Abstract

Autoregressive Character-Level Language Models: Comparative Analysis

Aryan Yadav

2023

Language, in its essence, is the bedrock of human communication, and the advent of artificial intelligence has opened new frontiers in understanding and replicating its complexities. This capstone project is an exploration into the world of autoregressive character-level language models, which are at the forefront of teaching machines how to mimic the nuances of human language. Specifically, this study focuses on the generation of names, a seemingly simple yet intricate task. The journey begins with the Bigram model, a basic yet crucial stepping stone in understanding language patterns. It's like learning to crawl before you walk. The Bigram model sets the stage by using pairs of characters to predict the next character, laying the groundwork for more complex models to follow. Advancing from the Bigram model, the study explores the Multilayer Perceptron (MLP). The MLP, with its ability to capture more intricate patterns through layers and nonlinear activations, represents the first foray into neural network architectures in this study. It's akin to moving from basic arithmetic to algebra; the fundamental principles remain, but the complexity and capabilities increase. Next, the project delves into the Recurrent Neural Network (RNN). The RNN is particularly fascinating because it introduces the concept of memory in processing sequences. It's not just about what you say but how what you've said before influences what you're about to say. This ability to remember previous inputs makes the RNN adept at handling sequences like text, a crucial aspect of language. The crescendo of this exploration is the Transformer model. The Transformer stands out with its self-attention mechanism, a breakthrough concept that allows the model to

weigh the importance of each part of the input data. Imagine having a conversation where you could pay equal attention to every word spoken, understanding the context and nuances fully. That's what the Transformer does with language. Each of these models was implemented, trained, and tested, not just in theory but in practice. The dataset comprised names, a familiar yet complex aspect of language. The models were tasked with generating names, both replicating those in the dataset and creating new, plausible ones. This practical approach allowed for a hands-on understanding of each model's strengths, limitations, and applicability in real-world scenarios. Comparing these models unveiled a fascinating progression in language modeling capabilities. The Bigram model, while simple, was limited in its predictive power. The MLP showed improvement but still lacked the sequential memory crucial for handling language. The RNN addressed this with its memory mechanism, leading to better name generation. However, it was the Transformer model that truly excelled, demonstrating superior performance in generating realistic and diverse names. This capstone project is not just about algorithms and models; it's about bridging the gap between human language and machine understanding. It shows how we've moved from basic statistical approaches to sophisticated neural networks, each step bringing us closer to machines that can understand and generate language more like humans. However, this journey is not without its challenges. Data quality, model complexity, and the balance between accuracy and creativity are just some of the hurdles faced along the way. These challenges highlight the ongoing quest in computational linguistics to create models that not only mimic but also understand and innovate in language usage.

Autoregressive Character-Level Language Models: Comparative Analysis

A Capstone Project
Presented to the Faculty of Computer Science
of
Ashoka University
in partial fulfillment of the requirements for the Degree of
Postgraduate Diploma in Advanced Studies and Research

by
Aryan Yadav

Advisor: Professor Debayan Gupta

December, 2023

Contents

List of Figures

List of Tables

Acknowledgements

I extend my deepest gratitude to Professor Debayan Gupta, my advisor, for his invaluable guidance and support throughout this challenging project.

Heartfelt thanks go to Dhaba, Roti Boti, and Rasananda for their dependable food deliveries, which were a cornerstone of my late-night work sessions.

I am immensely thankful to my friend Samvit Jatia for his guidance and encouragement in the initial phases of this project.

My sister, Aishwarya Yadav, has been a mentor and a constant source of support through my undergraduation and for that, I am eternally grateful.

Special thanks to Kyra Philip, whose assistance played a pivotal role in the success of this project.

I am also thankful to my girlfriend, Riwa Desai, for her companionship and engaging discussions during those long hours in the library.

Finally, my heartfelt appreciation goes to my parents, GS and Neelam Yadav, whose support has been fundamental in enabling me to pursue such fascinating projects.

Chapter 1

Introduction

In this project, I explore the intriguing domain of autoregressive character-level language models, an area that sits at the intersection of language and technology. As a fourth-year undergraduate in Computer Science, my fascination has been piqued by how machines can be taught to understand and replicate human language, particularly in the context of generating names. This project represents a foray into that realm.

My journey begins with the Bigram model, the most fundamental of the models under examination. It serves as the foundational block, providing a baseline for understanding more complex structures. From there, the exploration advances to more sophisticated models such as the Multilayer Perceptron (MLP), Recurrent Neural Network (RNN), and ultimately, the Transformer model. Each model is a step up in complexity and capability, offering a unique lens through which to view language processing.

This project is not just a theoretical exercise; it is grounded in practical application. Each model has been implemented, tested, and analyzed to understand its practical

efficacy. The objective is to draw comparisons across these models to discern advancements in language modeling, particularly in how they learn and generate language constructs like names.

The aim here is to analyze and understand the complexities and workings of these models and present their workings in a manner that is accessible yet comprehensive.

Chapter 2

Dataset

2.1 Overview

In this chapter, I delve into the dataset, which is the bedrock for my explorations with Bigram, MLP, RNN, and Transformer models. The effectiveness of these models in autoregressive character-level language modeling hinges on the quality and structure of this dataset.

2.2 Dataset Creation

2.2.1 Source and Content

'names.txt', a text file brimming with unique names, each on a new line, forms the foundation of my dataset. It consists of a total of 32,000 words - one on each line - imported from ssa.gov for the year 2018.

2.2.2 Loading and Initial Processing

Reading the dataset is more than a technical step; it's the first interaction with the data. By carefully stripping whitespace and discarding empty lines using Python's

file handling, I ensure that the dataset is as clean and consistent as possible and finally import all words.

2.3 Dataset Preprocessing

2.3.1 Character Set Extraction

From this dataset emerges a unique set of characters, each a small but significant part of the model's vocabulary. Sorted alphabetically, these characters provide a consistent base for further processing.

Visualization: Character Distribution

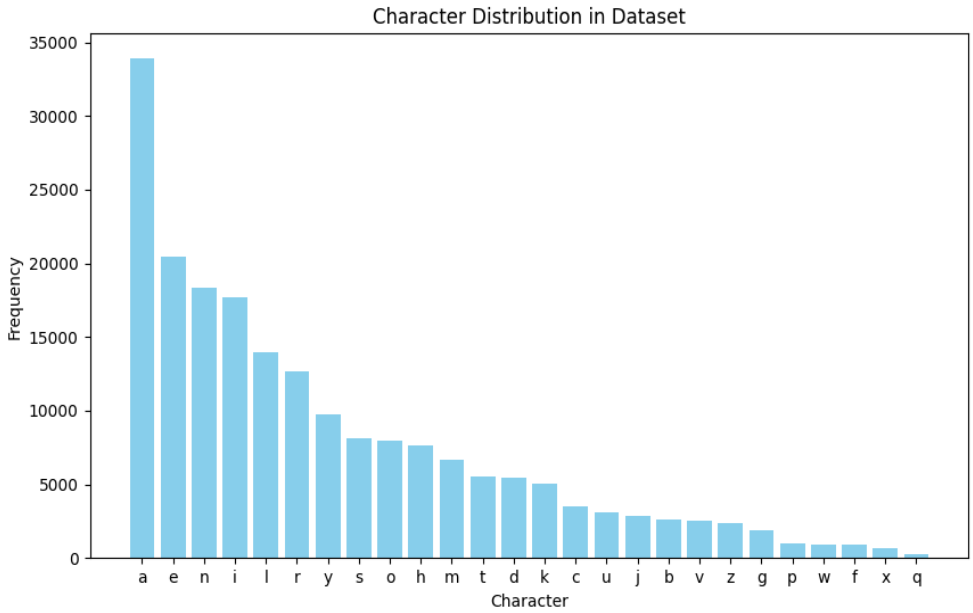

Figure 2.1: Dataset: Bar chart showcasing the frequency of each character.

2.3.2 Maximum Word Length

Determining the maximum word length is a strategic decision that shapes how the models perceive and process the data, defining the upper limit of character sequences they will encounter.

Visualization: Name Lengths Distribution

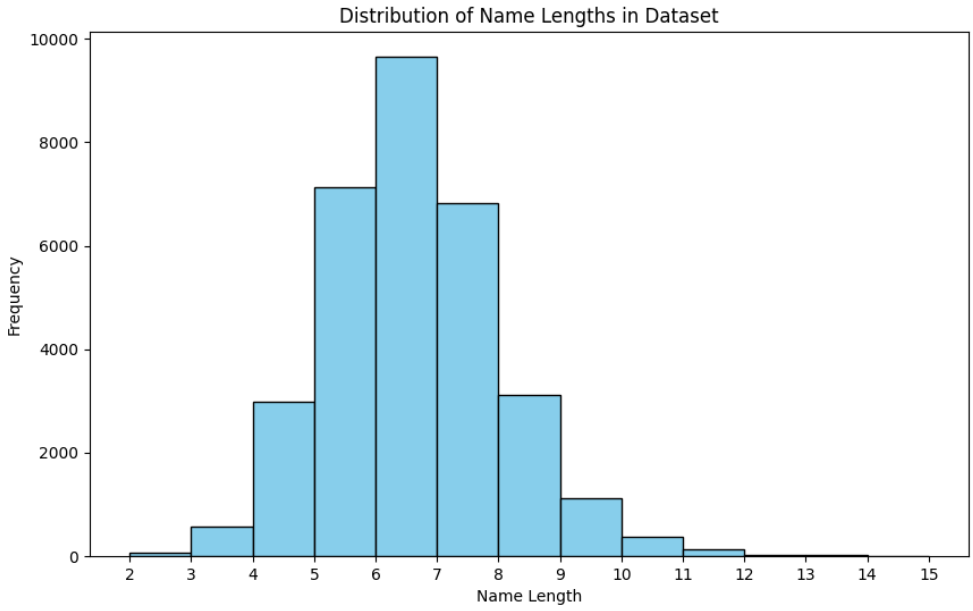

Figure 2.2: Dataset: Histogram illustrates the frequency of different name lengths.

2.4 Dataset Processing

2.4.1 Encoding

The transformation of each name into a sequence of numbers is a fascinating process. Here, each character is assigned a unique integer, turning text into a form that's amenable to algorithmic manipulation - a look-up table "stoi" mapping each character

"s" to a unique integer "i", for example, a=0, b=1 etc, and vice-versa - a itos look-up table.

2.4.2 Special Token

I use "." as a special token that will denote the beginning and/or end of a word. The introduction of special tokens to denote sequence start and end points is a nuanced approach that aids the model in understanding the boundaries of each name.

2.4.3 Train-Test Split

By dividing the dataset into training and test subsets, with a 90-10 split, I establish a framework for model training and evaluation, ensuring that the models are not just trained well but can also generalize to unseen data and be tested.

Chapter 3

Bigram Model

3.1 Introduction

This chapter delves into the Bigram model, a fundamental component in autoregressive character-level language models and how I have implemented it. It explores how simple character pairings can illuminate the structure and patterns of language.

3.2 Theoretical Foundations

3.2.1 Essence of Bigrams

A bigram is a sequence of two consecutive characters. The Bigram model uses this concept to predict each character based on its predecessor, laying the groundwork for understanding character sequence dependencies [1]. So, a word like "emily" would have these as its bigrams: .e, em, mi, il, ly, y. (remember the special token from 2.4.2?)

3.3 Implementation Insights

3.3.1 Preparing Data for Bigrams

The construction of the Bigram model involves cataloging every character pair in the dataset and counting their occurrences, breaking down names into letter pairs to analyze frequency.

Each name in the dataset is transformed into character pairs. This step is crucial for the Bigram model, converting text into a learnable format.

3.4 Visualization and Interpretation

3.4.1 Bigram Frequency Heatmap

A heatmap visually represents the frequency of character pairs, with different colors indicating different frequencies, making it easy to identify common pairs. Here we can infer that "a" and "n" appear together the most in all the words in this dataset.

