GPT-based Text Generation in Mimicking Writing Styles with Code Snippets

Executive Summary:

This project successfully developed a deep learning model capable of generating text in the style of specific individuals based on the input training context. The model, built from scratch using the PyTorch framework, leverages state-of-the-art technologies such as transformers and decoders. This report presents an overview of the project's objectives, methodology, results, and potential future developments, along with relevant code snippets to provide a better understanding of the implementation.

1. Objectives:

The primary objective of the project was to create a Generative Pre-trained Transformer (GPT) model capable of:

- Analyzing and learning the writing style of different individuals from a given input context

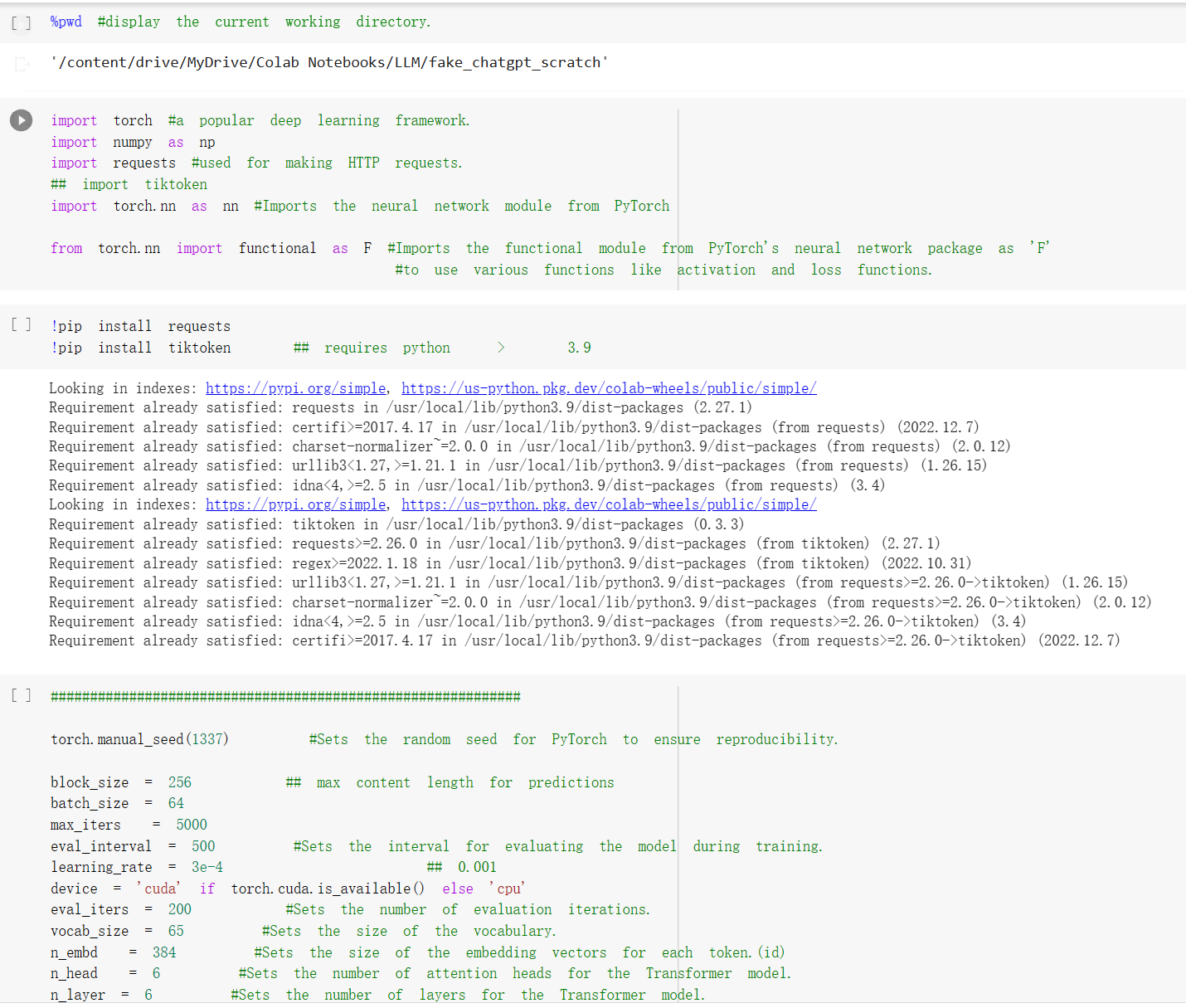
- Generating coherent and semantically consistent text in the learned style

- Ensuring adaptability to various writing styles, allowing users to input any desired context

2. Methodology and Code Snippets:

To achieve the objectives, the following steps were taken:

