# GPT-Based Language Model

project report submitted by

**PANDI SRIDINESH**
(22MA60R05)

Under the guidance of

**Dr. Prateep Chakraborty**

In partial fulfillment of the the requirements for the degree of

**MASTER OF TECHNOLOGY**

**COMPUTER SCIENCE AND DATA PROCESSING**
(2022-2024)

At


**INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR**

**DEPARTMENT OF MATHEMATICS,**
**INDIAN INSTITUTE OF TECHNOLOGY,**
**KHARAGPUR, WEST BENGAL, 721302.**

# CERTIFICATE

This is to certify that the project entitled **GPT-Based Language Model** submitted by **PANDI SRIDINESH (22MA60R05)** is a bonafide project work, done under my supervision. It is being submitted in partial fulfilment of the requirements of Master of Technology in Computer Science and Data processing at Indian Institute of Technology, Kharagpur.

**Dr. Prateep Chakraborty**
Assistant Professor,
Department of Mathematics,
Indian Institute of Technology,
Kharagpur.

Date: 11-04-2024,
Place: Kharagpur.

# DECLARATION

I, **PANDI SRI DINESH**, hereby declare that the material presented in the project work entitled **GPT-Based Language Model** is an original record of studied and bonafide work carried out by me under the guidance of Dr. Prateep Chakraborty, Assistant Professor, Department of Mathematics, Indian Institute of Technology, Kharagpur and has not been submitted by me elsewhere for the award of any degree, diploma, title or recognition before.

**PANDI SRI DINESH**
**(22MA60R05).**

Date: 11-04-2024,
Place: Kharagpur.

# Acknowledgement

I would like to express my special thanks to my mentor Dr. Prateep Chakraborty for all the discussions and support.

PANDI SRIDINESH

# GPT-Based Language Model

**Abstract**

In this project, we delve into the intricate workings of self-attention and multi-head attention mechanisms, fundamental components that underpin the cutting-edge Generative Pre-trained Transformer (GPT) architecture. By seamlessly integrating these mechanisms with the versatility of Multi-layer Perceptron (MLP), we engineer a powerful framework for character-level language model to generate coherent text based on a given input. Moreover, our research delves into the practical aspects of training the GPT architecture on various datasets, including the compact yet rich Tiny Shakespeare corpus and the expansive Spotify Million Songs dataset.

## 1. Introduction

The ability to understand and generate human-like text has long been a cornerstone of artificial intelligence research, with significant advancements witnessed in recent years. Central to this progress are sophisticated language models that leverage innovative architectures and learning mechanisms. In this project, we embark on a comprehensive exploration of one such paradigm-shifting architecture: the Generative Pre-trained Transformer (GPT).

At the heart of GPT lies a fusion of self-attention and multi-head attention mechanisms, augmented by the adaptability of Multi-layer Perceptron (MLP). These components synergize to form a robust framework capable of grasping intricate linguistic patterns and generating coherent text. Our endeavor is to delve deep into the inner workings of GPT, shedding light on its architecture, functionality, and practical applications.

A key aspect of our project involves the construction of a character-level language model using GPT. A language model is a statistical model that is designed to predict the likelihood of a sequence of words occurring in a natural language. Essentially, it's a model trained on a large corpus of text data to understand and generate human-like text. Language models are fundamental

to various natural language processing (NLP) tasks such as speech recognition, machine translation, sentiment analysis, and text generation. This model represents a significant advancement in natural language processing, offering the ability to generate text with remarkable fluency and coherence.

Furthermore, our research shows the practical aspects of training the GPT architecture on various datasets, including the compact yet rich Tiny Shakespeare corpus and the expansive Spotify Million Songs dataset.

The Tiny Shakespeare dataset is a corpus consisting of the complete works of William Shakespeare, one of the most renowned playwrights and poets in the English language. This dataset is often used in natural language processing (NLP) and machine learning research as a benchmark for text generation tasks due to its rich linguistic content and historical significance. Through training our Generative Pre-trained Transformer (GPT) model on the Tiny Shakespeare dataset, our aim is to generate text reminiscent of the style and language characteristic of William Shakespeare's literary works.

The Spotify Million Songs dataset is a comprehensive collection of audio features and metadata associated with over one million songs available on the Spotify music streaming platform. This dataset is a valuable resource for music-related research and analysis, providing detailed information about a diverse range of musical tracks across various genres, artists, and time periods. Through training our Generative Pre-trained Transformer (GPT) model on the Spotify Million Songs dataset, our objective is to generate musical compositions reminiscent of the diverse array of songs available on the Spotify platform.

Language modeling involves several key steps to develop a model capable of understanding and generating human-like text. Firstly, data collection is crucial, wherein a diverse corpus of text data is gathered from various sources, representing the linguistic patterns and styles relevant to the task at hand. Following this, data preprocessing is performed to clean and tokenize the text, converting it into a format suitable for training. Next, a suitable architecture for the language model is chosen, which may include traditional statistical models or modern deep learning architectures such as recurrent neural networks (RNNs), long short-term memory (LSTM) networks, or transformers. The chosen architecture is then trained on the preprocessed data, optimizing its parameters to minimize a defined loss function, typically measuring the model's ability to predict the next word in a sequence. During training, techniques such as dropout regularization and gradient clipping may be employed to prevent overfitting and stabilize training. Once trained, the model can

be evaluated on a separate validation dataset to assess its performance, and fine-tuned if necessary. Finally, the trained language model can be utilized for various downstream tasks, including text generation, completion, summarization, translation, sentiment analysis, and more, thereby demonstrating its utility in natural language processing applications.

In this project report, we will meticulously navigate through each of the aforementioned steps involved in language modeling, employing the renowned GPT architecture as our primary model. However, before embarking on this comprehensive exploration, it is imperative to contextualize our journey by introducing pivotal events in the evolution of Natural Language Processing (NLP) that have paved the way for the prominence of language models. By tracing the significant milestones and breakthroughs in the field, we aim to elucidate the transformative impact of these developments, setting the stage for our in-depth examination of language modeling techniques and applications.

## 2. Important events in NLP and rise of language models

Natural Language Processing (NLP) is a branch of artificial intelligence (AI) that focuses on enabling computers to understand, interpret, and generate human language in a way that is both meaningful and contextually relevant. It encompasses a range of tasks and techniques aimed at bridging the gap between human communication and computational analysis.

NLP primarily focuses on analyzing and understanding text data, which inherently possesses a sequential structure. Throughout history, addressing the challenges posed by sequential data has been a central concern in various fields. In the realm of NLP, Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks have emerged as foundational tools for processing sequential data effectively.

The sequential nature of text data presents unique challenges, as each word or token in a sentence depends on the preceding words for context and meaning. RNNs, with their recurrent connections, are well-suited for capturing these temporal dependencies by maintaining an internal state that evolves over time as new inputs are processed. This allows RNNs to model sequential patterns and context, making them valuable tools for tasks such as language modeling, sentiment analysis, machine translation, and text generation.