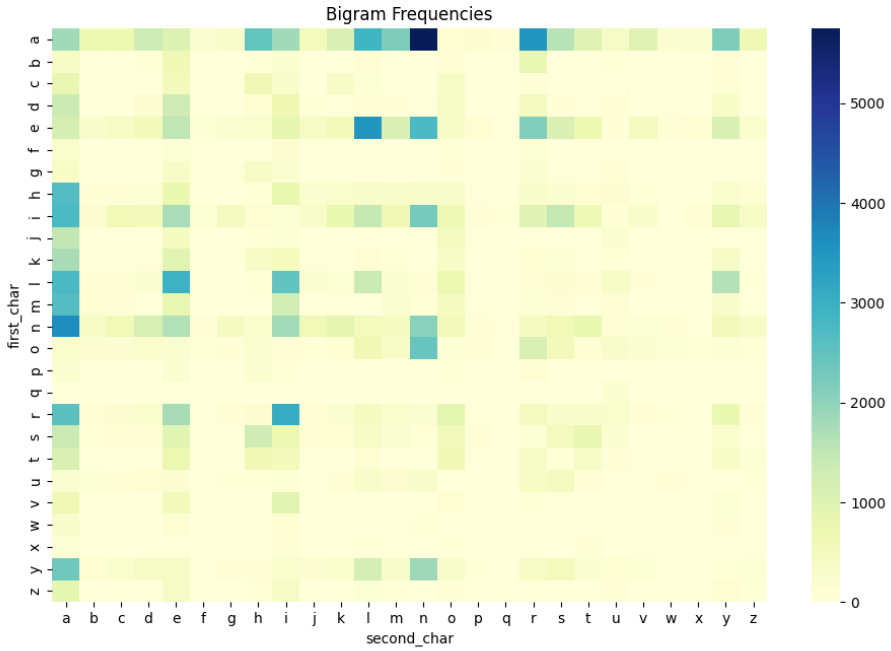

Figure 3.1: Bigram: Heatmap showing frequency of all character pairings

3.4.2 Most Common Bigrams

A bar chart of 10 most common bigrams indicates the most frequent pairs highlighting key character pairings.

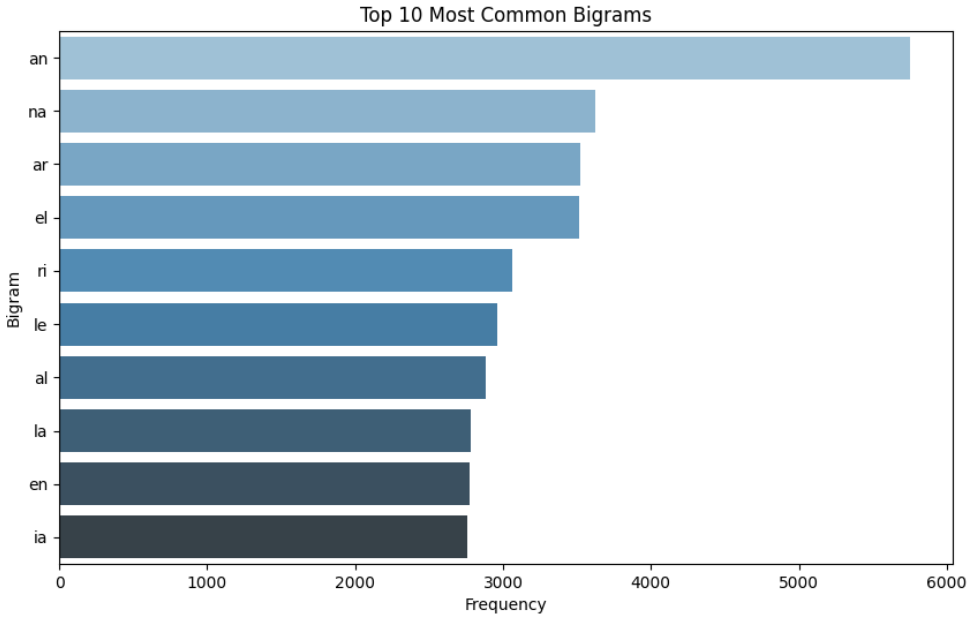

Figure 3.2: Bigram: Bar graph displaying the 10 most common bigrams

3.5 Overview of the Training Process

I trained the bigram model for 150,000 steps with 729 (27*27) parameters logging the train loss every 10 steps and the test loss every 500 steps.

3.5.1 Structure and Content of the Logs

Training logs follow a structured format, detailing step number, training loss value, and step duration, facilitating the analysis of the model's progress. The new test loss logged every 500 steps was compared to previously lowest test loss $\rightarrow$ if new test loss $\leq$ previous lowest test loss: save model, else: continue (if iterations left).//

Average test loss I found for this model is: 2.454 whereas the average test loss came out to be: 2.501.

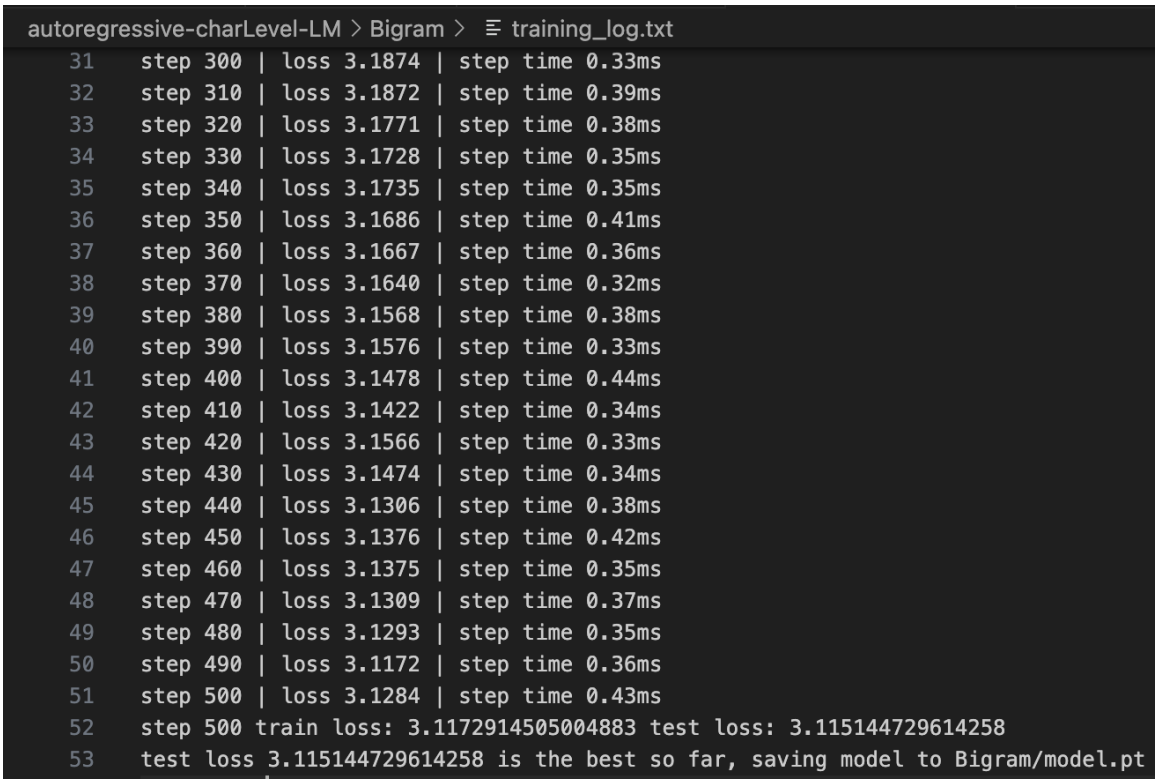

Figure 3.3: Bigram: Training logs excerpt.

3.5.2 Training Loss Visualization

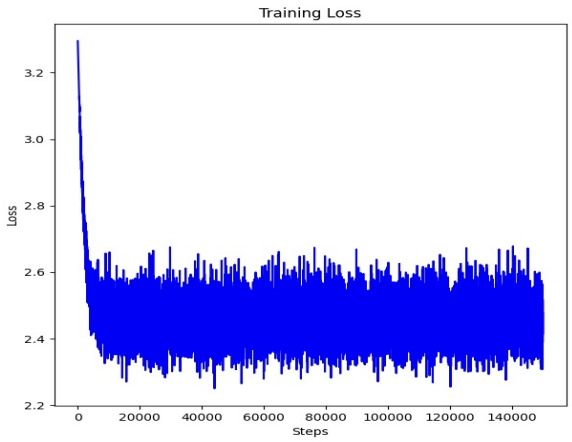

Figure 3.4: Bigram: Train loss

3.5.3 Test vs Train Loss Visualization

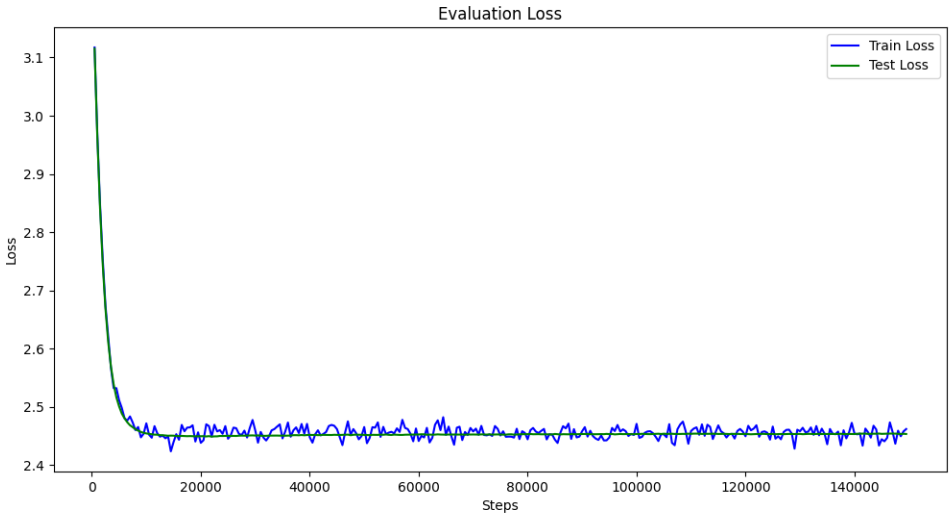

Figure 3.5: Bigram: Train vs Test loss

3.6 Model Evaluation

3.6.1 Interpreting Loss Values

As we can see from figure 3.4 the training loss starts off very high at 3.2958 and then averages out to 2.4538 and figure 3.5 shows us the comparison between training and test loss and the best test loss comes out to be 2.4495. Another inference is that the test loss doesn't seem to plateau which is an indicator of data overfitting.

3.7 Results

Finally I sampled names from the Bigram model at a test loss of 2.4495 and these were the results:

Table 3.1: Bigram: Final name samples

Category	**Samples**
In Train	ma, acey, an, che, niam, ki
In Test	(none)
New	jurn, h, ne, avenelitya, vyken, ninden, kackeseickhanie, a, ja, ano, jereyasah, chame, farahaele, rolama, esaila, ka, jassubaqula, bili, l, balaleieeh, tarorriyn, g, vaodann, niyioneininnnan, n, briricelavaegar, jarus, core, ainny, ney, afr, disan, a, lyr, e, nda, br, kyn, dijachoesare, gall, monjarharavalen, viabrarlenla, feddomani, eors

These did not turn out to be good mainly due to the nature of the bigram model. And with my best test loss value being 2.4... these results would be expected but bound to improve exponentially in the following models.

Chapter 4

MLP Model

4.1 Introduction

This chapter delves into the intricacies of the Multilayer Perceptron (MLP) model, a cornerstone in the realm of neural networks. Especially significant in the context of autoregressive character-level language models, the MLP model exemplifies how neural networks can capture and generate linguistic patterns [2].

4.2 Theoretical Foundations

4.2.1 Essence of MLPs

The Multilayer Perceptron, a type of feedforward neural network, consists of multiple layers including input, hidden, and output layers. Each layer is fully connected to the next, and the network employs nonlinear activation functions, allowing it to model complex relationships in data [2].

4.3 Implementation Insights

4.3.1 Data Processing for MLPs

For the MLP model, data preprocessing involves transforming names into a series of integers representing characters. These integer sequences serve as inputs to the MLP, which then predicts the subsequent characters in the sequence.

4.3.2 Architectural Details

The MLP model implemented in this study comprises an embedding layer, which maps character indices into high-dimensional vectors [2]. These vectors are processed through multiple linear layers interconnected with nonlinear activation functions - in my case I used the hyperbolic tangent function as my activation function. The output layer generates a probability distribution over possible subsequent characters.

