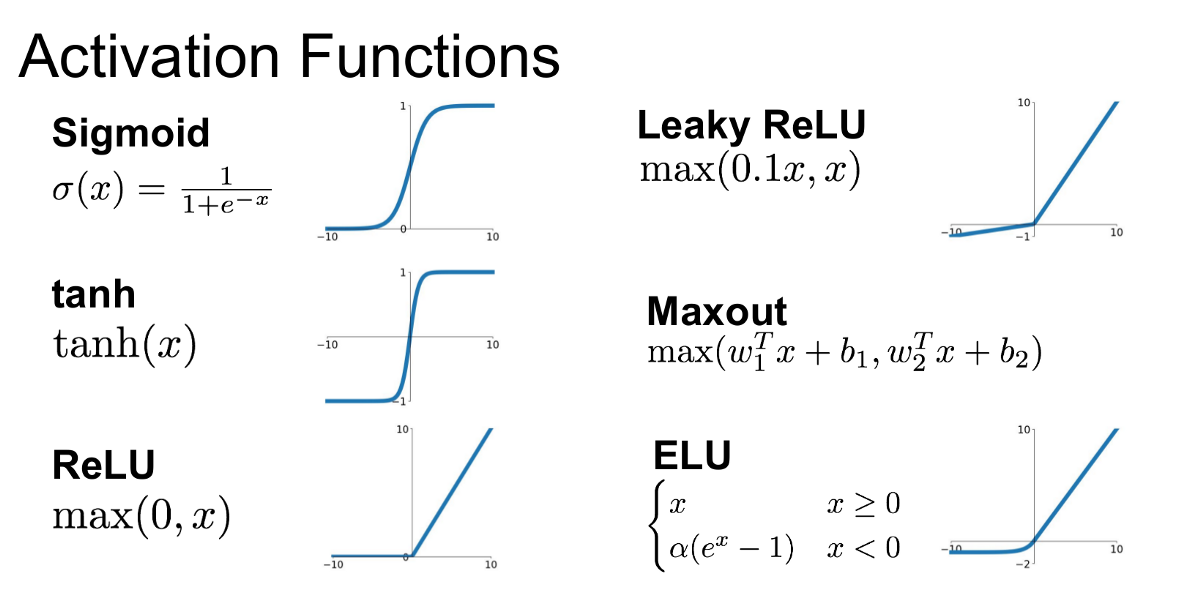


Figure 31: Activation Functions Curves

<https://medium.com/@shrutijadon/survey-on-activation-functions-for-deep-learning-9689331ba092>

TanH is commonly used in RNNs due to its zero-centred output. Tanh squashes the node outputs between -1 and 1, introducing zero-centering and helping with optimization by providing negative values. For this section, we can try different activation functions to see which is most suitable for the model.

**ReLU and Sigmoid**

As seen in Figure 31, ReLU is a piecewise linear function that will output the input directly if it is positive and negative values are mapped to zero immediately. Sigmoid squashes the node outputs between 0 and 1. The sigmoid function is a smooth curve, making the gradient flow during optimization smoother hence stabilizing the training process.