In terms of mapping inputs to outputs, Recurrent Neural Networks (RNNs) can be categorized based on their input-output relationships. Here are some common types:

*1. One-to-One (1-1):*

In this configuration, a single input is mapped to a single output. Standard feedforward neural networks fall under this category. RNNs are not typically used in this configuration, as their strength lies in handling sequential data. For instance Image classification is one application of this architecture.

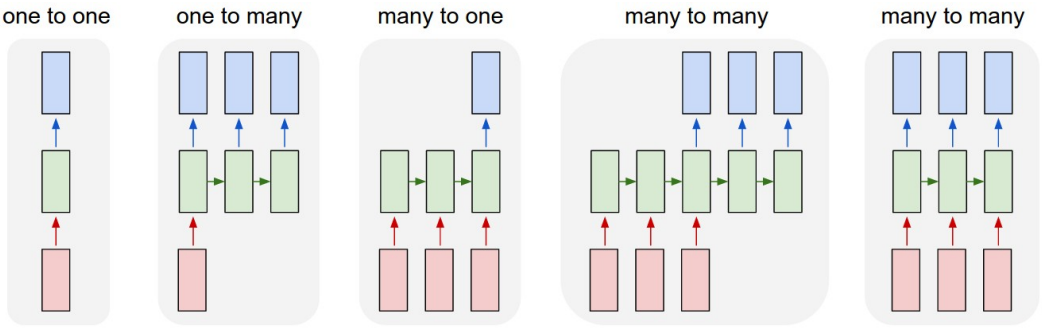


Figure 1: Types of RNNs

*2. One-to-Many (1-M):*

In this configuration, a single input is mapped to multiple outputs. The most famous application of this architecture is image caption generator, where you will take image as an input and output its apt caption.

*3. Many-to-One (M-1):*

In this configuration, multiple inputs are mapped to a single output. For example, sentiment analysis where the sentiment of a sentence is predicted based on multiple words as input. This configuration is commonly used in tasks such as sentiment analysis, where the entire sequence is processed to produce a single output.

*4. Many-to-Many (M-M):*

In this configuration, multiple inputs are mapped to multiple outputs. There are further subcategories within this configuration:

*4.1. Synchronous Many-to-Many:*

Each input corresponds to an output at the same time step. For example, part-of-speech tagging.

*4.2. Asynchronous Many-to-Many:*

Inputs and outputs are not aligned in time. For example, machine translation.

Many of the advanced techniques in Natural Language Processing (NLP) have been pioneered through the exploration of Asynchronous many-to-many Recurrent Neural Network (RNN) problems. Within this framework, machine translation has emerged as a pivotal domain, catalyzing significant advancements that ultimately culminated in the development of transformers. The inherently complex nature of machine translation tasks, involving the conversion of text from one language to another while preserving semantic meaning and syntactic structure, has served as a fertile ground for innovation.

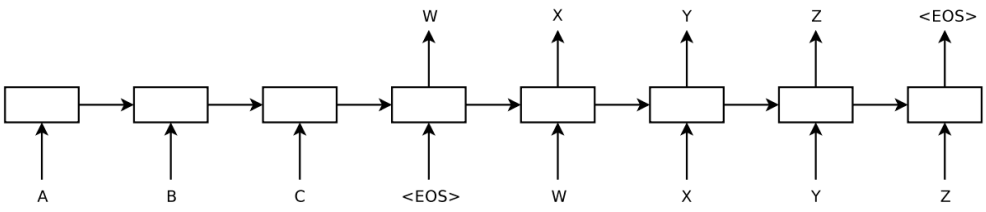


Figure 2: Encoder-Decoder architecture

In 2014, a significant milestone was achieved in the realm of handling asynchronous many-to-many Recurrent Neural Network (RNN) problems with the invention of the encoder-decoder architecture (Figure 2) by Ilya Sutskever and colleagues[1]. This innovative architecture marked a fundamental shift in the approach to sequence modeling, particularly in tasks where the input and output sequences are of varying lengths or are asynchronous in nature. The encoder-decoder architecture introduced a novel framework whereby an encoder network processes the input sequence into a fixed-length vector representation, capturing the essential features and context of the input. Subsequently, a decoder network generates the output sequence based on this encoded representation, effectively decoupling the lengths of the input and output sequences and enabling the model to handle asynchronous many-to-many relationships. This breakthrough paved the way for significant advancements in various domains, including machine translation, speech recognition, and text summarization, by providing a versatile and effective framework for processing sequential data.

In 2015, a significant enhancement to the encoder-decoder architecture proposed by Ilya Sutskever was introduced with the introduction of attention mechanisms. This breakthrough emerged in the paper titled "Neural Machine Translation by Jointly Learning to Align and Translate" authored by Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio[2]. The concept of attention mechanisms revolutionized the field of neural machine translation by enabling the model to dynamically align and focus on different parts of the input sentence while generating the corresponding output. Unlike traditional encoder-decoder models, which encode the entire input sequence into a fixed-length vector representation, attention mechanisms allow the model to selectively attend to relevant information at each decoding step. This enables more accurate and contextually informed translations, particularly in cases where the input and output sequences are of varying lengths or contain complex syntactic structures. The introduction of attention mechanisms marked a significant advancement in sequence-to-sequence learning, paving the way for improved performance in various natural language processing tasks beyond machine translation.

In 2017, Vaswani et al. addressed the computational inefficiencies inherent in the Bahdanau attention mechanism by introducing a groundbreaking paper titled "Attention is All You Need"[3]. This seminal work revolutionized the field by proposing an innovative solution to handle time complexity issues and facilitate parallel processing in neural networks. The authors introduced the concept of self-attention, which enables a neural network to weigh the importance of different input tokens based on their contextual relationships within the sequence, without relying on external alignment information. Moreover, they introduced the concept of multi-head attention, which allows the model to attend to different parts of the input sequence simultaneously, enhancing its ability to capture complex dependencies and patterns. These novel attention mechanisms formed the foundation of transformer architectures (Figure 3), which have since become the state-of-the-art approach for various natural language processing tasks. By leveraging self-attention and multi-head attention, transformers enable efficient processing of sequential data while achieving superior performance and scalability compared to traditional recurrent and convolutional architectures. Thus, the publication of "Attention is All You Need" marked a significant milestone in the advancement of deep learning, paving the way for transformative developments in natural language understanding and generation.