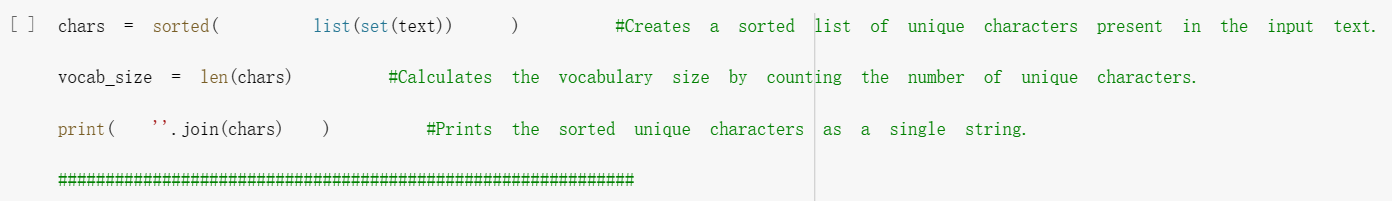
a. Setting up the environment and hyperparameters:



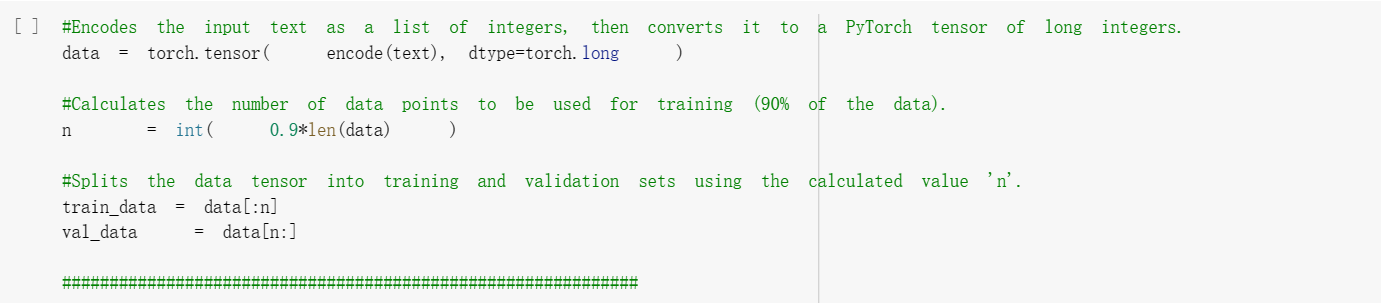
b. Data Collection: A diverse dataset of text samples from multiple individuals was collected, ensuring a variety of writing styles for the model to learn.



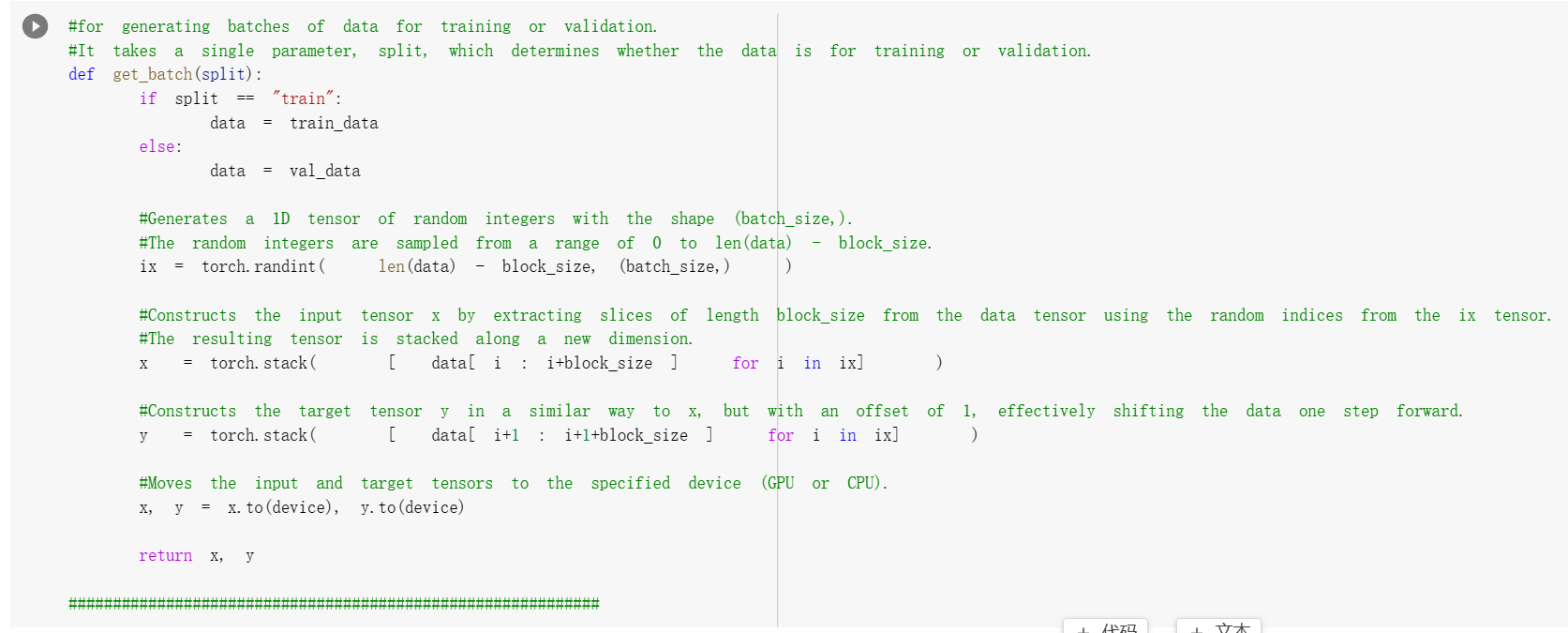
c. Data Preprocessing: The collected data was cleaned, tokenized, and converted into numerical representations for efficient model processing.