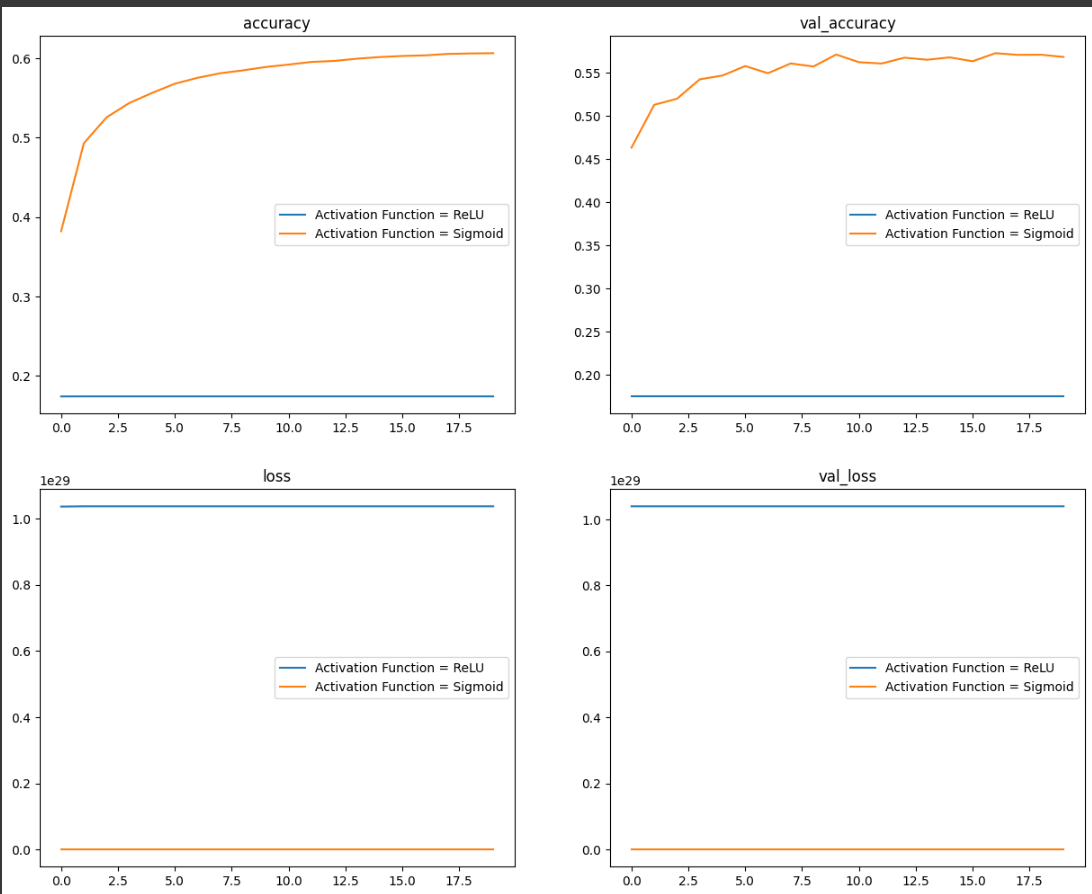


Figure 32: ReLU and Sigmoid

In Figure 32, we notice that accuracy and loss are straight lines at close to zero values for ReLU. This exhibits the dying ReLU problem where essentially a large gradient flowing through the ReLU node causes the weights to have a large negative bias, resulting in the node never activating again, with weights becoming zero. The node is unlikely to recover as the ReLU’s gradient at 0, is also 0, hence the gradient descent will not alter the weights.

For sigmoid, the peak scores are at epoch 17, val\_acc 0.5728, val\_loss 1.6308. However, sigmoid does not perform as well as TanH. This could be due to the sigmoid function not being zero-centred, causing weight updates to be sub-optimal.

**LeakyReLU**

LeakyReLU looks to counter the dying ReLU problem by introducing a small, non-zero gradient when the input is negative.

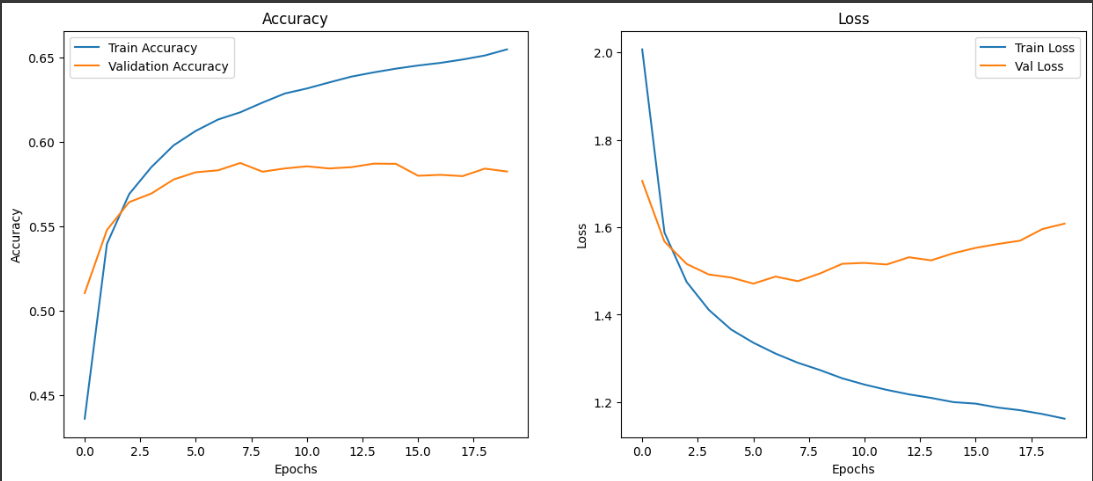


Figure 33: LeakyReLU

For LeakyReLU, the peak val\_acc was 0.5870 and val\_loss 1.5406 at epoch 15. We can also see overfitting starting from around epoch 9 as validation loss starts increasing while train loss continues decreasing. However, LeakyReLU was unable to outperform TanH.