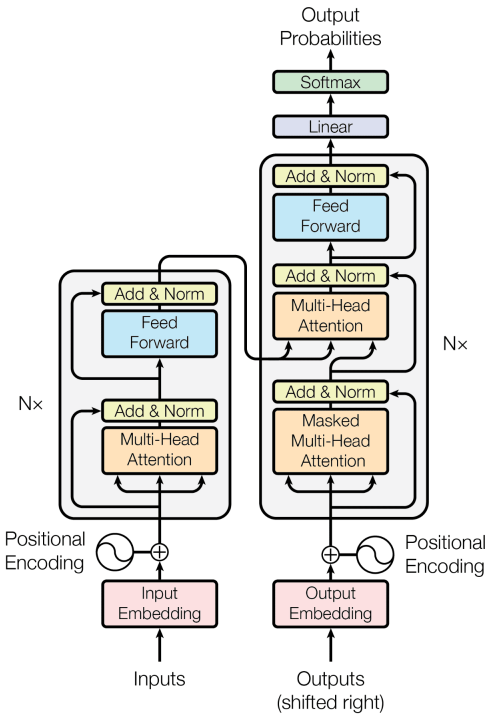


Figure 3: Transformer architecture

Until 2018, transfer learning was not widely adopted in the field of Natural Language Processing (NLP), and there were several reasons contributing to this phenomenon. One significant factor was the prevalent practice of using machine translation tasks for pre-training language models. While machine translation provided valuable training data and helped improve models' ability to understand and generate text, its applicability to a wide range of NLP tasks was limited.

However, in 2018, Jeremy Howard and colleagues[4] introduced a novel approach to pre-training language models based on language modeling tasks. This marked a departure from the traditional reliance on machine translation for pre-training. By utilizing language modeling, where the model learns to predict the next word in a sequence given previous words, Howard et al. demonstrated remarkable improvements in model performance.

This shift to language modeling for pre-training proved to be highly effective for several reasons. Firstly, language modeling tasks are inherently simpler and more generalizable compared to machine translation tasks, allowing models to capture broader linguistic patterns and nuances. Additionally, pre-training on

language modeling tasks enabled the models to learn rich contextual representations of the input text, which could then be fine-tuned for various downstream NLP tasks.

After the introduction of the ULMFiT[4] paper and the development of the transformer architecture, significant advancements were made in training very large language models.

The transformer architecture addressed many of the limitations of previous architectures, such as recurrent neural networks (RNNs) and long short-term memory networks (LSTMs), by enabling more efficient parallelization and capturing long-range dependencies in sequences. This allowed researchers to train much larger language models on vast amounts of text data.

The availability of large-scale datasets, advances in hardware infrastructure, and improvements in training algorithms also played crucial roles in the development of large language models. With access to extensive computational resources and the ability to efficiently parallelize training, researchers were able to train models with billions or even trillions of parameters.

These large language models, such as OpenAI's GPT (Generative Pre-trained Transformer)[5] series and Google's BERT (Bidirectional Encoder Representations from Transformers)[6], have demonstrated remarkable capabilities in various NLP tasks, including language understanding, text generation, and translation. They have achieved state-of-the-art performance on benchmark datasets and have become essential tools for researchers and practitioners in the field.

Furthermore, the success of large language models has spurred further research and innovation, leading to the development of even more sophisticated architectures and training techniques. Overall, the combination of transformer architecture, large-scale datasets, computational resources, and advancements in training algorithms has enabled the training of very large language models, opening up new possibilities for natural language understanding and generation.

In the following section, we present the GPT architecture utilized in the project.

# 3. GPT architecture

The architecture of GPT (Generative Pre-trained Transformer) represents a groundbreaking approach to natural language processing (NLP), built upon the Transformer model, a neural network architecture known for its effectiveness in handling sequential data. Unlike traditional recurrent neural networks (RNNs) and convolutional neural networks (CNNs), which rely on sequential processing or local context extraction, GPT leverages self-attention mechanisms to capture global dependencies and relationships within input sequences. This allows GPT to process and understand text at a more holistic level, leading to superior performance in various NLP tasks. Furthermore, GPT employs a multi-layered transformer decoder only architecture (Figure 4), where each layer consists of multiple attention heads, facilitating parallel computation and enhancing the model's capacity to capture nuanced linguistic patterns.

Now, let's delve into each component of the GPT architecture, examining its intricacies and functionalities in detail. First, let's explore the embedding layer.

Note: For the precise values of the dimensions of the matrices and neural network layers, refer to Section 4.

## 3.1. Embedding layer

The embedding layer in the GPT architecture plays a pivotal role in transforming input tokens into dense, continuous vector representations. These embeddings serve as the initial input to the model and capture the semantic and syntactic information of the input tokens.

In this project, we built a character-level model, so each token corresponds to a single character in the input text. The embedding layer maps each character to a high-dimensional vector space, capturing its intrinsic properties and relationships with other characters. This enables the model to learn complex patterns and dependencies at the character level, facilitating the generation of coherent and contextually relevant text. By leveraging character-level embeddings, GPT-based models can effectively handle out-of-vocabulary words and capture morphological variations, making them robust and adaptable to diverse linguistic phenomena.

The embedding layer consists of two main components: the **token embedding table** and the **position embedding table**. The token embedding table maps each input token to a high-dimensional embedding vector. This allows

the model to learn meaningful representations for each token in the vocabulary.

Additionally, the position embedding table assigns a unique embedding vector to each position in the input sequence. By combining token embeddings with position embeddings, the model gains the ability to capture both the content and the positional information of the input tokens. These embeddings are then fed into the transformer blocks of the model for further processing and generation of output logits.