4.4 Visualization and Interpretation

4.4.1 Character Embedding Visualization

Using t-SNE for dimensionality reduction, we visualize the embedding space of characters, offering insights into the learned representations.

As we can see from figure 4.1 the characters are all almost equally distant from each other, which conveys that the model is not very good at categorizing contextually similar characters together.

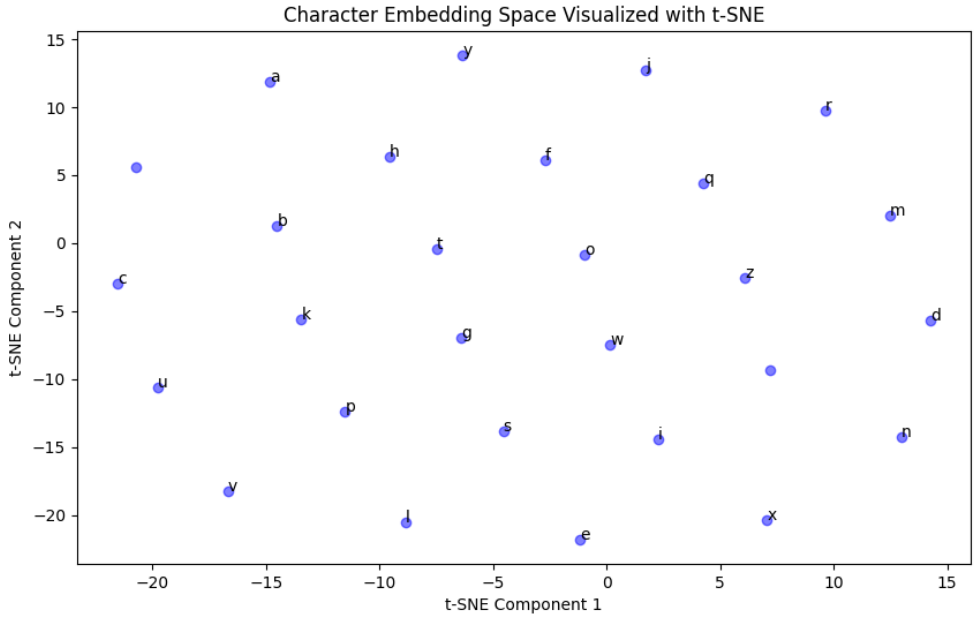

Figure 4.1: MLP: Character embeddings visualization using t-SNE.

4.4.2 Visualizing Model Responses to Names

The model's behavior is examined through its hidden layer activations and output probability distributions. Here I display both graphs for 4 specific names: "Charlotte," "Zoe," "Alexandria," and "Jacqueline."

I have chosen the following names: 'Charlotte', 'Zoe', 'Alexandria', 'Jacqueline' because 'Charlotte' and 'Alexandria' are long but normal names whereas 'Zoe' is a short one and 'Jacqueline' is a long and uncommon name (and wanted to see what the model thinks of the letters 'c', 'q', 'u' together since that is very rare).

Hidden Layer Activations for 'Charlotte'

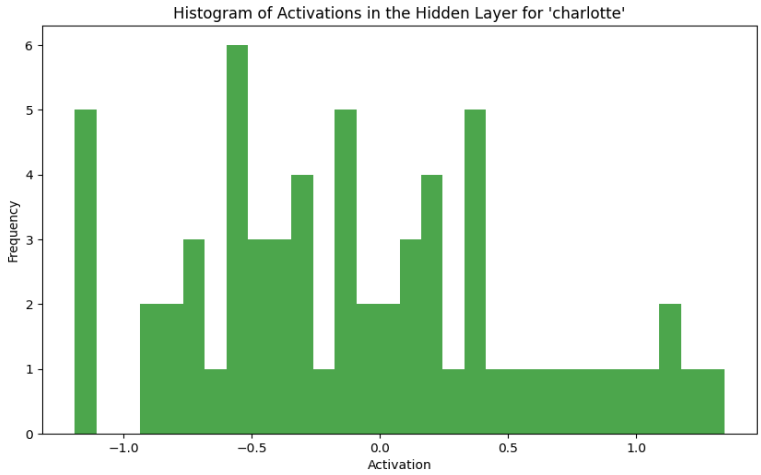

Figure 4.2: MLP: Hidden layer activations for 'Charlotte'.

Output Probability Distribution for 'Charlotte'

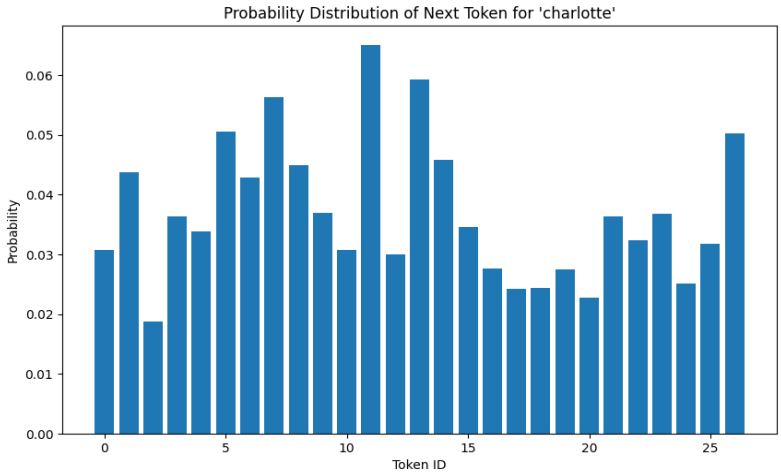

Figure 4.3: MLP: Output Probability Distribution for 'Charlotte'.

Hidden Layer Activations for 'Zoe'

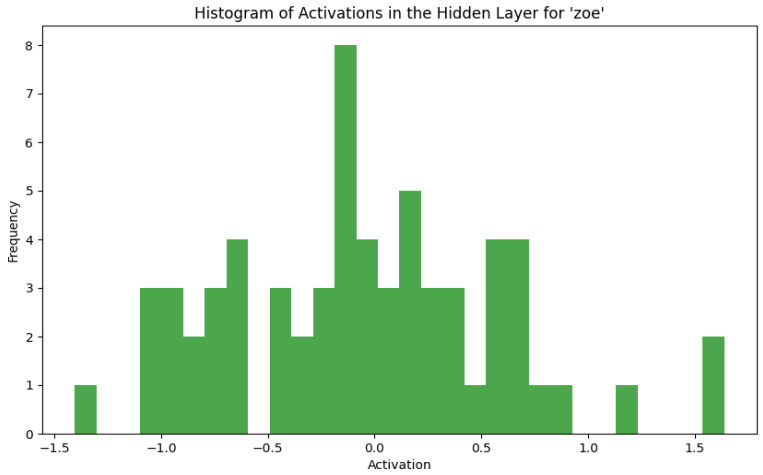

Figure 4.4: MLP: Hidden layer activations for 'Zoe'.

Output Probability Distribution for 'Zoe'

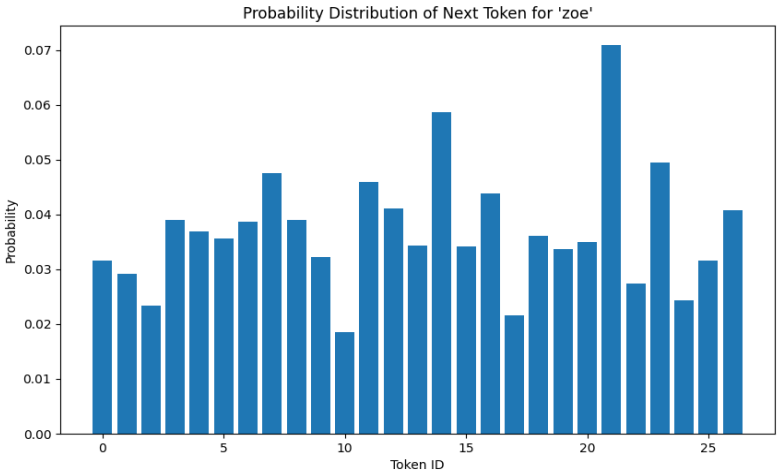

Figure 4.5: MLP: Output Probability Distribution for 'Zoe'.

Hidden Layer Activations for 'Alexandria'

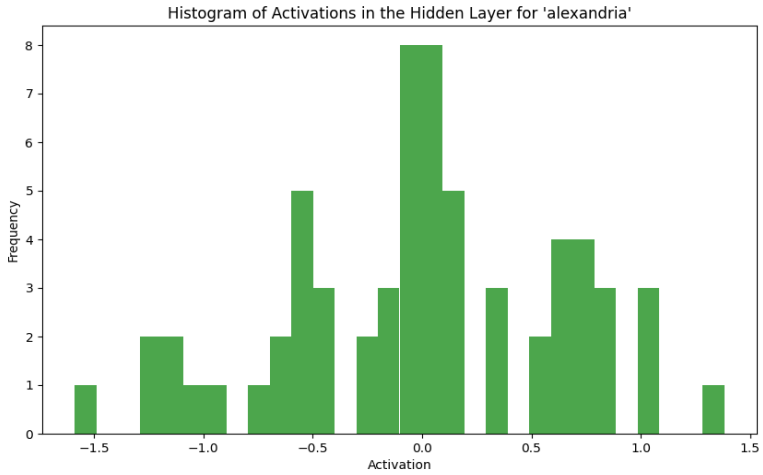

Figure 4.6: MLP: Hidden layer activations for 'Alexandria'.

Output Probability Distribution for 'Alexandria'

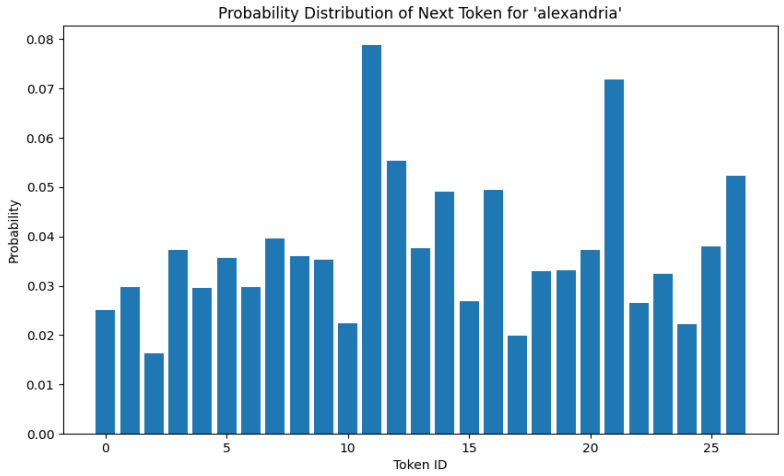

Figure 4.7: MLP: Output Probability Distribution for 'Alexandria'.

Hidden Layer Activations for 'Jacqueline'

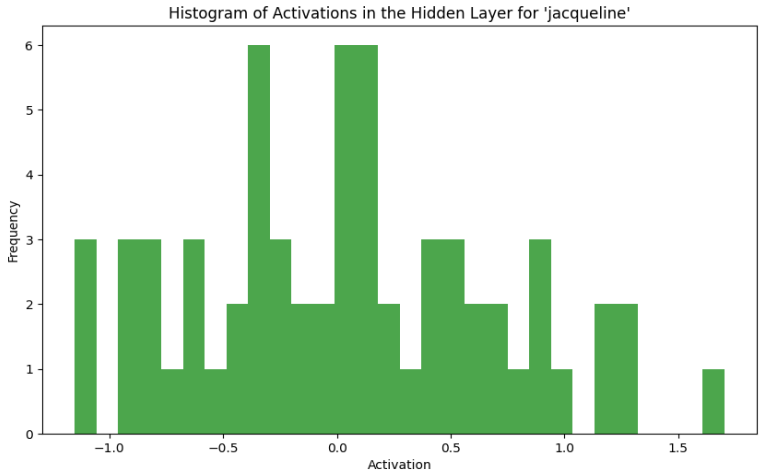

Figure 4.8: MLP: Hidden layer activations for 'Jacqueline'.