**Conclusion**

Hyperbolic tangent still seems to be the most suitable optimizer for the model. This could be due to our hyperparameters tuning being more adjusted to TanH as it was our default activation function.

## Model 12: Kernel Initializers

When neural network models start training on the data, they will start with some weight values first and then iteratively update them to better values. Kernel initializers are essentially statistical distributions or functions to use for weight initialization.

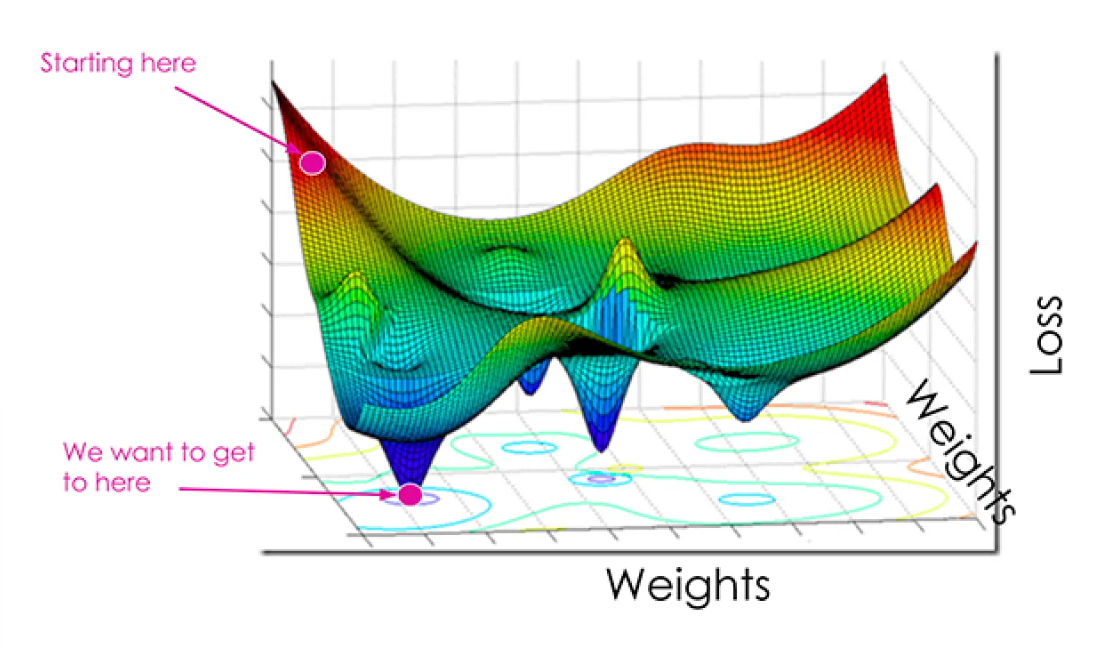


Figure 34: Loss Landscape

<https://www.v7labs.com/blog/cross-entropy-loss-guide>

Kernel initializers are crucial as they set the starting points of our model in the loss landscape. For example, in Figure 34, the yellow dot is the starting point of the model with a not-so-optimal weight initialization. After gradient descent, it ends up in the local minima (lower yellow point). If the hyperparameter values are not optimal, the model may take longer or never leave the local minima to get to the global minima.

Let's take the pink point as another starting point for a model with an optimal weight initialization. By having a better starting point, it can reach the global minimum faster and has a reduced chance of getting stuck in a local minimum.

Hence, by optimizing our model’s initialization position, we can improve the convergence speed, stability and overall performance of our model.

The default kernel initializer is glorot uniform. It looks to initialize the weights from a uniform distribution with the specific range that is determined by the size of input and output layers. The range of distribution is calculated as: range = sqrt(6 / (No of input neurons + No of output neurons)). Glorot uniform looks to make sure the gradients do not become too large or small during training, which can cause unstable training and poor convergence.