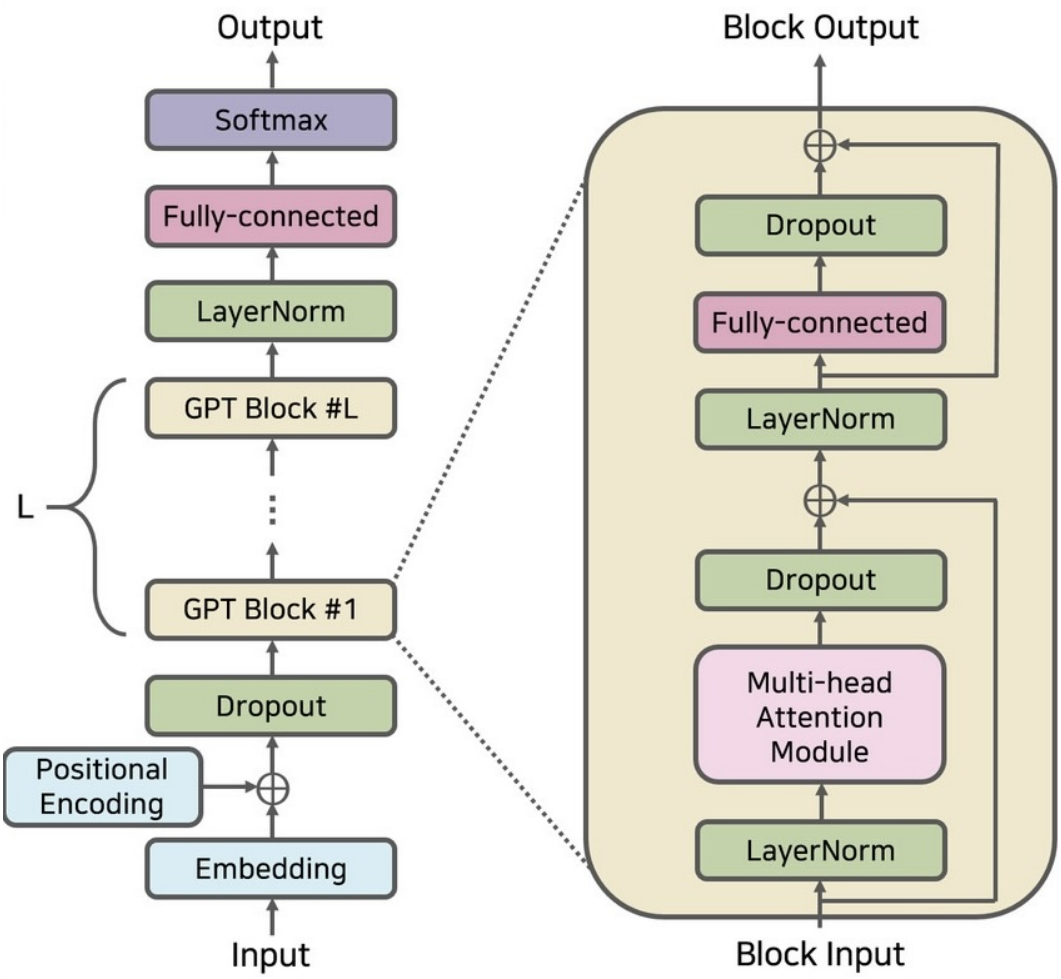


Figure 4: Generative Pre-Trained Transformer(GPT) architecture

Next, we'll plunge into the details of the GPT block architecture.

## 3.2. GPT Block

The GPT (Generative Pre-trained Transformer) block architecture (Figure 4) represents a cornerstone in the realm of natural language processing (NLP), embodying the transformative power of the transformer model in sequence modeling tasks. Each GPT block encapsulates a sequence of operations designed to capture intricate linguistic patterns and dependencies within input sequences. Comprising layers of multi-head self-attention mechanisms and feedforward neural networks, the GPT block enables the model to effectively process sequential data at various levels of abstraction. Through iterative application of these blocks, the model progressively refines its understanding of the input sequence, ultimately generating coherent and contextually relevant output.

First, we present a theoretical overview of the attention mechanism for reference, despite its absence in our model. Following this, we will delve into the concept of self-attention, which was utilized in our model.

### 3.2.1 Attention

The attention mechanism is a pivotal component in modern neural network architectures, particularly in the domain of natural language processing (NLP). At its core, attention allows models to focus on specific parts of input sequences while generating outputs, enabling more contextually informed predictions. Conceptually inspired by human attention, this mechanism assigns importance weights to different elements of the input sequence, highlighting relevant information for each output token. This is achieved through a series of computations that compare each element of the input sequence with a query vector, resulting in attention scores that indicate the relevance of each element. These attention scores are then normalized to form attention weights, which determine the contribution of each element to the final output. By dynamically adjusting these weights based on the context of the input sequence, attention mechanisms empower models to selectively attend to salient features, improving their ability to capture long-range dependencies and generate coherent outputs. From machine translation to text summarization, attention has become a cornerstone technique in NLP, driving significant advancements in model performance and enabling more nuanced and contextually rich language understanding and generation.

In the context of attention mechanisms, the concepts of query, key, and values play fundamental roles in determining how information is attended to within a sequence.

*Query:* The query vector represents the current position or token for which attention is being computed. It serves as a reference point to compare against all other elements in the sequence. The query vector is typically generated from the hidden state of the decoder in sequence-to-sequence models or from the current token embedding in self-attention mechanisms.

*Key:* The key vectors are representations of all elements in the sequence that are being attended to. Each key vector captures specific characteristics or features of its corresponding element in the input sequence. Keys are generated from the input embeddings or hidden states of the encoder in sequence-to-sequence models.

*Values:* The value vectors correspond to the actual content or information associated with each element in the sequence. They are used to compute the output of the attention mechanism and are typically identical to the key vectors. Values are generated from the same input embeddings or hidden states as the keys.

To compute attention, the query vector is compared against all key vectors using a similarity function, such as dot product or cosine similarity. This comparison yields attention scores, which reflect the relevance or importance of each element in the sequence relative to the query. These attention scores are then normalized to obtain attention weights, which determine the contribution of each value vector to the final attended output.

Next, we will explain the self-attention mechanism employed in our model.

#### 3.2.2. Self-Attention

In our GPT model, the self-attention mechanism we utilized is also referred to as scaled dot-product attention (Figure 5).

Self-attention is a mechanism that allows a neural network to weigh the importance of different words in a sequence when processing each word in the sequence. Mathematically, self-attention can be described using the following equations:

Given an input sequence of tokens $X = \{x_1, x_2, ..., x_n\}$, where $n$ is the length of the sequence, self-attention computes a set of attention weights for each token in the sequence.

*1. Query, Key, and Value matrices:*

Let $W_q$, $W_k$, and $W_v$ be weight matrices used to project the input tokens into query $Q$, key $K$, and value $V$ spaces, respectively. These weight matrices are learnable parameters of the self-attention mechanism.

$$Q = X \cdot W_q$$
$$K = X \cdot W_k$$
$$V = X \cdot W_v$$

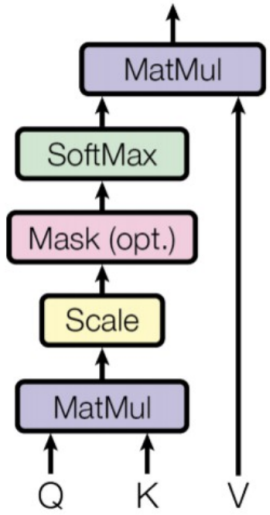


Figure 5: Scaled Dot-product Attention

*2. Attention scores:*

The attention scores $A$ are computed by taking the dot product of the query matrix $Q$ and the transpose of the key matrix $K$, scaled by the square root of the dimensionality of the key vectors ($\sqrt{d_k}$), where $d_k$ is the dimensionality of the key vectors.

$$A = \text{softmax}\left(\frac{Q \cdot K^T}{\sqrt{d_k}}\right)$$

We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

*3. Weighted sum of values:*

The attention weights $A$ are then used to compute a weighted sum of the value vectors $V$ to produce the output $O$ for each token in the sequence.

$$O = A \cdot V$$

These equations represent the fundamental operations of self-attention. By computing attention scores based on the similarity between query and key vectors and using these scores to weight the value vectors, self-attention allows neural networks to effectively capture dependencies and relationships between tokens in a sequence, facilitating more effective sequence modeling and generation.

Next, we'll provide details regarding the multi-head attention mechanism.