Output Probability Distribution for 'Jacqueline'

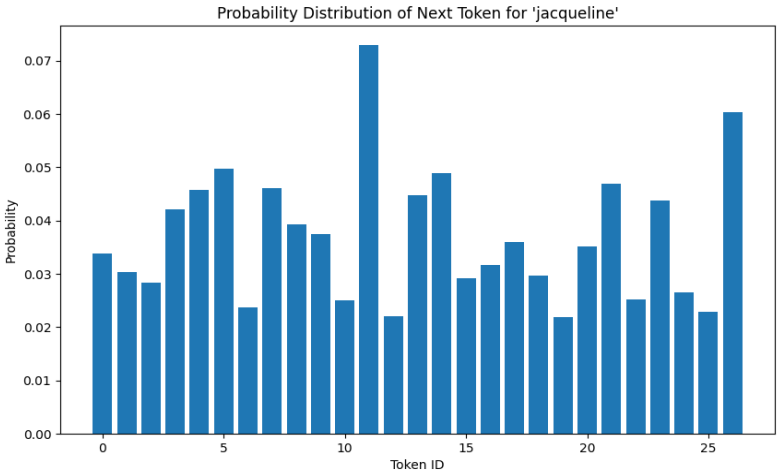

Figure 4.9: MLP: Output Probability Distribution for 'Jacqueline'.

4.4.3 Activations Results Analysis

For "charlotte", the activations are spread across a range of values, but none are particularly high. This could suggest that the model is considering many aspects of the input when thinking about this word, but it's not overly focused on any single feature. It's like it's carefully considering several different letters in the word before deciding what to do next.

In the case of "zoe", we see more variability with some higher peaks, which could indicate that the machine has found certain parts of this word to be more important or distinctive than others. It's like the machine is saying, "Oh, these particular letters are really interesting, and I should pay more attention to them!"

Looking at "alexandria", we observe a very high peak around the center. This could mean the machine has latched onto a specific feature or pattern within this word that it thinks is very important. It's as if the machine has a favorite part of the word that it focuses on more than the rest.

Finally, for "jacqueline", the activations are somewhat evenly spread with a couple of modest peaks. This implies that the machine is spreading its attention across multiple features of the input, similar to "charlotte", but with some parts slightly more prominent than others.

4.4.4 Probability Distribution Analysis

The probability distribution graphs for the next token predictions after 'charlotte', 'zoe', 'alexandria', and 'jacqueline' demonstrate the model's predictive confidence and variability. The spread of probabilities for 'charlotte' and 'jacqueline' indicates

a distributed uncertainty, suggesting a balanced approach with no singularly favored predictions.

Conversely, 'zoe' exhibits a pronounced peak, reflecting a strong model preference and potential predictive reliability. 'Alexandria' shows a moderate level of confidence with a few favored predictions. These patterns collectively highlight the model's nuanced approach to different inputs, showcasing both its decisiveness and its calculated hesitance in the face of ambiguity.

4.5 Training Process Overview

The MLP model underwent a rigorous training process spanning 150,000 steps. This process involved updating weights based on gradient descent, specifically using the AdamW optimizer, a variant of the Adam optimizer. AdamW is known for its effectiveness in handling sparse gradients and adaptive learning rates, making it suitable for our MLP model. Numberr of parameters the model was fed is 69147.

4.5.1 Training Log Structure

Training logs for the MLP model are structured to record the iteration step, the loss at each step, and the time taken for each iteration. This structured logging facilitates tracking the model's performance and identifying areas for improvement.

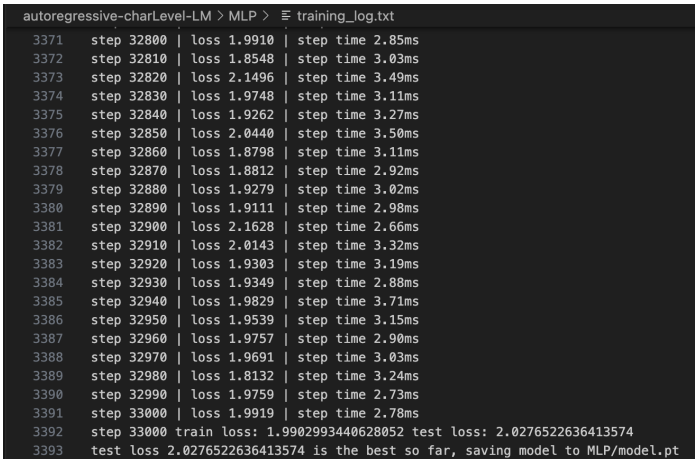

Figure 4.10: MLP: Training log excerpt

4.5.2 Training Loss Visualization

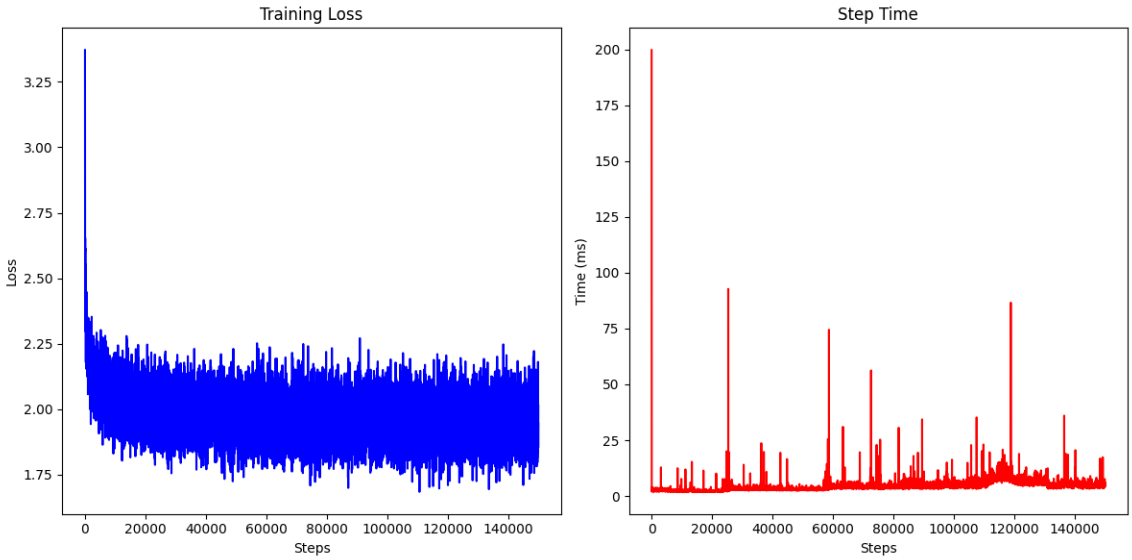

Figure 4.11: MLP: Train Loss

4.5.3 Test vs Train Loss Visualization

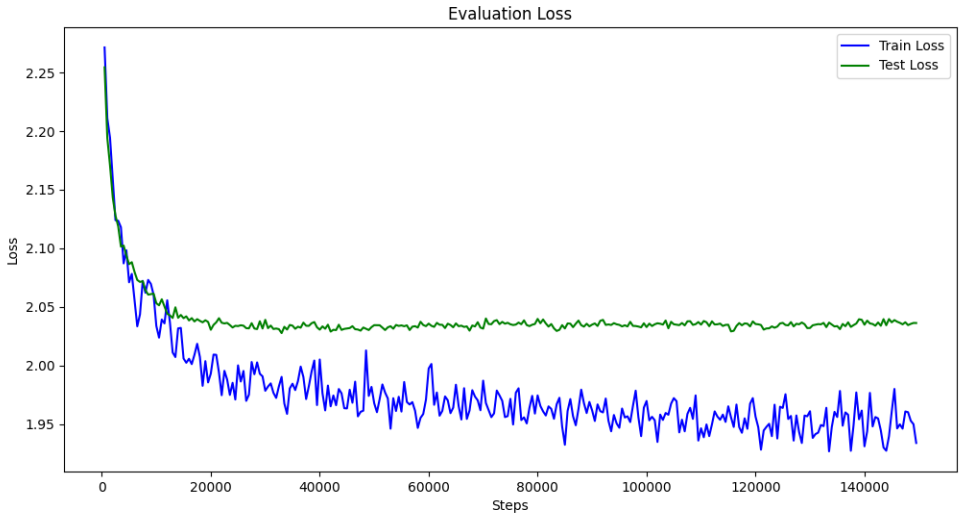

Figure 4.12: MLP: Train vs Test Loss

4.6 Model Evaluation

4.6.1 Interpreting Loss Metrics

Loss values offer a quantifiable measure of the model's predictive accuracy, with lower values indicating better performance [2]. As we can see from figure 4.11 the training loss starts off very high at 3.28 and then averages out to 1.99 and figure 4.12 shows us the comparison between training and test loss from which I infer the best test loss to be 2.01 which is still high.

Another inference is that the test loss plateau's and even slightly increases towards the end which, as mentioned before, is an indicator of data overfitting.

4.7 Results

Post-training, the MLP model was used to generate a variety of names. The output included names from the training dataset and generated original new names.

Table 4.1: MLP: Final name samples

Category	**Samples**
In Train	keilan, jaxlyn, kylen, lilia, dionna, elleny, iyanna, simona, daviya, joyce
In Test	(none)
New	alifa, ja, keidi, dayline, paikset, merianna, syma, caizlei, arim, samulle, sambo, adenna, aseiza, leldei, aalilie, yany, savan, pan, milax, jahay, aryl, pheno, arakoa, aderah, evei, galuon, akaess, jkarelyn, noyel, vithen, melily, kaleni, land, hanuliah, calyss, harume, janasiar, leyaleah, anoriyah, madynna

This outcome illustrates the MLP model's enhanced capability for name generation, surpassing the performance observed with the simpler Bigram model but it is still not optimum as we saw test data is overfit. Test loss (2.0 ...) is too high.

Chapter 5

RNN Model

5.1 Introduction

In this chapter, I explore the Recurrent Neural Network (RNN) model, a pivotal concept in sequential data processing and a critical component in my autoregressive character-level language model. The RNN model is adept at handling sequences, such as text, by maintaining a memory of previous inputs [3].

5.2 Theoretical Foundations

5.2.1 Essence of RNNs

Recurrent Neural Networks differ from other neural architectures due to their ability to process sequences of data. They achieve this through loops within their network structure, allowing information persistence. Each neuron in an RNN can be thought of as having a memory of previous data points, making them ideal for tasks like language modeling [3].

5.3 Implementation Insights

5.3.1 Data Processing for RNNs

Similar to the MLP model, the RNN processes data by transforming names into integer sequences that represent characters. The RNN, however, processes these sequences differently, using its memory mechanism to maintain a contextual understanding of the sequence.

This memory mechanism is the defining feature of RNNs. It allows the model to retain information from previous inputs in a sequence, enabling it to maintain a contextual understanding as it moves through the input. Each step in processing a sequence involves not just the current input (a character in this case) but also a 'memory' of what has come before [3].

5.3.2 Architectural Details

My RNN model, implemented in Python, utilizes an embedding layer to convert character indices into high-dimensional vectors. These vectors are then fed through a series of recurrent layers, each maintaining a state that captures information from previous inputs. For this model I employed the hyperbolic tangent function as my activation function. The final layer of the RNN outputs a probability distribution (visualised ahead) for the next character in the sequence.

5.4 Visualization and Interpretation

5.4.1 Character Embedding Visualization

Just like the MLP model, I use t-SNE to visualize the embedding space of characters in the RNN model. This visualization provides insights into how the RNN perceives

different characters in relation to each other based on their contextual usage, exactly as it was in - the closer a character is to another, the stronger the model believes they are similar contextually.