We can experiment with different kernel initializers to see if they are more suitable. I tried 3 other kernel initializers, HeNormal, Lecun Uniform and Orthogonal

HeNormal initializes the weights using values from a normal distribution with 0 mean and a standard deviation of sqrt(2/n) where n is the number of nodes in the previous layer. By initializing weights based on the previous layer, it helps to ensure the variance of activations remain constant across all layers of the model.

Lecun Uniform is similar to Glorot uniform but designed specially for networks with TanH activation functions. It initializes the weights uniformly in the range [-limit, limit], where limit is sqrt(3 / number of input nodes). It is typically used with activation functions that have saturation characteristics (such as TanH).

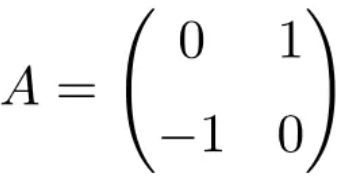


Figure 35: Example of orthogonal matrix

Orthogonal initializes weights of the layer with an orthogonal matrix. An orthogonal matrix is a square matrix where its rows and columns are orthonormal unit vectors. We use orthogonal initialization to help preserve the geometry of weight space during training. It is typically used in recurrent networks where we want to preserve the dynamics of hidden states.

A group of graphs with different colored lines

Description automatically generated

Figure 36: Kernel Initializers

Overall, glorot uniform had the highest peak performance of all the kernel initializers. There are some reasons for this such as HeNormal’s incompatibility with our current activation function (TanH) where initialized weights were too large for TanH, causing saturation. It is also possible that since we have been tuning hyperparameters with glorot uniform as our default, our hyperparameter values are now more suited for glorot uniform, hence the difference in performance.

Nonetheless, I continued to use glorot uniform. Here, I noted that our model still experiences some overfitting as seen from the variance of train and validation scores and fluctuations in validation curves showing that the model is fitting too closely to data noise.

## Model 13: Dropout Layers

Lastly, I introduced explicit regularisation to reduce the model overfitting. I tried dropout layers which randomly set a fraction of nodes in a layer to zero. By temporarily removing a percentage of nodes, we introduce noise in the learning process, making the model more robust and less sensitive to specific weights.