#### 3.2.3. Multi-Head Attention

Multi-head attention (Figure 6) is a mechanism that enhances the expressive power of self-attention mechanisms by allowing the model to attend to different parts of the input sequence simultaneously across multiple "heads". Each head operates independently, enabling the model to capture different aspects of the input sequence. Here's a description of multi-head attention with mathematically:

Let's consider a sequence of input vectors $X = \{x_1, x_2, ..., x_n\}$, where $n$ is the length of the sequence.

1. *Query, Key, and Value Matrices*:

   We project the input vectors into query $Q$, key $K$, and value $V$ spaces using learnable weight matrices $W_q \in \mathbb{R}^{d_{\text{model}} \times d_k}$, $W_k \in \mathbb{R}^{d_{\text{model}} \times d_k}$, and $W_v \in \mathbb{R}^{d_{\text{model}} \times d_v}$, respectively.

   $$\begin{aligned} Q &= X \cdot W_q \\ K &= X \cdot W_k \\ V &= X \cdot W_v \end{aligned}$$

   Here, $d_{\text{model}}$ is the dimensionality of the input vectors, $d_k$ is the dimensionality of the key vectors, and $d_v$ is the dimensionality of the value vectors.

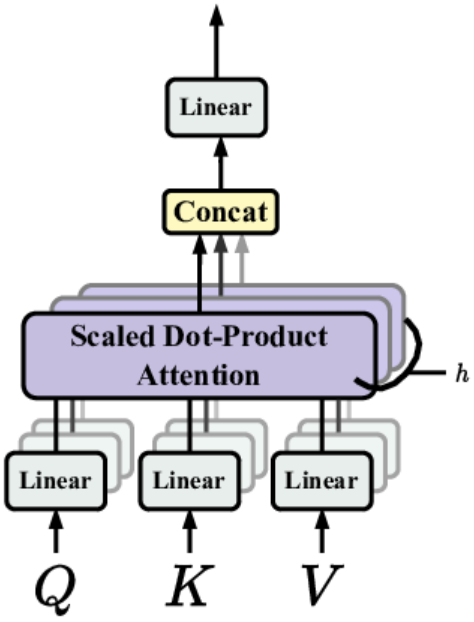


Figure 6: Multi-Head Attention

2. *Splitting into Multiple Heads*:

   We split the query, key, and value matrices into $h$ heads. For each head $i$, we obtain query $Q_i \in \mathbb{R}^{n \times \frac{d_k}{h}}$, key $K_i \in \mathbb{R}^{n \times \frac{d_k}{h}}$, and value $V_i \in \mathbb{R}^{n \times \frac{d_v}{h}}$ matrices.

$$
\begin{aligned}
Q_i &= Q \cdot W_{qi} \\
K_i &= K \cdot W_{ki} \\
V_i &= V \cdot W_{vi}
\end{aligned}
$$

3. *Scaled Dot-Product Attention for Each Head*:

   We compute the attention scores $A_i \in \mathbb{R}^{n \times n}$ for each head $i$ using the scaled dot-product attention mechanism.

$$
A_i = \text{softmax}\left(\frac{Q_i \cdot K_i^T}{\sqrt{d_k/h}}\right)
$$

4. *Weighted Sum of Values for Each Head*:

We use the attention scores $A_i$ to compute the output $O_i \in \mathbb{R}^{n \times \frac{d_v}{h}}$ for each head $i$ by taking the weighted sum of the value vectors $V_i$.

$$O_i = A_i \cdot V_i$$

5. *Concatenation and Linear Transformation*:

   Finally, we concatenate the outputs of all heads and apply a linear transformation to obtain the final multi-head attention output $O \in \mathbb{R}^{n \times d_{\text{model}}}$.

$$O = \text{Concat}(O_1, O_2, ..., O_h) \cdot W_o$$

   Here, $W_o \in \mathbb{R}^{h\frac{d_v}{h} \times d_{\text{model}}}$.

Next, we'll describe the masked multi-head attention mechanism used in our model.

#### 3.3.4. Masked Multi-Head Attention

Masked multi-head attention is an extension of the standard multi-head attention mechanism that incorporates masking to prevent attending to future tokens in the sequence during training. This is particularly useful in autoregressive models like GPT where the generation of each token depends only on previously generated tokens.

Before computing the attention scores, we apply a mask to the key matrix to prevent attending to future tokens. This mask is typically a lower triangular matrix where all elements above the diagonal are set to negative infinity, ensuring that future tokens receive zero attention.

Masked Multi-Head Attention helps in maintaining the causality constraint and improves the quality of generated sequences, especially in tasks like language modeling and text generation. Because of these reasons we used masked multi-head attention in our model.

Next, we'll provide a concise overview of the feed-forward network utilized in our model, along with the precise mathematical equation.

### 3.3. Fully connected feed-forward Networks

In addition to Masked Multi-Head Attention, each of the GPT block contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU

activation in between.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

While the linear transformations are the same across different positions, they use different parameters from layer to layer.

### 3.4. Layer Normalization

Layer normalization (LayerNorm)[9] in GPT architecture is a technique used to normalize the activations of each layer independently within the model. It operates similarly to batch normalization but normalizes the values along the feature dimension instead of the batch dimension.

In GPT architecture, layer normalization is typically applied before the self-attention mechanism and the feed-forward network within each transformer block. It helps in stabilizing the training process by reducing the internal covariate shift and accelerating convergence.

Mathematically, layer normalization is defined as follows:

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \odot \gamma + \beta$$

Where:

- $x$ is the input tensor to be normalized,
- $\mu$ is the mean of the input tensor $x$,
- $\sigma^2$ is the variance of the input tensor $x$,
- $\epsilon$ is a small constant (typically added for numerical stability),
- $\gamma$ is a learnable scale parameter,
- $\beta$ is a learnable shift parameter, and
- $\odot$ represents element-wise multiplication.

Layer normalization ensures that the activations of each layer have zero mean and unit variance, which helps in improving the stability and efficiency of the training process. It has been shown to be effective in various deep learning architectures, including transformers like GPT.

### 3.5. Droupout

Dropout is a regularization technique commonly used in neural networks to prevent overfitting[7]. In the context of GPT (Generative Pre-trained Transformer), dropout is applied to the activations of neurons in the network during training.

The idea behind dropout is to randomly drop out (i.e., set to zero) a proportion of neurons in the network during each training iteration. This prevents individual neurons from becoming overly reliant on the presence of other specific neurons, thus promoting more robust and generalizable learning.

Mathematically, dropout is implemented by multiplying the activations of neurons by a binary mask during training. The binary mask has values of 0 or 1, indicating whether each neuron is dropped out or kept, respectively. The mask is randomly generated for each training iteration according to a specified dropout probability.

