

## **End of Summer FURSCA Report**

### **Building a Language Model using Transformer Architecture**

**Medha Mohan**

#### **Introduction**

Thanks to the groundbreaking Transformer architecture introduced in the influential "Attention is All You Need" paper in 2017, the field of Natural Language Processing (NLP) has seen incredible advancements. The Transformer's self-attention mechanisms revolutionized the way computers capture long-range dependencies between words, overcoming limitations of earlier models.

This research focuses on developing a language model using the Transformer architecture. NLP involves enabling computers to comprehend and process human language in a manner similar to human understanding. Language modeling, a vital task in NLP, entails predicting the next word or character in a sequence based on contextual information. The primary objective is to develop a Generatively Pretrained Transformer (GPT) model that accurately predicts the next word in a given sequence.

#### **Results:**

The main steps of the project are:

##### Step 1: Data Preparation

For our project, we utilized a text dataset of Shakespeare's works for language modeling. We used Python's requests library to download a text file. Next, we preprocessed the text data to remove any unwanted characters, special symbols, or HTML tags using Python's string manipulation functions. The dataset was tokenized into individual characters to form the input sequences for the language model using Python's string operations. This ensured that each character in the text became a unique token for the language model. We split the dataset into training and validation sets using Python's data splitting libraries, such as scikit-learn's `train_test_split` function. The validation set was essential for evaluating the model's performance during training and ensuring that it generalizes well to unseen data. We then organized the data into manageable batches and chunks, making it perfect for feeding into the model during training. This organization maintained the flow of the data and ensured smooth processing. To help the model understand sentence boundaries during language generation, we introduced special tokens like "START" and "END." These tokens served as essential signposts for the model's comprehension, maintaining the continuity of sentences. Since sentences can vary in length, we wisely applied padding, ensuring that all sequences had the same length. This careful

handling of sentence length variations preserved the natural rhythm of the text. To optimize the training process, we implemented masking. This approach allowed the model to focus on valid tokens while gracefully ignoring padding tokens. The continuity of the learning process was kept intact by focusing on relevant information. When numerical features were present, we didn't leave them out. Instead, we applied normalization and scaling to standardize the data, harmonizing it with the textual information. Converting tokenized text data into numerical representations, such as word embeddings, was crucial. This numerical conversion maintained the continuity of the information while enabling efficient processing by the GPT model.

### Step 2: Baseline Bigram Model

To establish a baseline for our GPT model, we implemented a simple bigram language model. We used Python's NumPy library to create a n-gram model. This involved counting the occurrences of each character pair (bigram) in the training data. We created a dictionary to store the bigram frequencies in the training data and computed the probability distribution of the bigrams by dividing their counts by the total number of bigrams in the dataset. To then generate text, we sampled characters based on their probabilities, starting with a seed character. We used NumPy's random sampling functions to select characters according to the probability distribution of the bigrams. This baseline model helped us gain insights into language modeling principles and served as a reference for comparison as we built more complex models.

### Step 3: Self-Attention Mechanism

Implementing the self-attention mechanism was a critical step in building the GPT model. We used Python and NumPy to construct the self-attention module. This involved calculating attention scores for each character in the input sequence based on its similarity to other characters. We computed the attention scores using dot product attention, where the input sequence was multiplied with query, key, and value matrices to compute attention weights. We applied the softmax function to normalize the attention weights, obtaining the final attention probabilities for each character. The self-attention mechanism allowed the model to focus on relevant parts of the input sequence during language generation, capturing long-range dependencies between characters.

### Step 4: Positional Encoding

To incorporate positional information, we introduced positional encoding into the input embeddings. We used Python and NumPy to generate positional encoding vectors for each character position in the input sequence. The positional encoding vectors were added to the input embeddings before passing them through the self-attention mechanism. This addition allowed the model to understand the order and sequence of characters in the text, addressing the lack of inherent positional information in transformer-based architectures. Positional encoding was

crucial for the GPT model to differentiate characters based on their positions, enhancing its ability to generate contextually relevant text.

#### Step 5: Transformer Architecture

Building the Transformer architecture involved stacking multiple layers of self-attention and feedforward neural networks. We used Python and PyTorch, a deep learning framework, to implement the Transformer. Each layer consisted of a self-attention module, a feedforward neural network, and residual connections. Layer normalization was applied after each self-attention and feedforward layer to stabilize the training process. These architectural components allowed the model to capture complex dependencies between characters and improved the flow of gradients during training. The Transformer architecture formed the core of the GPT model, enabling it to generate coherent and contextually relevant text.