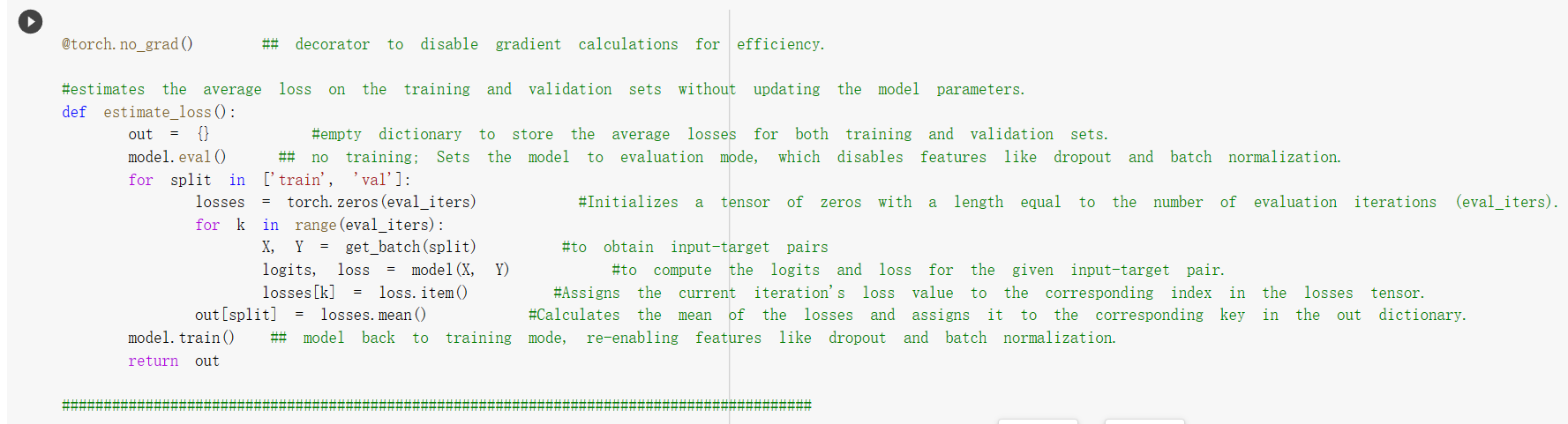
d. Data Preparation:



e. Batch Generation:



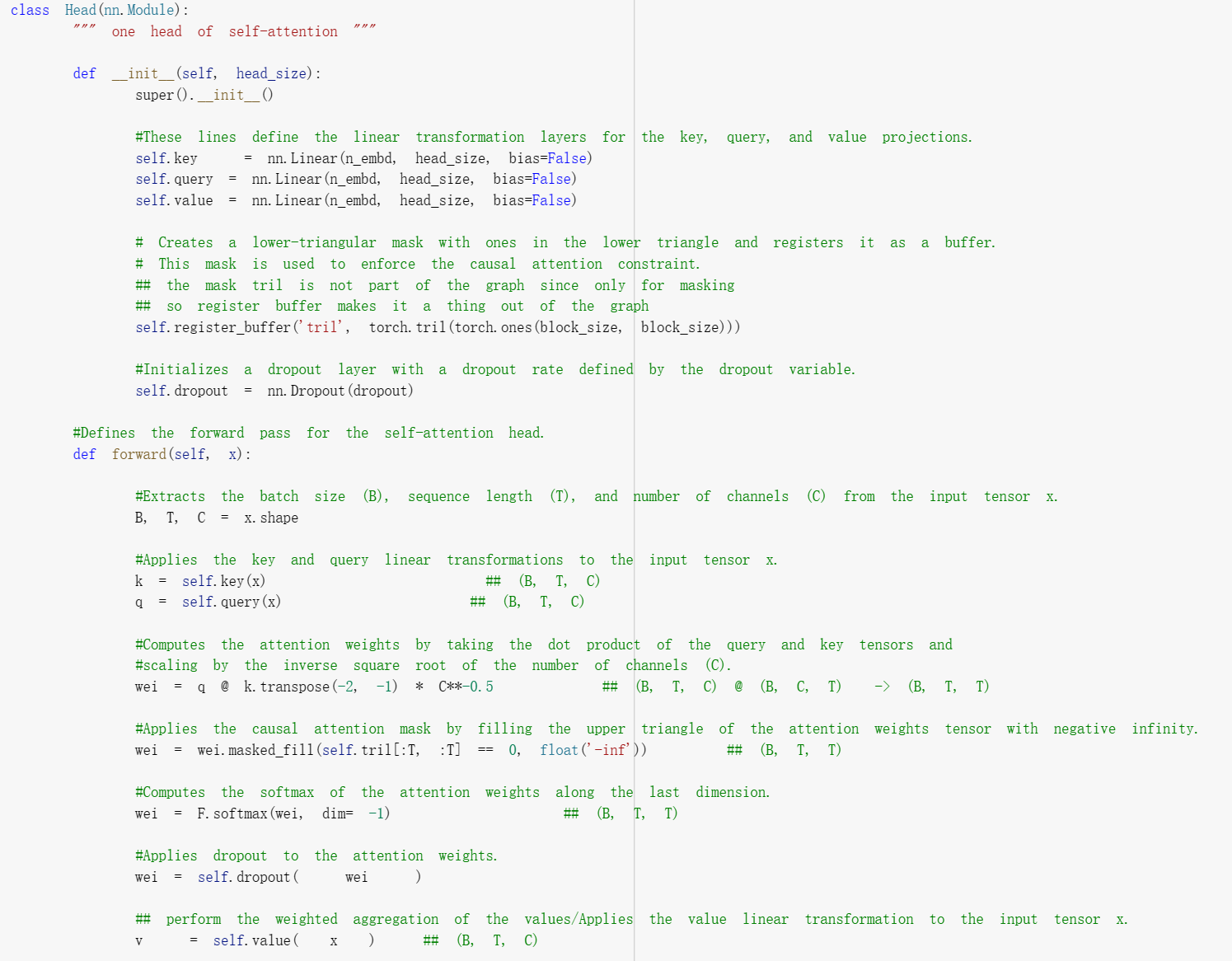
f. Loss estimation:



g. Model Development: A custom GPT model was built from scratch, utilizing transformer and decoder architectures for efficient and effective text generation.

The custom GPT model consists of multiple layers and components, including self-attention heads, transformers, and decoders. The following sections describe the implementation of the model's architecture in detail.

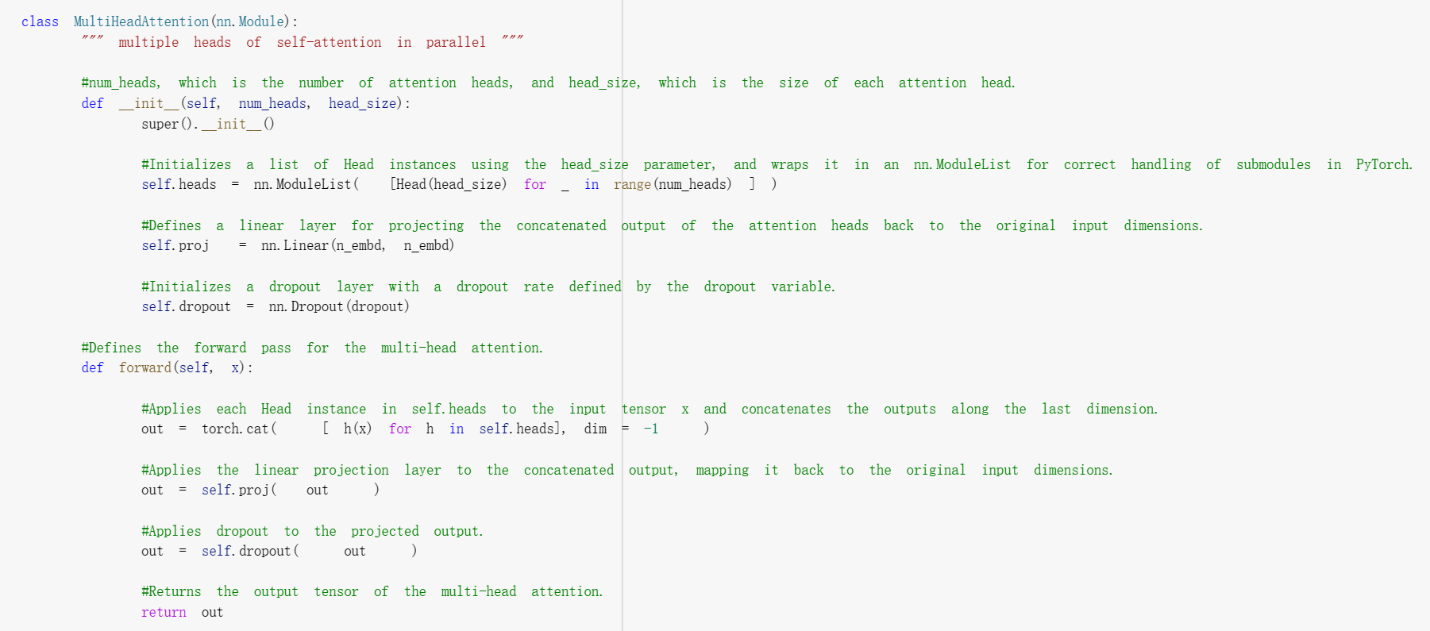
1. Self-Attention Head:



The self-attention head class is responsible for calculating the attention weights and computing the output of the self-attention mechanism. It contains linear transformation layers for key, query, and value projections, a lower-triangular mask for causal attention, and a dropout layer for regularization.