And as we can see some characters are much more closer (and farther) than others conveying that this model is learning more than the MLP model

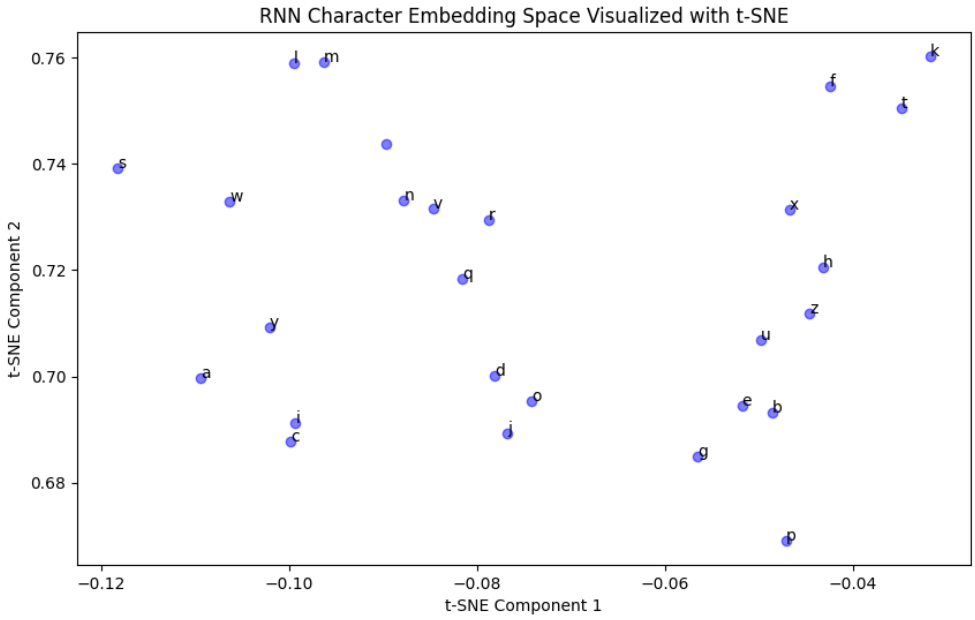

Figure 5.1: RNN: Character embeddings visualization using t-SNE.

5.4.2 Visualzing Model Responses to Names

To understand how the RNN model processes individual names, I examine both the hidden layer activations and the output probability distributions for the same 4 selected names. These visualizations help in comprehending the model's internal mechanics just like in MLP.

Hidden Layer Activations for 'Charlotte'

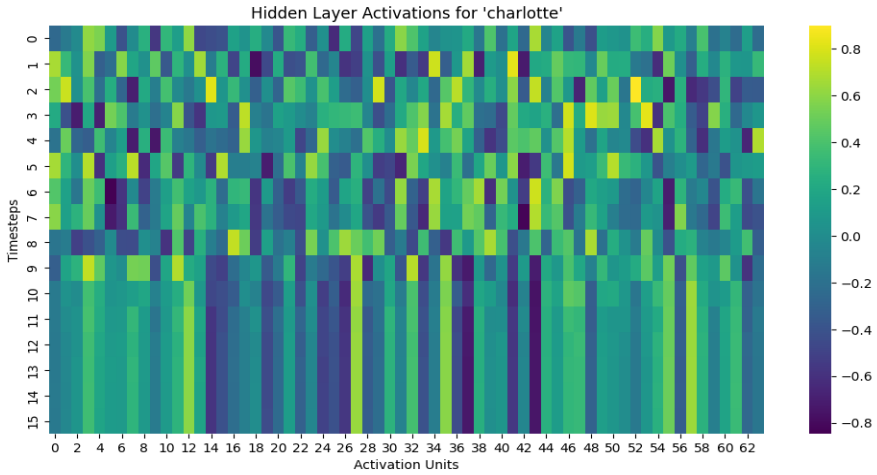

Figure 5.2: RNN: Hidden layer activations for 'Charlotte'.

Output Probability Distribution for 'Charlotte'

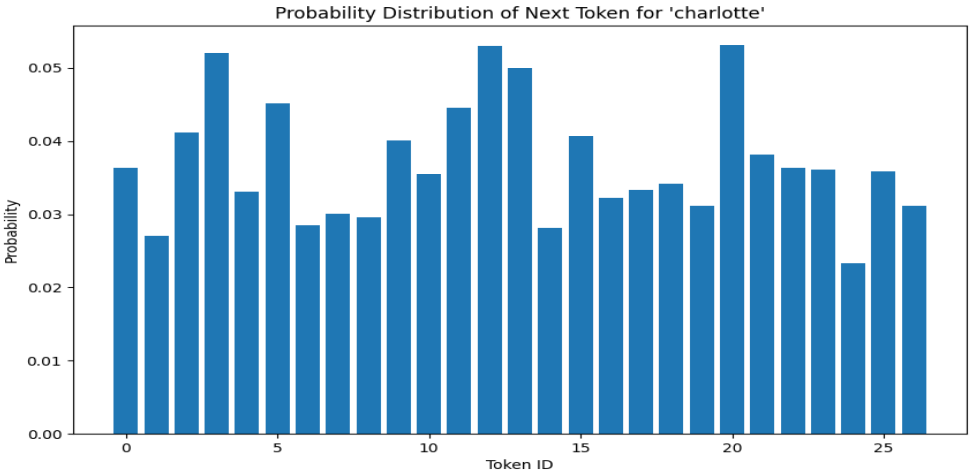

Figure 5.3: RNN: Output Probability Distribution for 'Charlotte'.

Hidden Layer Activations for 'Zoe'

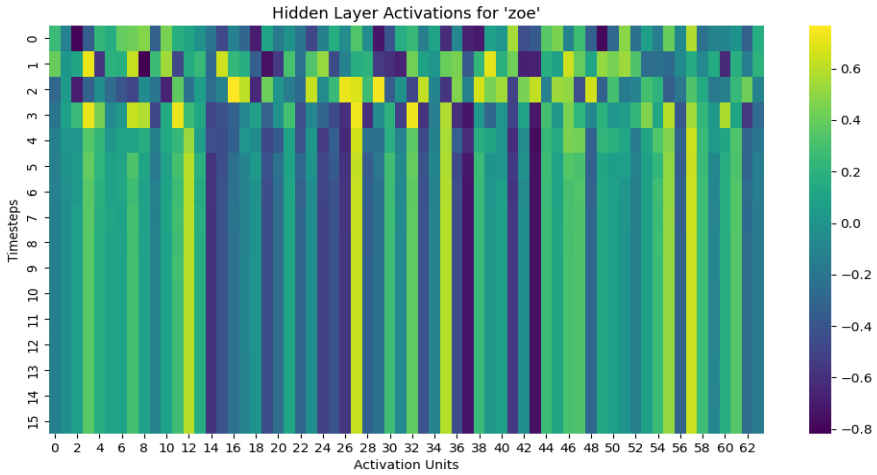

Figure 5.4: RNN: Hidden layer activations for 'Zoe'.

Output Probability Distribution for 'Zoe'

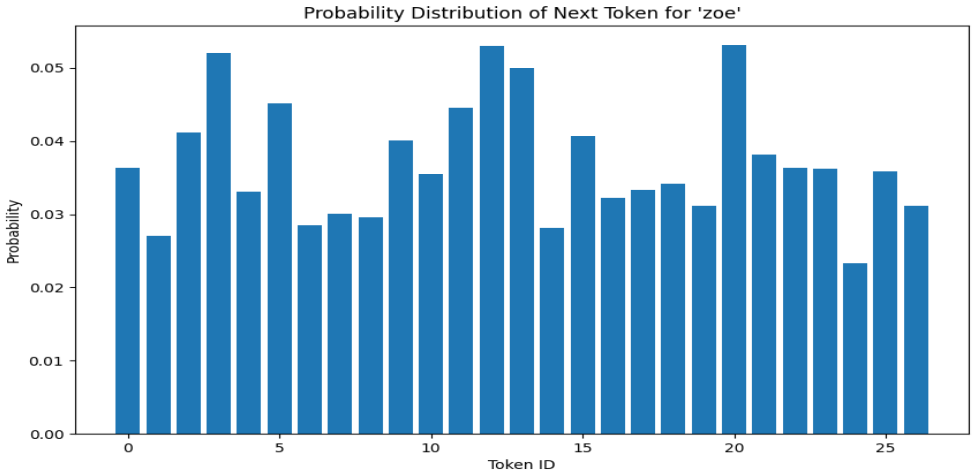

Figure 5.5: RNN: Output Probability Distribution for 'Zoe'.

Hidden Layer Activations for 'Alexandria'

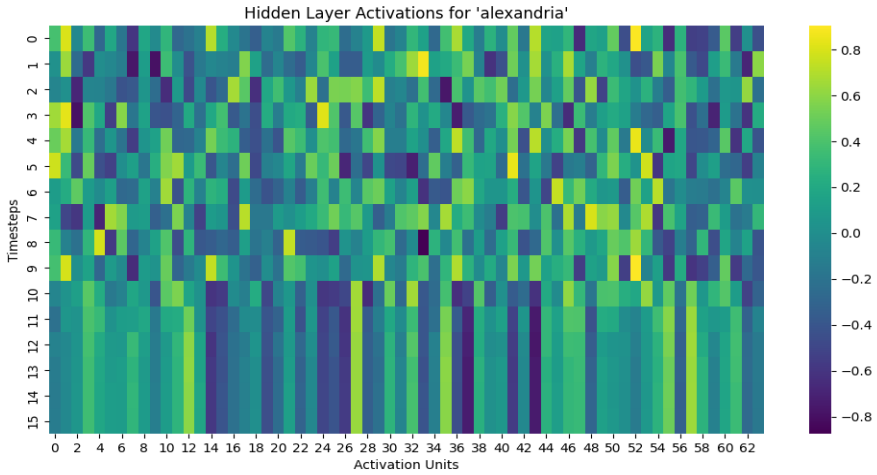

Figure 5.6: RNN: Hidden layer activations for 'Alexandria'.

Output Probability Distribution for 'Alexandria'

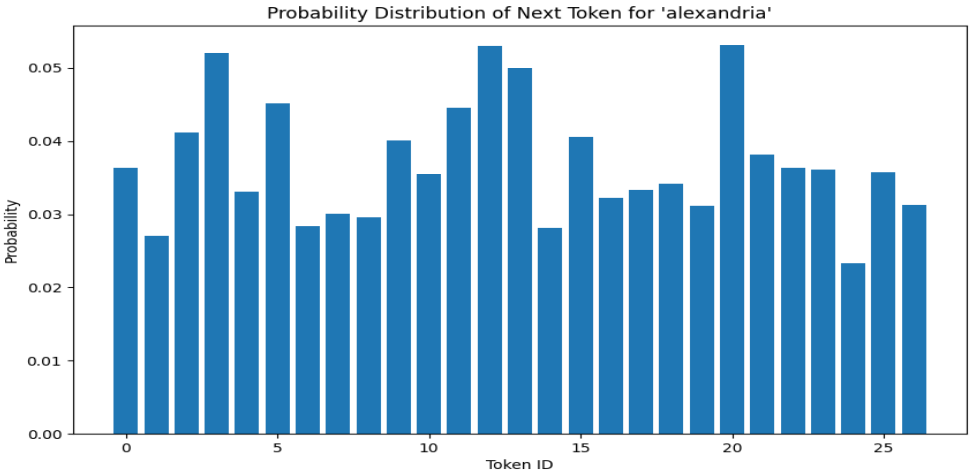

Figure 5.7: RNN: Output Probability Distribution for 'Alexandria'.

Hidden Layer Activations for 'Jacqueline'

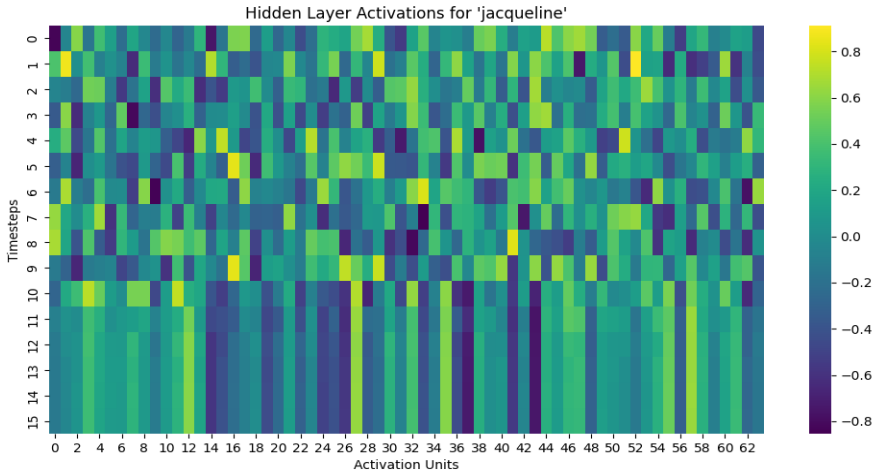

Figure 5.8: RNN: Hidden layer activations for 'Jacqueline'.