Let's denote the activation of a neuron $i$ in a particular layer as $a_i$. During training, we apply dropout to $a_i$ by multiplying it by a binary mask $m_i$ as follows:

$$a_i' = a_i \times m_i$$

where $a_i'$ is the modified activation of neuron $i$. The binary mask $m_i$ is randomly generated for each training iteration according to a specified dropout probability $p$. Each element of the binary mask is independently sampled from a Bernoulli distribution with probability $1 - p$ for being set to 1 (i.e., kept) and probability $p$ for being set to 0 (i.e., dropped out).

During inference (i.e., when making predictions), dropout is not applied, and instead, the activations are scaled by $1 - p$ to ensure that the expected value of the activations remains the same as during training.

### 3.6. Residual connections

Residual connections[8], also known as skip connections, are a fundamental component of the architecture of deep neural networks, including GPT (Generative Pre-trained Transformer). They are used to mitigate the vanishing gradient problem and facilitate the training of very deep networks.

The idea behind residual connections is to add the input of a particular layer directly to the output of that layer, forming a shortcut connection. Mathematically, this can be represented as follows:

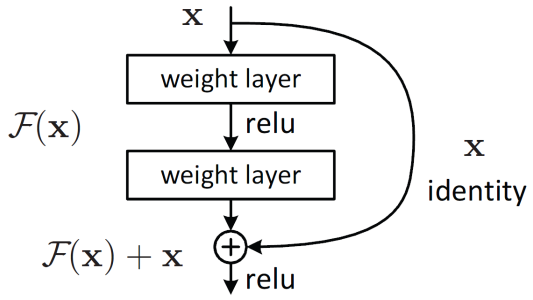


Figure 7: Residual connection

Let's denote the input to a layer as $x$ and the output of the layer (after passing through the layer's operations) as $F(x)$. The output of the layer with a residual connection is then given by:

$$\text{Output} = F(x) + x$$

This operation is performed element-wise, meaning that each element of the input tensor $x$ is added to the corresponding element of the output tensor $F(x)$ (Figure 7).

The intuition behind residual connections is that if the layer is able to learn an identity mapping (i.e., if $F(x) = x$), then the output of the layer will be equal to the input $x$, and the gradient with respect to the input will be preserved. This allows the gradient to flow directly through the shortcut connection without being significantly affected by the operations of the layer. As a result, gradient vanishing is mitigated, and the training of deep networks becomes more stable.

Next, we will describe the optimizer used to train the GPT model

### 3.7. AdamW optimizer

The AdamW optimizer, an extension of the Adam optimizer, is a popular optimization algorithm used in training neural networks. It combines techniques from both adaptive moment estimation (Adam) and weight decay (commonly denoted as L2 regularization).

Mathematically, the AdamW optimizer updates the parameters (weights) of the neural network according to the following steps:

1. **Initialization**:

- Initialize the parameters $\theta$ of the neural network and the moments $m$ and $v$ to zero. Let $t = 0$ denote the initial time step.

2. **Compute Gradient**:
   - Compute the gradients of the loss function $L(\theta)$ with respect to the parameters $\nabla_\theta L(\theta)$ using backpropagation.
3. **Update Moment Estimations**:
   - Update the first moment estimate $m$ and the second moment estimate $v$ for each parameter:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla_\theta L(\theta)$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (\nabla_\theta L(\theta))^2$$

   Here, $\beta_1$ and $\beta_2$ are the decay rates for the first and second moment estimates, typically close to 1 (e.g., 0.9 and 0.999, respectively).
4. **Bias Correction**:
   - Since the moment estimates are biased towards zero at the beginning, we apply bias correction to them:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

5. **Weight Update**:
   - Update the parameters $\theta$ using the bias-corrected moment estimates:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t$$

   Here, $\eta$ is the learning rate and $\epsilon$ is a small constant (e.g., $10^{-8}$) added to the denominator for numerical stability.
6. **Weight Decay (L2 Regularization)**:
   - Finally, perform weight decay (L2 regularization) by updating the parameters using:

$$\theta_{t+1} = \theta_{t+1} - \lambda \cdot \eta \cdot \theta_t$$

   Here, $\lambda$ is the weight decay coefficient.

The AdamW optimizer incorporates weight decay directly into the parameter updates, unlike the original Adam optimizer. This modification has been shown to improve the generalization performance of neural networks by effectively controlling the growth of the weights during training.

Having covered all the necessary prerequisites for implementing the GPT model, the subsequent section will focus on the implementation of our GPT-based language model in Python. Additionally, we will provide a detailed discussion of the output generated by the model in parallel with the implementation process.

# 4. Implementation of GPT-Based Language Model and Results

In this section, we will guide the reader through the Jupyter notebook detailing the implementation of the GPT-based language model.

**OBJECTIVE:**
To train a Generative Pre-trained Transformer (GPT) based language model on the Spotify Million Songs dataset and then utilize the trained model to generate new songs.

We start our Python code by importing the necessary libraries.

## 4.1. Importing the required python libraries

```
import torch
import torch.nn as nn
from torch.nn import functional as F
torch.manual_seed(1337)
```

This Python code imports the PyTorch library and sets a manual seed for reproducibility. Let's break down what each line does:

- `import torch`: This imports the PyTorch library, which is a popular deep learning framework in Python.
- `import torch.nn as nn`: This imports the neural network module `nn` from PyTorch, which contains useful classes for building neural networks.
- `from torch.nn import functional as F`: This imports the functional interface of PyTorch's neural network module `nn` and renames it as `F`. The functional interface contains functions like activation functions, loss functions, and other utilities.

- `torch.manual_seed(1337)`: This sets the random seed of PyTorch's random number generator to 1337. Setting a manual seed ensures that the random numbers generated by PyTorch are reproducible across runs, which is useful for debugging and testing purposes.

Next, we will proceed to load the dataset into our program.

### 4.2. Dataset

The dataset utilized is known as the Spotify Million Songs Dataset. It was obtained from the following URL: `https://www.kaggle.com/datasets/notshrirang/spotify-million-song-dataset`.

The Spotify Million Songs Dataset is a large-scale collection of audio features and metadata for a million songs available on the Spotify platform. It includes various attributes for each song, such as artist name, track name, release year, tempo, danceability, energy, key, and more. This dataset is valuable for research and analysis in music recommendation systems, genre classification, audio processing, and machine learning applications. Researchers and data scientists can use it to train models for tasks like song recommendation, music genre classification, mood detection, and more. The dataset provides a rich resource for exploring the characteristics of music and building intelligent systems that can understand and interact with music content effectively.

The dataset is downloaded as a CSV (Comma-Separated Values) file, and its size is approximately 74.9 megabytes, occupying memory when loaded into a program for analysis or processing.

Next, we provide the code for loading the dataset into our program

```
import pandas as pd

df = pd.read_csv('/storage/sridinesh/MTP/spotify_millsongdata.csv',encoding='
    utf-8')
text = df['text'].str.cat()
```

Here we are reading a CSV file named "spotify_millsongdata.csv" using the Pandas library. The file is assumed to be located at the path "/storage/sridinesh/MTP/" on the filesystem. The `encoding='utf-8'` parameter specifies the character encoding to be used for reading the file.