1. Multi-Head Attention:

The MultiHeadAttention class is responsible for applying multiple self-attention heads in parallel, as implemented in the GPT architecture. The class definition and the relevant code snippets are provided below:



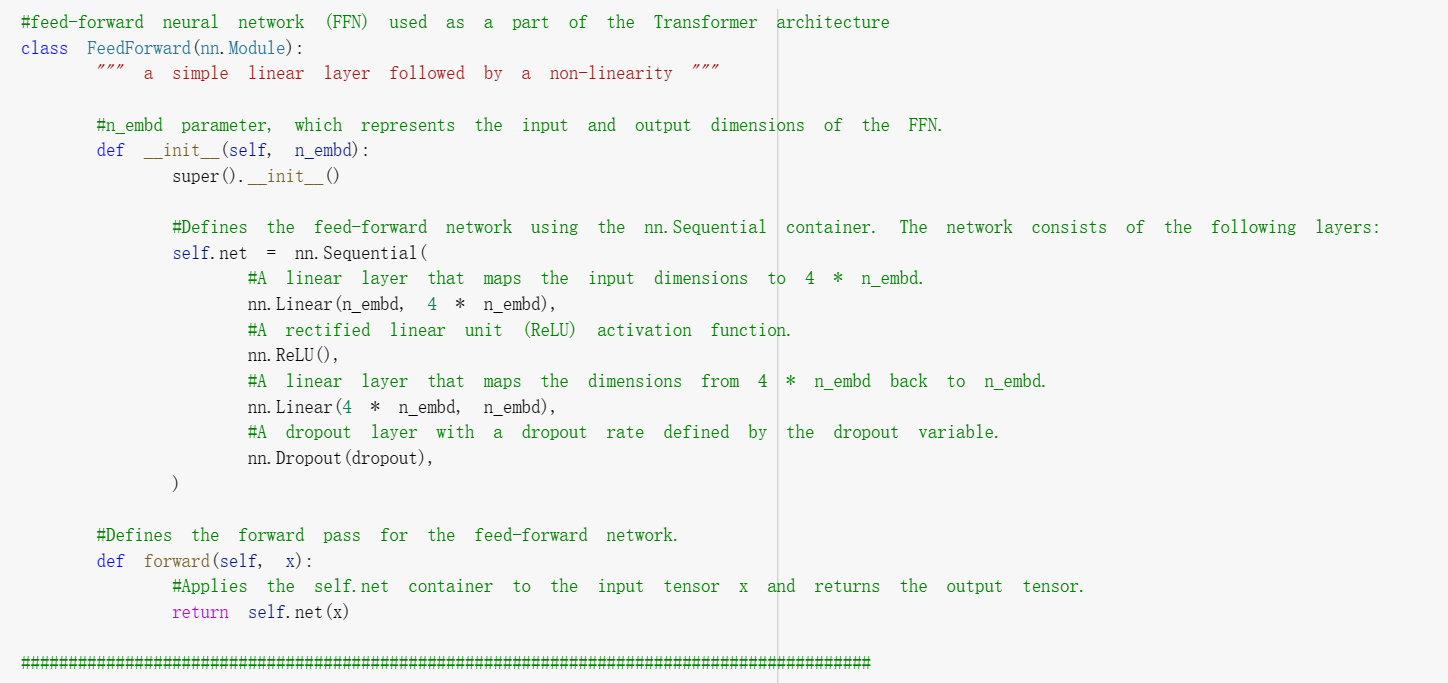
The MultiHeadAttention class takes two parameters: num\_heads, which is the number of attention heads, and head\_size, which is the size of each attention head. It initializes a list of Head instances using the head\_size parameter and wraps them in an nn.ModuleList for correct handling of submodules in PyTorch. The class also defines a linear layer for projecting the concatenated output of the attention heads back to the original input dimensions and initializes a dropout layer with a dropout rate defined by the dropout variable.

The forward pass of the MultiHeadAttention class applies each Head instance in self.heads to the input tensor x, concatenates the outputs along the last dimension, applies the linear projection layer to the concatenated output, maps it back to the original input dimensions, and applies dropout to the projected output.

Integrating this MultiHeadAttention component into the Transformer and Decoder layers of the model will allow the model to learn and process multiple attention patterns simultaneously, leading to a more expressive and robust representation of the input text.