Output Probability Distribution for 'Jacqueline'

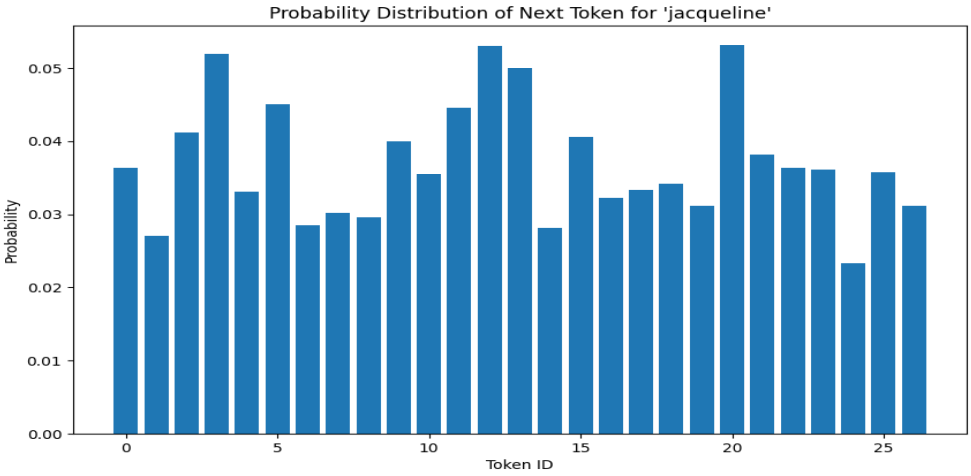

Figure 5.9: RNN: Output Probability Distribution for 'Jacqueline'.

5.4.3 Activations Results Analysis

In this heatmap, the different colors correspond to the magnitude of the activations, with a scale shown on the right. The scale goes from -0.8 to 0.8, indicating that the activations can be either positive or negative. The varying colors across the heatmap suggest that the RNN's hidden units are activated differently at each time step. This is typical for an RNN, as it processes sequential data where each time step corresponds to a part of the sequence (e.g., a character in the word "charlotte").

The activations across the RNN's hidden layer show significant variability both across the time steps and the different activation units, which is visible in the changing colors in the heatmap. This indicates that the RNN is dynamically adjusting its internal state as it processes the sequential input. Certain columns, which represent individual activation units, exhibit a consistent activation pattern across the time steps. This consistency may suggest that these units are specifically tuned to recognize certain features or patterns within the sequence, such as specific letters or their combinations that make up the input word.

5.5 Training Process Overview

The training of the RNN model spanned 150,000 iterations, involving a continual update of weights to minimize prediction errors. The optimization technique employed was the AdamW optimizer, chosen for its efficiency in sparse gradient scenarios. Total model parameters = 11803

5.5.1 Training Log Structure

The training logs of the RNN model are meticulously structured to record crucial information such as the iteration step, loss at each step, and the time taken per iteration.

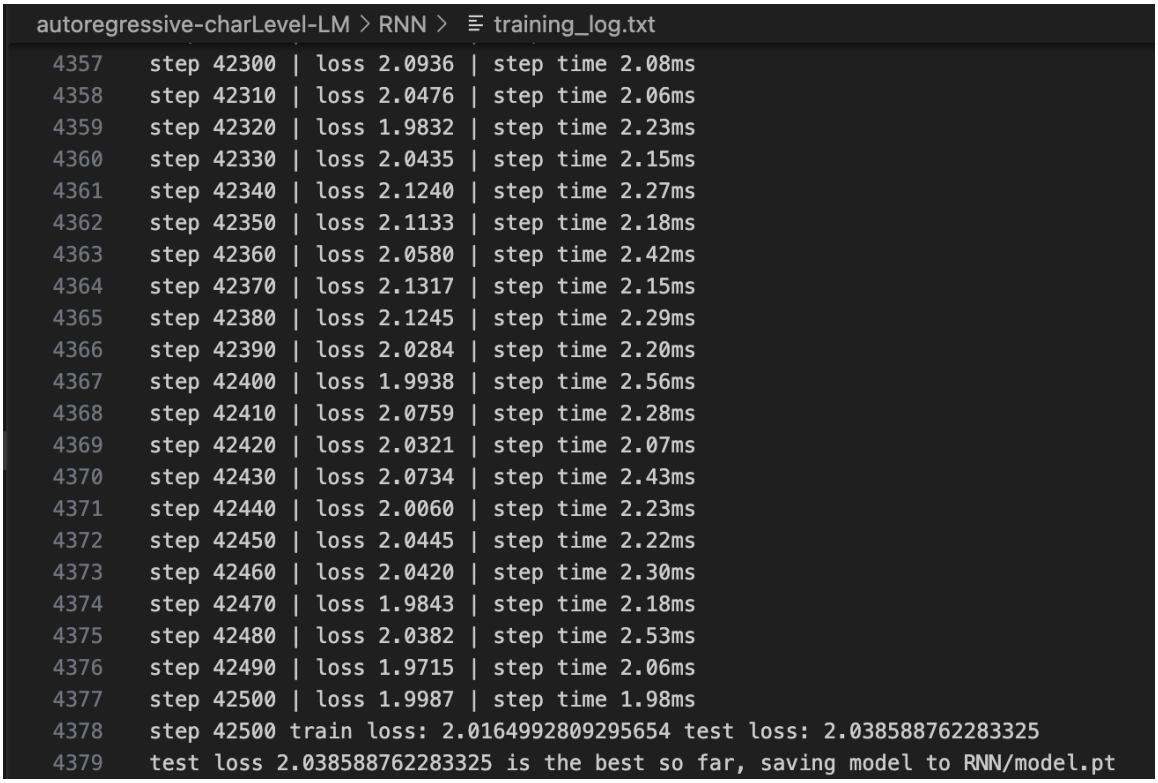

Figure 5.10: RNN: Training logs excerpt

5.5.2 Training Loss Visualization

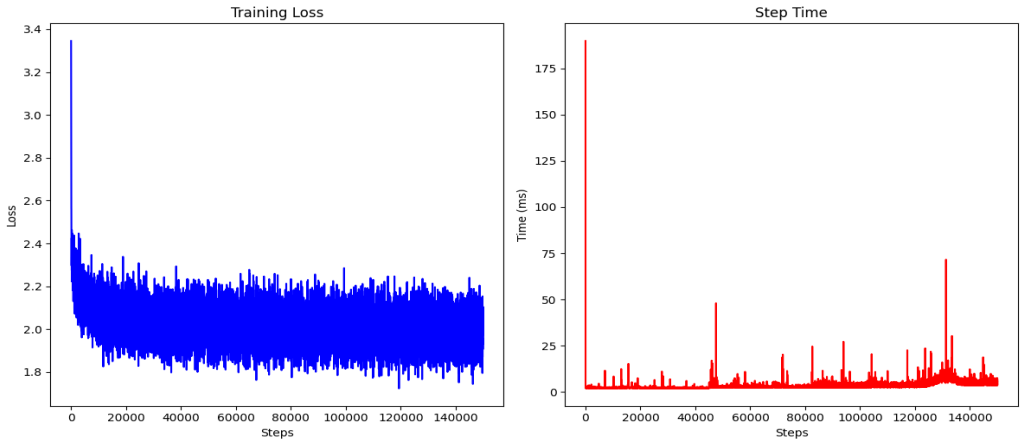

Figure 5.11: RNN: Train Loss

5.5.3 Test vs Train Loss Visualization

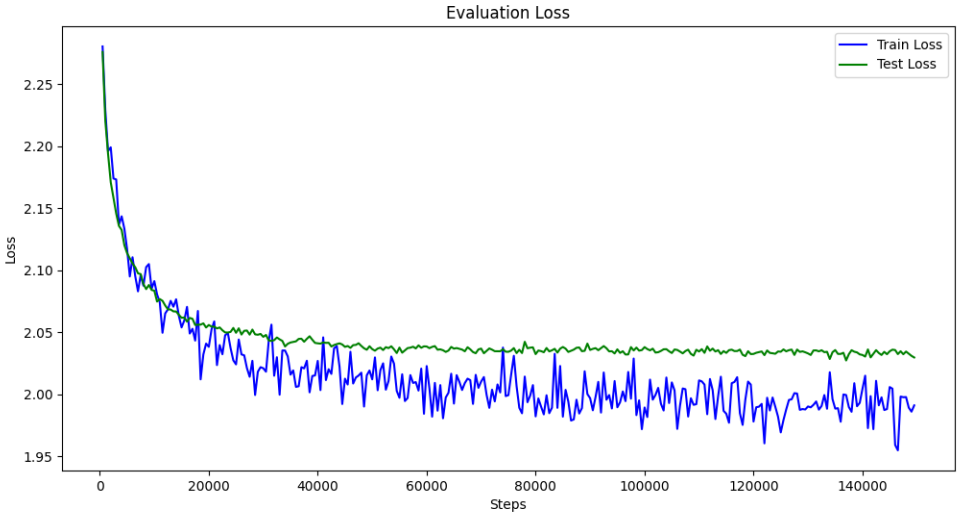

Figure 5.12: RNN: Train vs Test Loss

5.6 Model Evaluation

5.6.1 Interpreting Loss Metrics

The loss values in the RNN model training serve as an indicator of the model's performance, with lower values signifying better predictive accuracy. These values are particularly crucial in evaluating the model's ability to understand and generate language patterns. This model's average train loss was 2.01 whereas the average test loss was higher at 2.05 - highly becuase the model overfit the data but less than in the case of MLP model.

5.7 Results

The final stage of the RNN model involved generating names using the trained model. This process resulted in a mix of names from the training dataset and new, creatively generated names, demonstrating the model's generative capabilities.

Table 5.1: RNN: Final name samples

Category	**Samples**
In Train	devin, kaylie, rya, adelyne, jamariah, laya, ramiya, aftyn, kailee, kristy, amarie
In Test	(none)
New	abraden, alcijah, zanri, joo, priston, lyonio, otri, dhiana, mehim, mania, sullan, wyzler, nikiah, talid, sygen, dalkyn, kywen, yduniko, aiilala, anyse, skylinie, jailan, hayon, soubai, angeon, jerrissa, othilla, bethaelle, jabe, muzzir, foum, jaxlyl, jerjee, harm, kenzelyn, narsham, alexelle, fena, sojaya

We can see that the names have been improved greatly from where we began but there still remains room for improvement.

Chapter 6

Transformer Model

6.1 Introduction

In this chapter, I delve into the Transformer model, an influential architecture in the domain of deep learning, particularly for sequence modeling tasks. Unlike traditional RNNs, the Transformer model leverages attention mechanisms to process sequences, offering a more parallelized and efficient approach. This model forms a crucial part of my autoregressive character-level language model, designed to generate plausible name-like sequences.

6.2 Theoretical Foundations

6.2.1 The Transformer's Unique Approach

The Transformer model stands apart from traditional sequential models like RNNs by eschewing recurrence and instead using a mechanism called self-attention. This allows each position in the sequence to attend to all other positions in the same sequence, thereby enabling parallel processing and capturing long-range dependencies more effectively [4].

6.3 Implementation Insights

6.3.1 Data Processing for the Transformer

The data processing for the Transformer follows a similar approach to that of the RNN and MLP models. Names are transformed into sequences of integers representing characters. However, the Transformer processes these sequences through its self-attention mechanism, allowing it to simultaneously process all parts of the sequence [4].

6.3.2 Architectural Details

In my implementation, the Transformer model is constructed with configurable layers, embedding dimensions, and heads. The model configuration is defined using a ModelConfig class, allowing for flexibility and customization. An embedding layer converts character indices into high-dimensional vectors, followed by multiple self-attention layers. Each layer in the Transformer consists of a self-attention mechanism and a feed-forward neural network, with normalization and dropout applied at key points.