Right now we were able to complete only upto Step 5. Our future steps include:

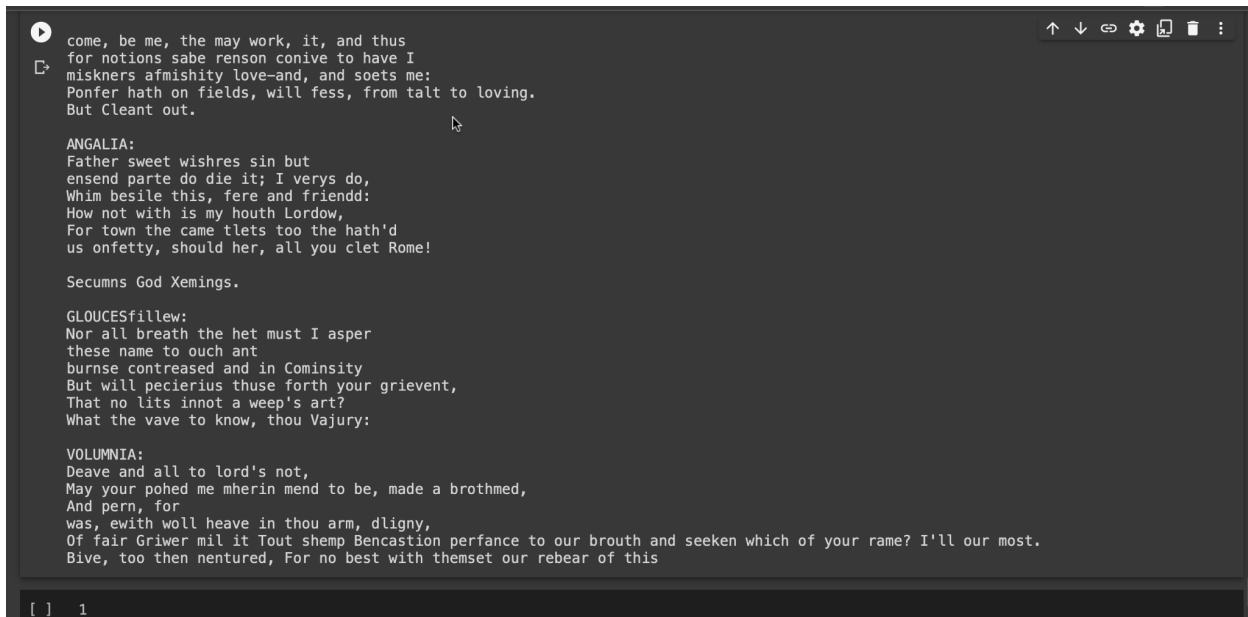
#### Step 6: Model Training and Optimization

For model training, we would want to use a stochastic gradient descent (SGD) or the Adam optimizer in PyTorch. We will then implement the language modeling loss, such as cross-entropy loss, to measure any discrepancy between the predicted characters and the ground truth characters during training. We would also like to tune the hyperparameters, such as learning rates, batch sizes, and the number of layers, to optimize the model's performance. We would also want to PyTorch's automatic differentiation capabilities to compute gradients efficiently, which will allow us to update the model's parameters during optimization. This will ensure that the GPT model can learn to generate text effectively from the input sequences and improve its performance over training iterations.

#### Step 7: Evaluation and Iteration

The trained GPT model will be evaluated on the validation set to assess its language generation capabilities and overall performance. We will then analyze the generated text, and check for coherence, contextuality, and overall fluency. Based on the evaluation results, we will iterate over the model architecture and hyperparameters to further enhance its performance. We will then finish by conducting multiple training runs, adjusting parameters, and making architectural changes as needed. This iterative process will allow us to improve the GPT model's language generation capabilities and address any issues or limitations.

A picture of our output:



```
▶ for notions sabe renson conive to have I
  miskners afmishity love-and, and soets me:
  Ponfer hath on fields, will fess, from talt to loving.
  But Cleant out.

ANGALIA:
Father sweet wishres sin but
ensend parte do die it; I verys do,
Whim besile this, fere and friendd:
How not with is my houth Lordow,
For town the came tlets too the hath'd
us onfetty, should her, all you clet Rome!

Secumns God Xemings.

GLOUCESfillow:
Nor all breath the het must I asper
these name to ouch ant
burnse contreased and in Cominsity
But will pecierius thuse forth your grievent,
That no lits innot a weep's art?
What the vave to know, thou Vajury:

VOLUWNIA:
Deave and all to lord's not,
May your pohed me mherin mend to be, made a brothmed,
And pern, for
was, ewith woll heave in thou arm, dligny,
Of fair Griwer mil it Tout shemp Bencastion perfance to our brouth and seeken which of your rame? I'll our most.
Bive, too then nentured, For no best with themset our rebear of this

[ ] 1
```

## Conclusion

For future work this fall we would like to complete our remaining steps. Once we successfully finish this we want to take what we have learned from this and use it build the chatbot. This hands-on project provided us with a deep understanding of transformers, self-attention mechanisms, and language modeling, and equipped us with valuable practical experience in deep learning and natural language processing. It laid the groundwork for future exploration and more advanced projects in the field of AI and machine learning.

I would like to thank FURSCA, Elizabeth Palmer and Renee Kreger for a wonderful summer research experience. I would also like to thank Dr. Bollman for guiding us.

### To the Jean Bengel Laughlin and Sheldon Laughlin Endowment for Student Research

Thank you for giving me the opportunity to participate in Albion's FURSCA program this summer. This summer provided many benefits that will greatly influence my future education and research endeavors. Thank you!