After reading the CSV file into a DataFrame `df`, it concatenates all the values in the 'text' column into a single string using the `str.cat()` method. This

results in a single string variable named `text` that contains the concatenated text from all the rows in the 'text' column of the DataFrame.

Now, the variable `text` contains all the data.

```
print("length of dataset in characters: ", len(text))
```

```
Output: length of dataset in characters: 70368523
```

Next, let's look at the first 1000 characters of our dataset.

```
# let's look at the first 1000 characters
print(text[:1000])
```

```
Output:
Look at her face, it's a wonderful face
And it means something special to me
Look at the way that she smiles when she sees me
How lucky can one fellow be?

She's just my kind of girl, she makes me feel fine
Who could ever believe that she could be mine?
She's just my kind of girl, without her I'm blue
And if she ever leaves me what could I do, what could I do?

And when we go for a walk in the park
And she holds me and squeezes my hand
We'll go on walking for hours and talking
About all the things that we plan

She's just my kind of girl, she makes me feel fine
Who could ever believe that she could be mine?
She's just my kind of girl, without her I'm blue
And if she ever leaves me what could I do, what could I do?

Take it easy with me, please
Touch me gently like a summer evening breeze
Take your time, make it slow
Andante, Andante
Just let the feeling grow
```

```
Make your fingers soft and light
Let your body be t
```

The below code block above outputs all the unique characters present in our dataset, which we have stored in the variable `text`. The variable `vocab_size` represents the number of unique characters in our dataset.

```python
# here are all the unique characters that occur in this text
chars = sorted(list(set(text)))
vocab_size = len(chars)
print(''.join(chars))
print(vocab_size)
```

Output:

```
    !"'(),-.0123456789:?ABCDEFGHIJKLMNOPQRSTUVWXYZ[]abcdefghijklmnopqrstuvwxyz
77
```

The below code sets up encoding and decoding functions for converting strings to lists of integers and vice versa, using the provided character set. These functions can be useful for tasks like text encoding and decoding in natural language processing or sequence modeling.

```python
# create a mapping from characters to integers
stoi = {ch:i for i,ch in enumerate(chars)}
itos = {i:ch for i,ch in enumerate(chars)}
encode = lambda s: [stoi[c] for c in s]  # encoder: take a string, output a
    list of integers
decode = lambda l: ''.join([itos[i] for i in l])  # decoder: take a list of
    integers, output a string
```

This code is creating a character-to-integer mapping and its reverse for a given set of characters. Here's what each part of the code does:

1. `stoi = { ch:i for i,ch in enumerate(chars) }`: This line creates a dictionary `stoi` where each character in the list `chars` is mapped to its corresponding index in the list `chars`. It effectively creates a mapping from characters to integers.
2. `itos = { i:ch for i,ch in enumerate(chars) }`: This line creates a dictionary `itos` where each index in the list `chars` is mapped to its corresponding character in the list `chars`. It creates the reverse mapping of `stoi`, mapping integers back to characters.
3. `encode = lambda s: [stoi[c] for c in s]`: This line defines a lambda function called `encode`. This function takes a string `s` as input and returns a list of integers. For each character `c` in the input string `s`, it

retrieves the corresponding integer index from the `stoi` dictionary and appends it to the list.

4. `decode = lambda l: ''.join([itos[i] for i in l])`: This line defines a lambda function called `decode`. This function takes a list of integers `l` as input and returns a string. For each integer `i` in the input list `l`, it retrieves the corresponding character from the `itos` dictionary and joins them together to form a string.

For instance lets encode and decode "hii there":

```
print(encode("hii there"))
print(decode(encode("hii there")))
```

```
Output: [58, 59, 59, 2, 70, 58, 55, 68, 55]
         hii there
```