Figure 6.1: Transformer: Character embeddings visualization using t-SNE

The Python implementation leverages PyTorch's neural network module (torch.nn). The model parameters, such as the number of layers (n_layer=2), the size of embeddings (n_embd=32), and the number of attention heads (n_head=4), are specified as arguments in the ModelConfig dataclass and I implemented the GELU as my activation function.

```
config = ModelConfig(vocab_size=vocab_size, block_size=block_size,
n_layer=4, n_head=4, n_embd=64, n_embd2=64)
```

6.4 Visualization and Interpretation

6.4.1 Character Embedding and Attention Visualization

The Transformer model's understanding of characters and their contextual relationships is visualized using t-SNE, providing a 2D representation of the high-dimensional embedding space [4]. Similarly to RNN, the closer each character is to each other, the similar the model thinks they are contextually.

We notice from figure 6.1 and 5.1 that transformer's character embeddings are very similar to RNN model's and both differ from MLP's (figure 4.1) expressing how much more these models are learning than MLP.

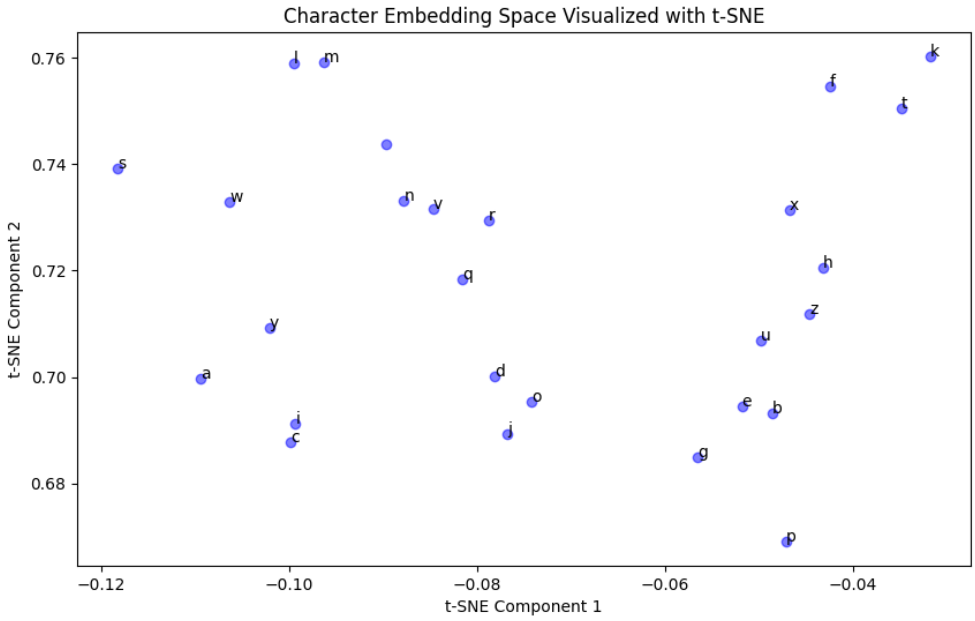

Figure 6.2: Transformer: Character embeddings visualization using t-SNE

6.4.2 Visualzing Model Responses to Names

To understand how the RNN model processes individual names, I examine both the attention weights and the output probability distributions for selected names.

Self-Attention Weights for 'Charlotte'

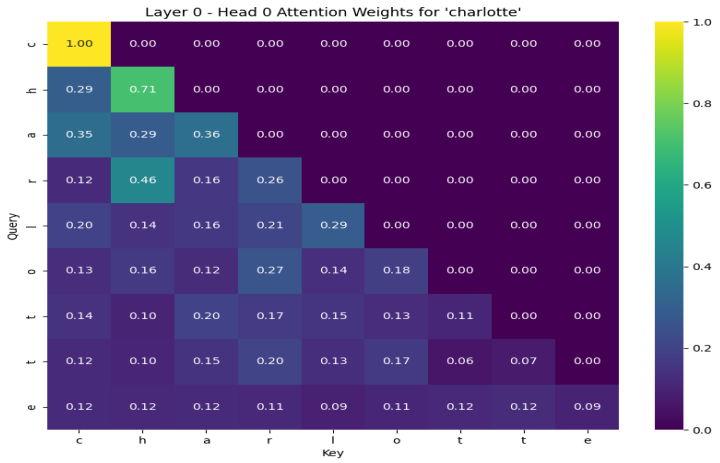

Figure 6.3: Transformer: Self-Attention Weights for 'Charlotte'.

Transformer: Output Probability Distribution for 'Charlotte'

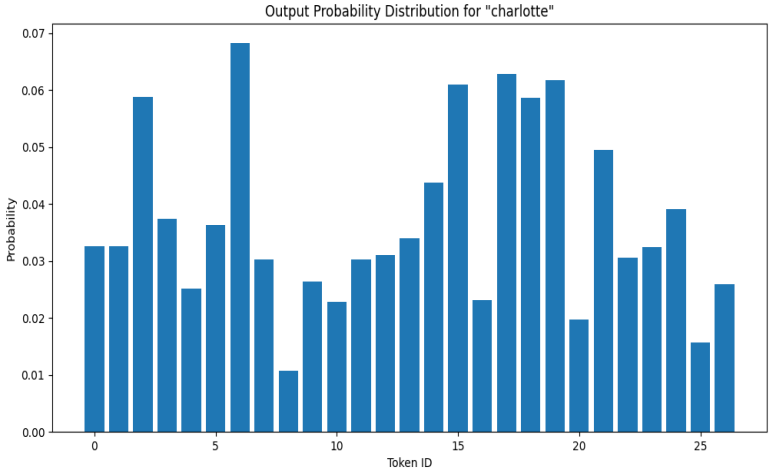

Figure 6.4: Output Probability Distribution for 'Charlotte'.

Transformer: Self-Attention Weights for 'Zoe'

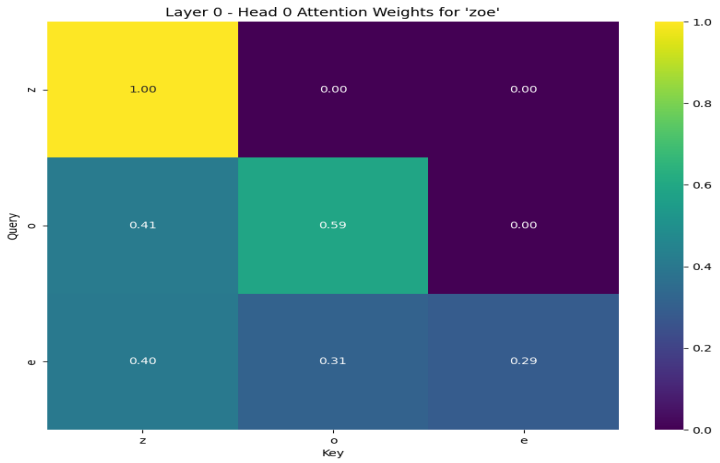

Figure 6.5: Transformer: Self-Attention Weights for 'Zoe'.

Output Probability Distribution for 'Zoe'

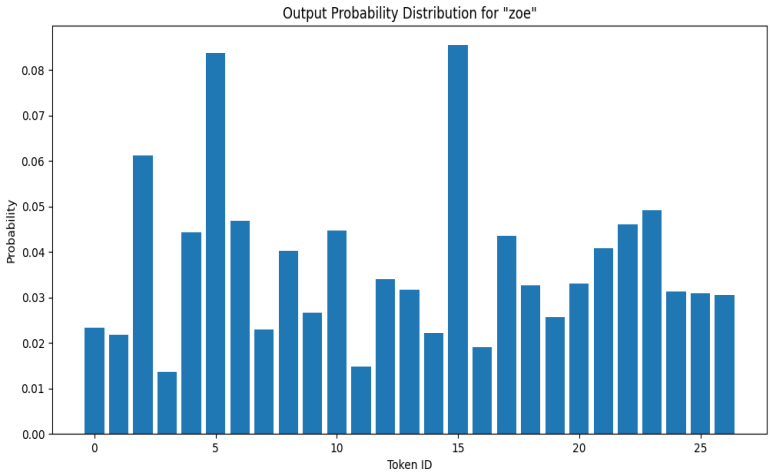

Figure 6.6: Transformer: Output Probability Distribution for 'Zoe'.

Self-Attention Weights for 'Alexandria'

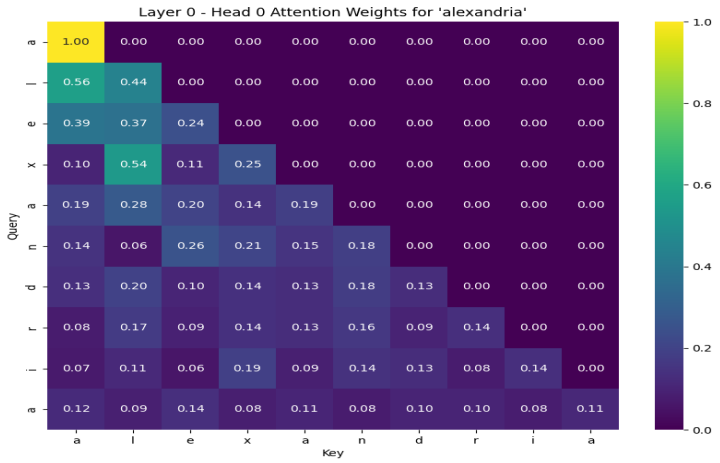

Figure 6.7: Transformer: Self-Attention Weights for 'Alexandria'.

Output Probability Distribution for 'Alexandria'

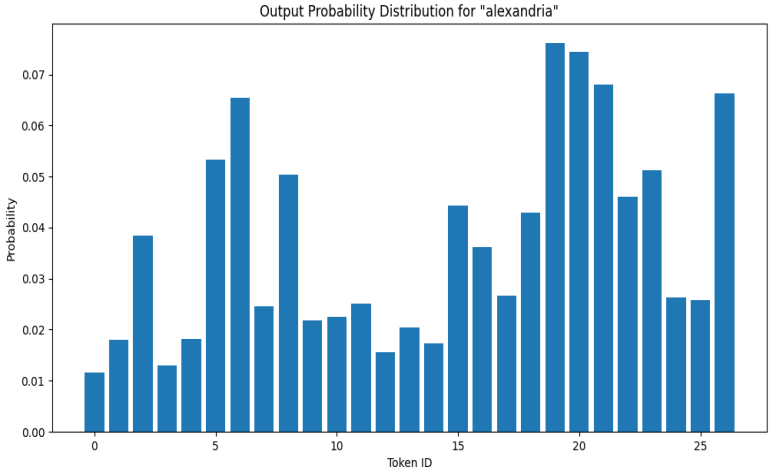

Figure 6.8: Transformer: Output Probability Distribution for 'Alexandria'.

Self-Attention Weights for 'Jacqueline'

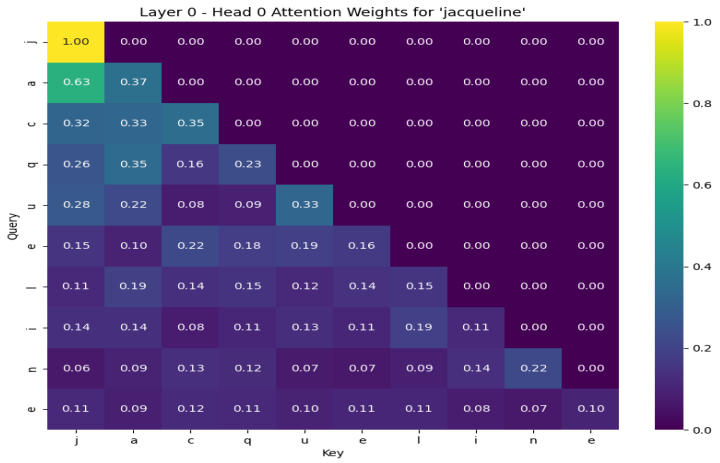

Figure 6.9: Transformer: Self-Attention Weights for 'Jacqueline'.