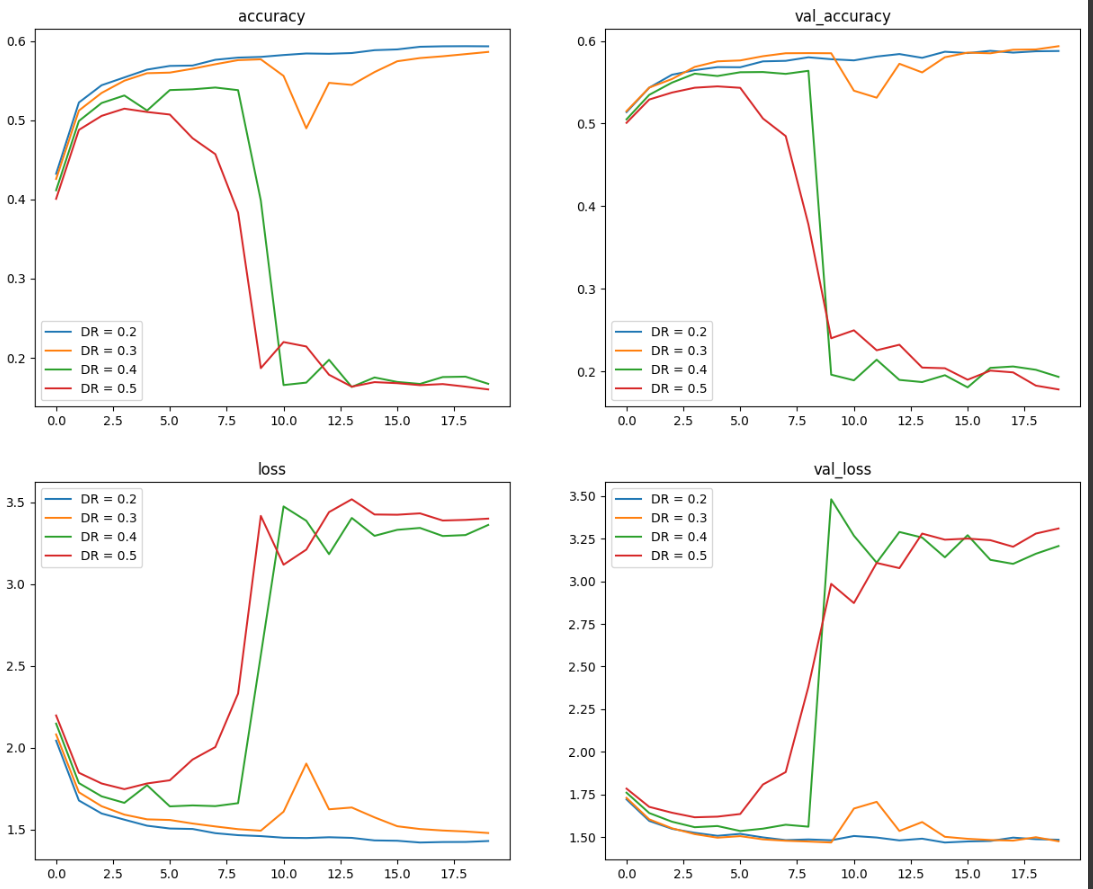


Figure 37: Dropout Rates

Here, I tried different dropout rates which specify the percentage of nodes that are dropped. From Figure 37, DR = 0.4 and DR = 0.5 show an extreme drop in model performance after epoch 8. The high dropout rates seem to result in too many nodes being dropped out, resulting in too much information being discarded, decreasing the model’s ability to learn patterns from the data.

DR = 0.3 still shows some fluctuations in both training and validation. DR = 0.2 resulted in a smaller deviation between train and validation curves as compared to the model without dropout. Additionally, there was no significant impact on validation scores. Hence, we will use dropout layers with a rate of 0.2.

## Final Model

A screenshot of a computer screen

Description automatically generated

Figure 38: Final Model

Our final model shows little deviation between train and test curves, our validation accuracy hovers around 0.596 and validation loss around 1.46. On our test dataset, we get 0.5965 accuracy and 1.4653 loss. As there is little deviation between validation and test, this shows our model is robust enough to generalise consistently with new data.

# Use the developed Model to Generate Texts

## Sampling Techniques

Here, I tried two different sampling techniques. To discern the better technique, BLEU score and human assessment will be used to check the quality of generated texts.

Sampling is done on the same original text: “any ways. For one thing, he hated the summer holidays more “. I standardized the original text to make evaluation easier and fairer across sampling techniques.

### Stochastic Sampling

Stochastic sampling uses randomization, where samples are drawn randomly from the probability distribution.

For stochastic sampling, I will use temperature as well. Temperature is used to define the creativity and diversity of the texts generated. A higher temperature leads to more diverse and creative text while lower values result in more focused and deterministic texts.

A screen shot of a computer code

Description automatically generated

Figure: Stochastic sampling

Above is the defined code for stochastic sampling. After logarithmic, exponential and normalization transformations, we do multinomial sampling with np.random.multinomial. Here, a random draw is taken from a multinomial distribution based on the probabilities in preds. This is like rolling a biased dice. Then, we choose the next character with the highest probability from the draw.

A screenshot of a computer screen

Description automatically generated

Figure : Stochastic Sampling generated text

When we look at the BLEU score, the average BLEU score of all the temperatures is relatively low at around 0.2386. This shows that there is little overlap between the generated text and the original text in terms of n-grams. Typically, a higher overlap in text generation shows that there is some continuity and context between the original and generated texts.

When we read through the generated texts, we see that the words created are quite coherent and most words are spelt correctly for temperatures 0.2 and 0.5. In temperature 0.5, we see that the model can create new words that kind of resemble actual words such as “happing” and “carculates”. In temperature 0.2, we can see repetition and character generation, especially for the word “corner”.

As the BLEU score suggests as well, our generated texts do not have much continuity when we look at semantics.

### Top K Sampling

Top K sampling is a probabilistic sampling technique where of all the predicted tokens, only the top-k most likely tokens are considered. Essentially, we look to limit the most probable tokens rather than sample the entire vocab which helps control the randomness of generated text and produces more coherent and relevant sequences. Hopefully, with top K, we would be able to increase the overlap between original and generated texts by decreasing the randomness.

A screenshot of a computer screen

Description automatically generated

Figure : Top K Sampling generated text

When we look at the BLEU scores, the average BLEU scores across the temperature is lower at 0.2176 compared to stochastic sampling hence showing lesser continuity between generated and original texts.

When we read through the texts, I would say the generation randomness is around the same as stochastic sampling with temperature 0.5. Similarly, there doesn’t seem to be any meaning in the words generated.