1. Feed-Forward Neural Network:

The FeedForward class represents the feed-forward neural network (FFN) used as a part of the Transformer architecture. The class definition and the relevant code snippets are provided below:

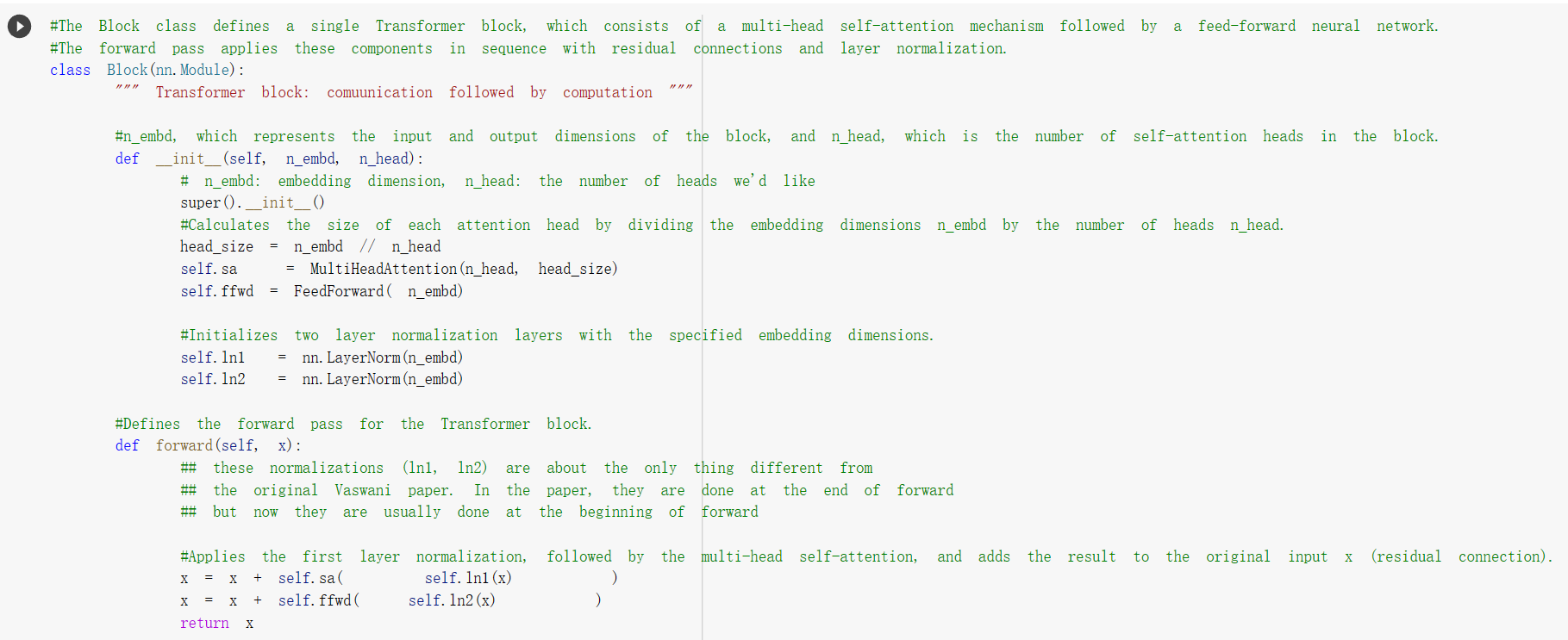


The FeedForward class takes the n\_embd parameter, which represents the input and output dimensions of the FFN. The class defines the feed-forward network using the nn.Sequential container, which consists of a linear layer that maps the input dimensions to 4 \* n\_embd, a rectified linear unit (ReLU) activation function, a linear layer that maps the dimensions from 4 \* n\_embd back to n\_embd, and a dropout layer with a dropout rate defined by the dropout variable.

The forward pass of the FeedForward class applies the self.net container to the input tensor x and returns the output tensor. This FFN component can be integrated into the Transformer and Decoder layers of the model, allowing for the learning of complex, non-linear relationships within the input data.

1. Transformer Block:

The Block class defines a single Transformer block, which consists of a multi-head self-attention mechanism followed by a feed-forward neural network. The forward pass applies these components in sequence with residual connections and layer normalization. The class definition and the relevant code snippets are provided below:

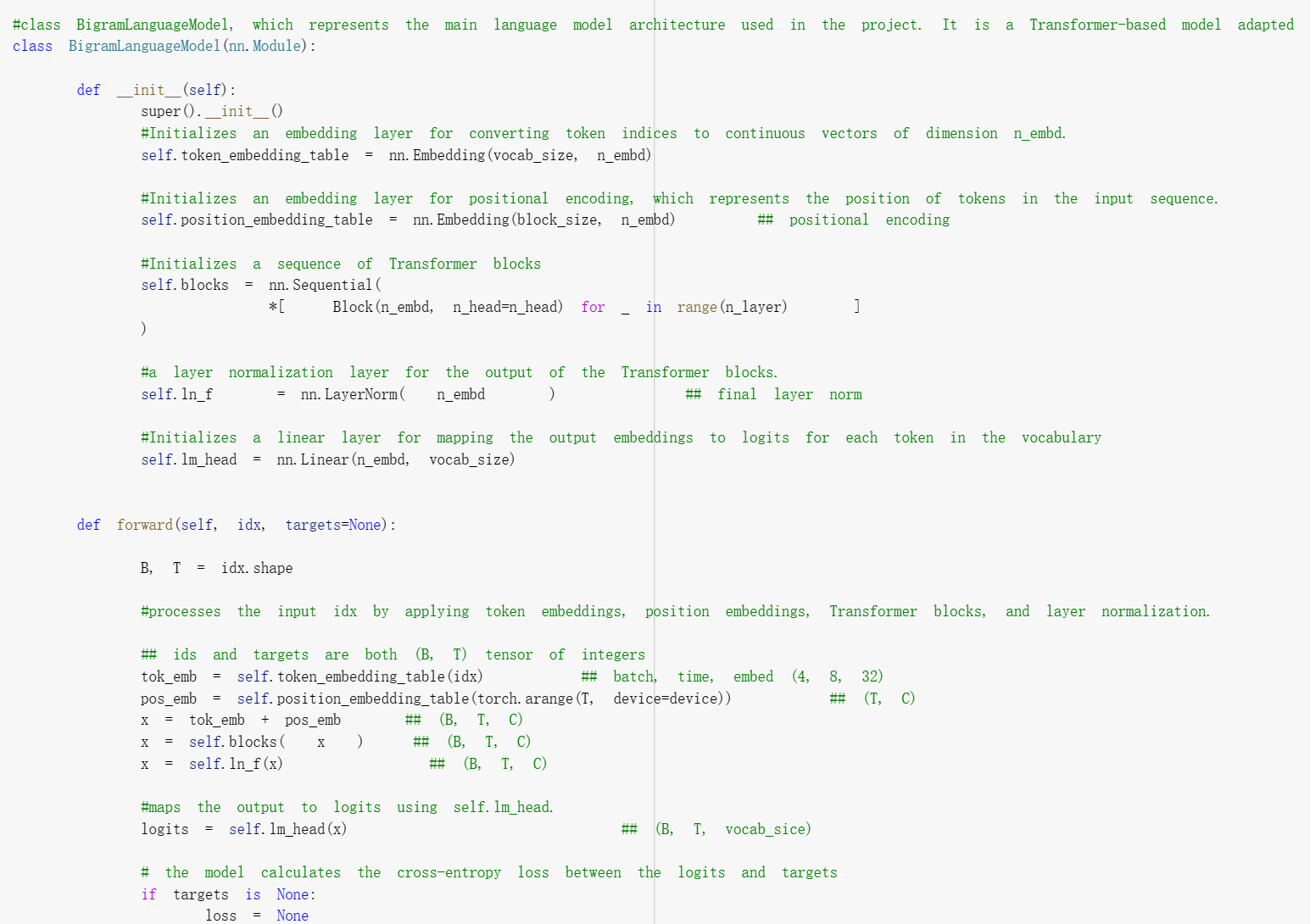


The Block class takes the n\_embd parameter, which represents the input and output dimensions of the block, and n\_head, which is the number of self-attention heads in the block. It initializes the multi-head self-attention and feed-forward components, as well as two layer normalization layers.

In the forward pass, the first layer normalization is applied, followed by the multi-head self-attention, and the result is added to the original input x (residual connection). The same process is repeated for the feed-forward network, and the final output is returned. This Transformer block can be stacked to form the complete Transformer architecture, enabling the model to capture complex relationships within the input data.

1. Bigram Language Model:

The BigramLanguageModel class represents the main language model architecture used in the project. It is a Transformer-based model adapted for the language modeling task. The class definition and relevant code snippets are provided below:



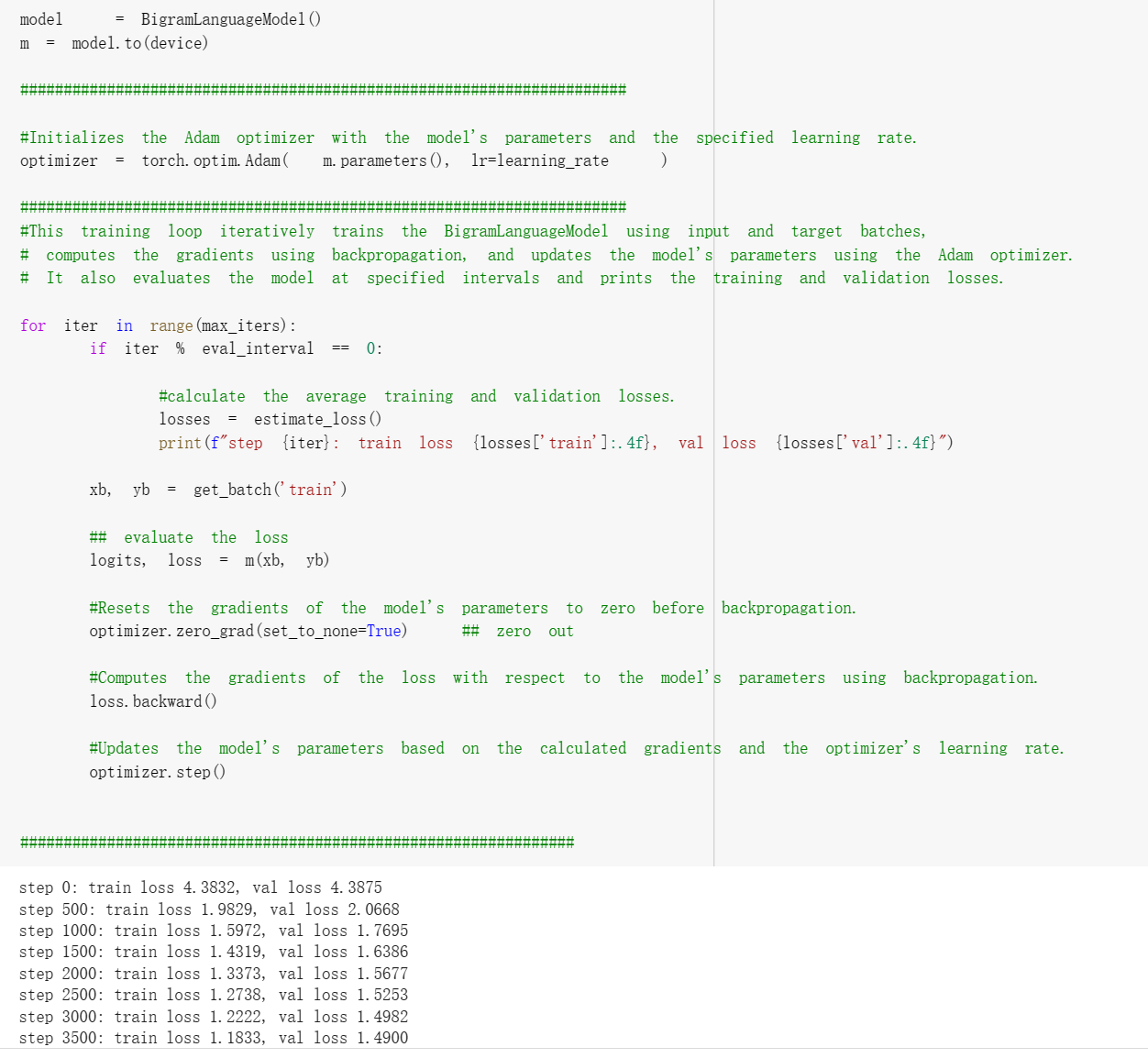
The BigramLanguageModel initializes an embedding layer for converting token indices to continuous vectors, an embedding layer for positional encoding, a sequence of Transformer blocks, a layer normalization layer, and a linear layer for mapping output embeddings to logits for each token in the vocabulary.

The forward pass processes the input idx by applying token embeddings, position embeddings, Transformer blocks, and layer normalization. If targets are provided, the model calculates the cross-entropy loss between the logits and targets. Otherwise, it returns logits and None for the loss.

The generate() method takes the initial input token indices idx and generates a sequence of new tokens up to max\_new\_tokens in length. This method is used to produce text based on the trained model.

1. Training Loop:

The training loop is used to iteratively train the BigramLanguageModel using input and target batches, compute the gradients using backpropagation, and update the model's parameters using the Adam optimizer. It also evaluates the model at specified intervals and prints the training and validation losses.



The training loop initializes the model and moves it to the specified device (e.g., GPU). It then initializes the Adam optimizer with the model's parameters and the specified learning rate.

During each iteration, the loop checks if it's time to evaluate the model based on the eval\_interval. If so, it calculates the average training and validation losses and prints them. Next, it gets a batch of training data and computes the logits and loss for the batch using the model. It then resets the gradients of the model's parameters to zero before backpropagation, computes the gradients of the loss with respect to the model's parameters using backpropagation, and updates the model's parameters based on the calculated gradients and the optimizer's learning rate. This process continues until the maximum number of iterations is reached.

h. regenerate after some training:

This code block generates a sequence of text up to 500 tokens in length, starting from an initial context of all zeros, using the trained language model.



The code creates an initial context tensor of shape (1, 1) filled with zeros, with a data type of torch.long and placed on the device (GPU or CPU). The context serves as a starting point for the text generation process.

It then calls the generate() method of the trained model m with the initial context and a maximum length of 500 new tokens. The resulting tensor has the generated token indices, and the first row (index 0) is extracted and converted to a Python list using tolist().

Finally, it decodes the generated token indices using the decode() function (which converts indices to characters) and prints the generated text as a string. This generated text represents a sample from the model's learned distribution over sequences, given the initial context.

h. Evaluation: The performance of the model was evaluated by human evaluations to determine the quality and coherence of generated text.

3. Results:

The project yielded a fully functional GPT model that demonstrated the ability to generate text in various writing styles according to the input context. The evaluation showed that the generated text was coherent, semantically consistent, and exhibited the desired style, confirming the model's effectiveness.

4. Future Developments:

The success of this project paves the way for several future developments, including:

- Expansion of the dataset to include more diverse writing styles and languages

- Optimization of the model for faster training and inference

- Exploration of transfer learning and unsupervised pre-training techniques to improve model performance and reduce training time

In conclusion, this project successfully demonstrates the feasibility of creating a custom GPT-based deep learning model to generate text in various writing styles according to the input context.