Output Probability Distribution for 'Jacqueline'

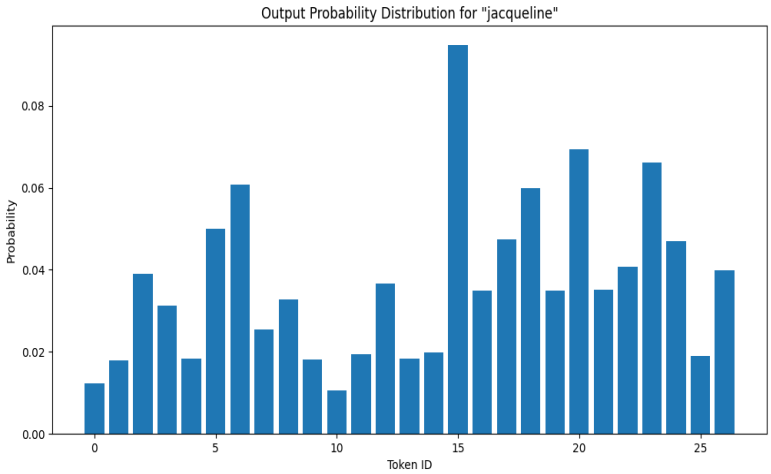

Figure 6.10: Transformer: Output Probability Distribution for 'Jacqueline'.

6.4.3 Self-Attention Weights Analysis

In my case, I have taken the self-attention weights heatmap for how my model's first layer's first head is calculating the attention weights for it's input sequence (here, a name like "charlotte").

In the heatmap for "charlotte", the letter 'c' at the beginning seems to be considered very important (bright yellow square), likely because it's the start of the word. You can see that 'h' also pays more attention to 'c', which might be because those two letters often come together in words.

In the "zoe" heatmap, the attention is heavily focused on 'z', the first letter (big bright yellow spot), which makes sense because 'z' is a unique letter and can change the sound of a word a lot. The letters 'o' and 'e' also pay attention to each other (blue and green squares), which could be because they often work together in English to make specific sounds.

For "alexandria," there's a lot of yellow and green throughout the heatmap, meaning the attention is spread out. The first letter 'a' gets a lot of focus from 'l' and 'e' (yellow spots), which could be because 'a' and 'l' often start words together, and 'e' often follows 'a' in English.

In "jacqueline," the first letter 'j' gets the most attention overall (big bright yellow spot), which is typical because the beginning of a word is important for understanding it. There are various other spots where the letters pay attention to each other, like 'q' and 'u' (because they're always together in English) and 'e' and 'l' (which might be because they make a common sound together).

6.5 Training Process Overview

I trained this model over 50,000 iterations multiple times trying to tune the hyperparameters for best output - least test loss all the while logging losses to visualise and help understand it's behaviour. The AdamW optimizer, known for its effectiveness in handling sparse gradients is also employed. The model's total parameters were 0.20 million or 204,544.

6.5.1 Training Log Structure

The training logs are structured to meticulously record vital details such as iteration steps, loss values at each step, and iteration time. This structured logging let's us analyze the decrease (or increase) in training and test loss over time - the following visualisations will help us with the same. In this case, the best test loss I achieved for this transformer on this dataset (without overfitting the data) was 1.97.

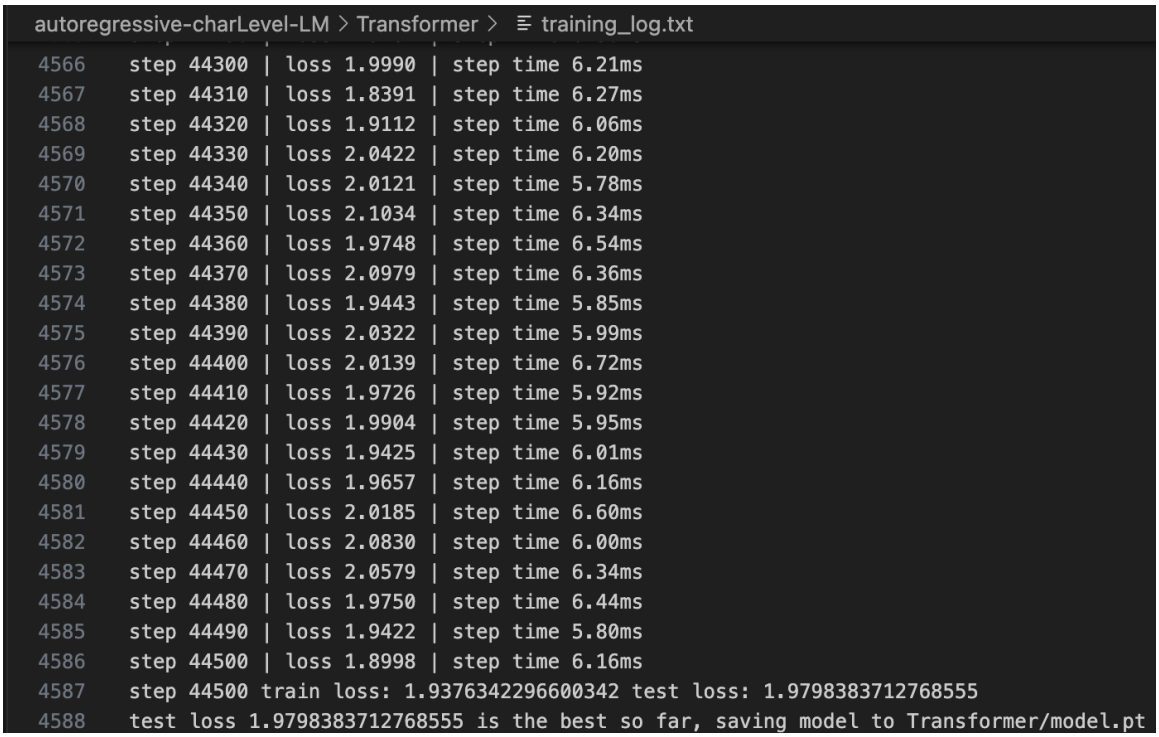

Figure 6.11: Transformer: Training logs excerpt

6.5.2 Training Loss Visualization

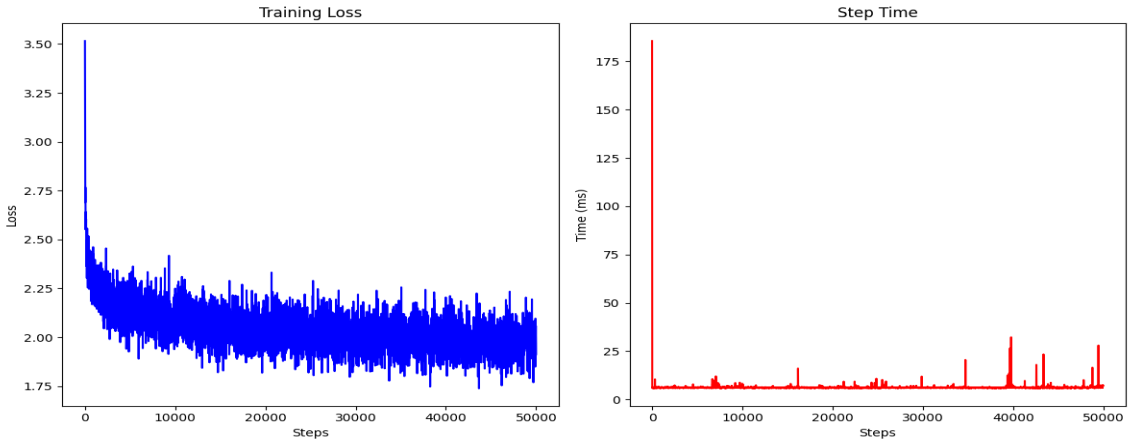

Figure 6.12: Transformer: Train Loss

6.5.3 Training and Test Loss Visualization

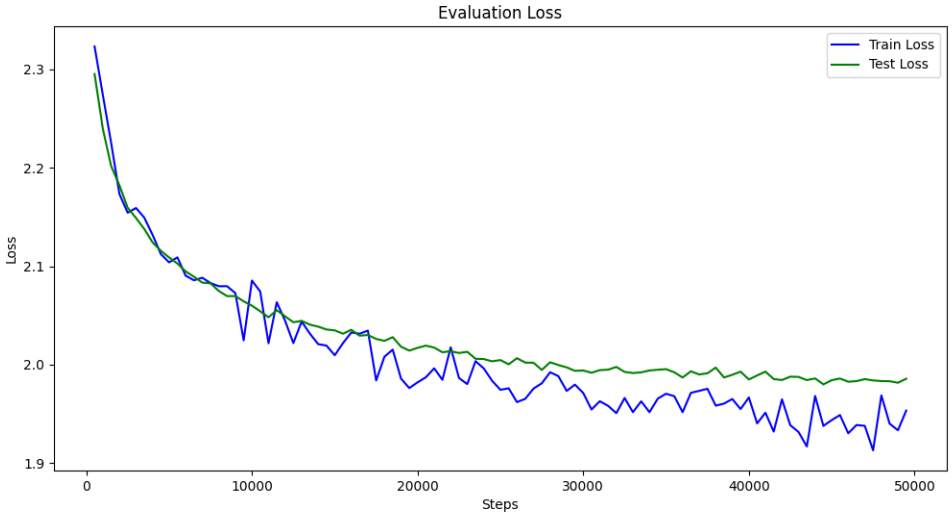

Figure 6.13: Transformer: Train vs Test Loss

6.6 Model Evaluation

6.6.1 Loss Metrics Interpretation

The loss metrics serve as a quantitative measure of the model's accuracy in character prediction. A lower loss indicates a higher proficiency of the model in generating believable and contextually appropriate names.

6.7 Results

The Transformer model's generative capability is showcased in the final stage, where it generates a mix of known and novel names. This demonstrates the model's ability to not just replicate but also creatively generate plausible name-like sequences.

Table 6.1: Transformer: Final name samples

Category	**Samples**
In Train	annali, keylen, xianna, bryson, casie, layten, lukas, damari
In Test	samira
New	adaura, myrant, noli, madson, lazenica, aiah, aalys, demania, milke, lyeicn, davisha, chugeyu, ayia, heandry, sulrin, nousi, wendenn, hiacipi, baita, zofire, raki, mian, aoan, gave, rainegan, consoj, raxyn, laelanne, sjalamie, jeffanvi, kamysen, rieley, caraldo, westestle, brahney, saidda, ati, fenya, nevily, khibeth, wilderfer

With these results we can see that the names have significanlty improved over not just the Bigram model but the MLP and RNN models. This is a tiny transformer with only 0.20 million parameters and a 32,000 word dataset hence, some names still sound off.

Chapter 7

Conclusion

7.1 Comparative Analysis

The **Bigram model**, while basic, laid the foundation for understanding character dependencies. However, its predictive power was limited, often generating less coherent names. The **MLP model** marked an improvement, leveraging higher-dimensional embeddings and nonlinear activations to capture more complex patterns. Yet, its lack of sequential memory restricted its performance. The **RNN model** introduced sequence memory, significantly enhancing name generation by retaining context from previous characters. It still grappled with long-term dependencies. The **Transformer model** emerged as the most proficient capturing long-range dependencies through self-attention mechanisms. It outperformed the other models in terms of the level of samples names produced but there still remains room for improvement.

7.2 Limitations and Future Work

While the models demonstrated promising results, certain limitations warrant attention:

Data Dependency: The quality of generated names is heavily dependent on the dataset's diversity and size.

Model Complexity: Larger models like the Transformer demand considerable computational resources.

Generalization: The models' ability to generalize beyond the training data scope is still an area for improvement.

Future work could explore:

Data Augmentation: Expanding the dataset with more diverse name samples.

Hybrid Models: Combining the strengths of different architectures.

Fine-tuning: Employing advanced training techniques for better generalization.

Application-Specific Models: Tailoring models for specific linguistic tasks or genres.

Thank you.

Bibliography

[1] Wataru Takano and Yoshihiko Nakamura. Bigram-based natural language model and statistical motion symbol model for scalable language of humanoid robots. In *2012 IEEE International Conference on Robotics and Automation*, pages 1232–1237, 2012.

[2] Vincent Jauvin Bengio, Ducharme. A neural probabilistic language model. In *Journal of Machine Learning Research 3*, volume 3, page 1137–1155, 2003.

[3] Burget Cernocky Khudanpur2 Mikolov, Karafiat. Recurrent neural network based language model. In *INTERSPEECH*, pages 1045–1048, 2010.

[4] Parmar Uszkoreit Jones Gomez Kaiser Polosukhin Vaswani, Shazeer. Attention is all you need. In *arXiv:1706.03762*, volume 7, 2017.