**Conclusion**

As stochastic sampling showed better overall BLEU score, I decided to use stochastic sampling. When it comes to text meanings, neither sampling method produced understandable texts hence we will base evaluation off BLEU. I will also use temperature 0.5 as it is random enough to avoid repetition but deterministic enough to avoid generating completely indecipherable words as seen with temperature 1.0.

## User Input

Here, I tried to input some texts to see the quality of text generation for new data. I used a sentence from another Harry Potter book: “Hagrid raised a gigantic fist and knocked three times on the castle door.”

A computer code on a black background

Description automatically generated

Like the data preprocessing section above, I created a sampled array, and one hot encoded for each subsequent sequence. For text generation, I used temperature 0.5.

A screenshot of a computer program

Description automatically generated

Honestly, the generated text did not make any sense to me. The model also created quite a few new words such as “prodetending” which looks like a mis-spelt “pretending”. The generated text also does not have any relevance or continuation from the original text.

A positive is that the model does try to include dialogue as denoted by the quotation marks which shows that we are possibly one step closer to mimicking JK Rowling. Additionally, the generation seems to include paragraphing, as seen by the separation of the generated texts by a two-line space.

# Summary

Here is what I have done:

**Data Preprocessing**

Input text was converted into fixed-length sequences with a specified maximum length (maxlen) and a step size for sliding the window over the input sequences.

Next, sequences are extracted from the text using the defined maxlen and step size. Each sequence is stored in a list, and the next character following the end of the sequence is stored in another list (next\_chars), serving as the target value for predictions.

Unique characters in the text were identified. Unusual and upper-case characters are retained to preserve the writing style of the author. A total of 86 unique characters are identified.

Sequences are vectorized and characters are one-hot encoded. Two arrays (x and y) are created to store vectorized representations. For each sequence, characters are one-hot encoded using a dictionary mapping unique characters to indices.

The data is then split into training and testing sets sequentially, with the first 80% of records used for training and the remaining 20% for testing.

**Model Evaluation**

To evaluate models for training, I used accuracy and loss. For text generation, I used human assessment and an additional BLEU score. BLEU score measures the quality of generated texts using n-grams (consecutive character sequences). It considers matched character counts between generated and original texts, with clipped precision accounting for the maximum frequency of characters in the original text.

**Model Tuning**

I first trained up two different models, LSTM and GRU. I tuned the training hyperparameter values such as learning rate, batch size and model complexity. As LSTM showed heavy overfitting, I also experimented with data augmentation at different strengths, however, it was unsuccessful and I commented that it was unsuitable for character generation tasks. GRU had a very unstable learning process but after tuning learning rate and batch size, training and validation curves were a lot smoother.

For LSTM: Layer size 128 -> 160

For GRU:

RMSProp Learning Rate 0.01 -> 0.001

Batch size 64 -> 128

In the end, I chose LSTM out of the two. Next, I tried Adam optimizer with different learning rates, however, RMSProp still saw better validation scores. Hence I continued to use RMSProp. For activation functions, I tested ReLU, Sigmoid and LeakyReLu, however, I concluded that default TanH allowed for the best model performance. For Kernel initializers, I tried HeNormal, Lecun Uniform and Orthogonal but decided to use the default Glorot Uniform. Lastly, as our model still experiences overfitting, I used dropout layers to make the model less sensitive to specific weights. Dropout rate reduced model overfitting without impacting validation scores hence I adopted it.

My final model hovers around 0.596 validation accuracy and 1.46 validation loss.

**Text Generation**

I compared Stochastic Sampling with temperature and Top K sampling. Stochastic sampling showed a better overall BLEU score hence I continued to use it. I also used temperature 0.5 as it was random enough to avoid repetition but focused enough to generate readable words.

After inputting a sentence from another Harry Potter book, I evaluated the generated text. Unfortunately, the generated text did not make sense to me, but it did try to include dialogue and paragraphing, which shows it can follow the author's writing style to some degree.

**Further Improvements**

Some further improvements would to be experiment with different hyperparameter values for different kernel initializers and activation functions. One thing I noted when experimenting, was that the default turned out to be better as I had been tuning hyperparameters using the default hence the hyperparameter values were more suited to the default.

I could also have introduced explicit regularization earlier. While I did introduce implicit regularization early on, the model was still overfitted and could have affected my experimentation process.