So, after encoding "hii there" becomes [58, 59, 59, 2, 70, 58, 55, 68, 55]. Next we encode the entire text dataset.

```
# let's now encode the entire text dataset and store it into a torch.Tensor
data = torch.tensor(encode(text), dtype=torch.long)
print(data.shape, data.dtype)
print(data[:1000]) # the 1000 characters we looked at earier will to the GPT
    look like this
```

```
Output: torch.Size([70368523]) torch.int64
tensor([34, 65, 65, 61,  2, 51, 70,  2, 58, 55, 68,  2, 56, 51, 53, 55,  8,  2,
        59, 70,  5, 69,  2, 51,  2, 73, 65, 64, 54, 55, 68, 56, 71, 62,  2, 56,
        51, 53, 55,  2,  2,  1,  0, 23, 64, 54,  2, 59, 70,  2, 63, 55, 51, 64,
        69,  2, 69, 65, 63, 55, 70, 58, 59, 64, 57,  2, 69, 66, 55, 53, 59, 51,
        62,  2, 70, 65,  2, 63, 55,  2,  2,  1,  0, 34, 65, 65, 61,  2, 51, 70,
         2, 70, 58, 55,  2, 73, 51, 75,  2, 70, 58, 51, 70,  2, 69, 58, 55,  2,
        69, 63, 59, 62, 55, 69,  2, 73, 58, 55, 64,  2, 69, 58, 55,  2, 69, 55,
        55, 69,  2, 63, 55,  2,  2,  1,  0, 30, 65, 73,  2, 62, 71, 53, 61, 75,
         2, 53, 51, 64,  2, 65, 64, 55,  2, 56, 55, 62, 62, 65, 73,  2, 52, 55,
        22,  2,  2,  1,  0,  2,  2,  1,  0, 41, 58, 55,  5, 69,  2, 60, 71, 69,
        70,  2, 63, 75,  2, 61, 59, 64, 54,  2, 65, 56,  2, 57, 59, 68, 62,  8,
         2, 69, 58, 55,  2, 63, 51, 61, 55, 69,  2, 63, 55,  2, 56, 55, 55, 62,
         2, 56, 59, 64, 55,  2,  2,  1,  0, 45, 58, 65,  2, 53, 65, 71, 62, 54,
         2, 55, 72, 55, 68,  2, 52, 55, 62, 59, 55, 72, 55,  2, 70, 58, 51, 70,
         2, 69, 58, 55,  2, 53, 65, 71, 62, 54,  2, 52, 55,  2, 63, 59, 64, 55,
```

```
22,  2,  2,  1,  0, 41, 58, 55,  5, 69,  2, 60, 71, 69, 70,  2, 63, 75,
 2, 61, 59, 64, 54,  2, 65, 56,  2, 57, 59, 68, 62,  8,  2, 73, 59, 70,
58, 65, 71, 70,  2, 58, 55, 68,  2, 31,  5, 63,  2, 52, 62, 71, 55,  2,
 2,  1,  0, 23, 64, 54,  2, 59, 56,  2, 69, 58, 55,  2, 55, 72, 55, 68,
 2, 62, 55, 51, 72, 55, 69,  2, 63, 55,  2, 73, 58, 51, 70,  2, 53, 65,
71, 62, 54,  2, 31,  2, 54, 65,  8,  2, 73, 58, 51, 70,  2, 53, 65, 71,
62, 54,  2, 31,  2, 54, 65, 22,  2,  2,  1,  0,  2,  2,  1,  0, 23, 64,
54,  2, 73, 58, 55, 64,  2, 73, 55,  2, 57, 65,  2, 56, 65, 68,  2, 51,
 2, 73, 51, 62, 61,  2, 59, 64,  2, 70, 58, 55,  2, 66, 51, 68, 61,  2,
 2,  1,  0, 23, 64, 54,  2, 69, 58, 55,  2, 58, 65, 62, 54, 69,  2, 63,
55,  2, 51, 64, 54,  2, 69, 67, 71, 55, 55, 76, 55, 69,  2, 63, 75,  2,
58, 51, 64, 54,  2,  2,  1,  0, 45, 55,  5, 62, 62,  2, 57, 65,  2, 65,
64,  2, 73, 51, 62, 61, 59, 64, 57,  2, 56, 65, 68,  2, 58, 65, 71, 68,
69,  2, 51, 64, 54,  2, 70, 51, 62, 61, 59, 64, 57,  2,  2,  1,  0, 23,
52, 65, 71, 70,  2, 51, 62, 62,  2, 70, 58, 55,  2, 70, 58, 59, 64, 57,
69,  2, 70, 58, 51, 70,  2, 73, 55,  2, 66, 62, 51, 64,  2,  2,  1,  0,
 2,  2,  1,  0, 41, 58, 55,  5, 69,  2, 60, 71, 69, 70,  2, 63, 75,  2,
61, 59, 64, 54,  2, 65, 56,  2, 57, 59, 68, 62,  8,  2, 69, 58, 55,  2,
63, 51, 61, 55, 69,  2, 63, 55,  2, 56, 55, 55, 62,  2, 56, 59, 64, 55,
 2,  2,  1,  0, 45, 58, 65,  2, 53, 65, 71, 62, 54,  2, 55, 72, 55, 68,
 2, 52, 55, 62, 59, 55, 72, 55,  2, 70, 58, 51, 70,  2, 69, 58, 55,  2,
53, 65, 71, 62, 54,  2, 52, 55,  2, 63, 59, 64, 55, 22,  2,  2,  1,  0,
41, 58, 55,  5, 69,  2, 60, 71, 69, 70,  2, 63, 75,  2, 61, 59, 64, 54,
 2, 65, 56,  2, 57, 59, 68, 62,  8,  2, 73, 59, 70, 58, 65, 71, 70,  2,
58, 55, 68,  2, 31,  5, 63,  2, 52, 62, 71, 55,  2,  2,  1,  0, 23, 64,
54,  2, 59, 56,  2, 69, 58, 55,  2, 55, 72, 55, 68,  2, 62, 55, 51, 72,
55, 69,  2, 63, 55,  2, 73, 58, 51, 70,  2, 53, 65, 71, 62, 54,  2, 31,
 2, 54, 65,  8,  2, 73, 58, 51, 70,  2, 53, 65, 71, 62, 54,  2, 31,  2,
54, 65, 22,  1,  0,  1,  0, 42, 51, 61, 55,  2, 59, 70,  2, 55, 51, 69,
75,  2, 73, 59, 70, 58,  2, 63, 55,  8,  2, 66, 62, 55, 51, 69, 55,  2,
 2,  1,  0, 42, 65, 71, 53, 58,  2, 63, 55,  2, 57, 55, 64, 70, 62, 75,
 2, 62, 59, 61, 55,  2, 51,  2, 69, 71, 63, 63, 55, 68,  2, 55, 72, 55,
64, 59, 64, 57,  2, 52, 68, 55, 55, 76, 55,  2,  2,  1,  0, 42, 51, 61,
55,  2, 75, 65, 71, 68,  2, 70, 59, 63, 55,  8,  2, 63, 51, 61, 55,  2,
59, 70,  2, 69, 62, 65, 73,  2,  2,  1,  0, 23, 64, 54, 51, 64, 70, 55,
 8,  2, 23, 64, 54, 51, 64, 70, 55,  2,  2,  1,  0, 32, 71, 69, 70,  2,
62, 55, 70,  2, 70, 58, 55,  2, 56, 55, 55, 62, 59, 64, 57,  2, 57, 68,
65, 73,  2,  2,  1,  0,  2,  2,  1,  0, 35, 51, 61, 55,  2, 75, 65, 71,
68,  2, 56, 59, 64, 57, 55, 68, 69,  2, 69, 65, 56, 70,  2, 51, 64, 54,
 2, 62, 59, 57, 58, 70,  2,  2,  1,  0, 34, 55, 70,  2, 75, 65, 71, 68,
 2, 52, 65, 54, 75,  2, 52, 55,  2, 70])
```

In the above code block, the `torch.tensor()` function is used to convert the encoded text data into a tensor of type `torch.long`. Then, the shape and data type of the tensor are printed, followed by displaying the first 1000 characters of the encoded text data.

The initial portion of our dataset, consisting of the first thousand characters written in English, will appear differently to our GPT model once it undergoes encoding with the function we have defined. This process involves transforming the English text into a sequence of integers using the encode function. Consequently, the GPT model will interpret this sequence of integers, representing the characters, rather than directly processing the original English text. This encoding enables the GPT model to work with numerical data, allowing it to analyze and generate responses based on the underlying patterns and structures within the text data.

The encoded text data is stored in the variable `data`.

Now let's split the data into training and validation sets.

```
# Let's now split up the data into train and validation sets
n = int(0.9*len(data)) # first 90% will be train, rest val
train_data = data[:n]
val_data = data[n:]

print(train_data.shape)
print(val_data.shape)
```

```
Output: torch.Size([63331670])
        torch.Size([7036853])
```

The above code splits the encoded text data into training and validation sets. Here's a breakdown of what each part of the code does:

1. `n = int(0.9*len(data))`: This line calculates the index `n`, which represents 90% of the length of the encoded data. It determines the point at which the dataset will be split into the training and validation sets.
2. `train_data = data[:n]`: This line creates the training set by slicing the encoded data from the beginning up to index `n`. It includes the first 90% of the data.
3. `val_data = data[n:]`: This line creates the validation set by slicing the encoded data from index `n` onwards. It includes the remaining 10% of the data.

4. `print(train_data.shape)`: This line prints the shape of the training dataset, indicating the number of elements in each dimension.

5. `print(val_data.shape)`: This line prints the shape of the validation dataset, indicating the number of elements in each dimension.

The output indicates that the length of the training dataset (`train_data`) is 63,331,670, and the length of the validation dataset (`val_data`) is 7,036,853.

Next, we introduce the hyper parameters of the model.

### 4.3. Hyper parameters

```
# hyperparameters
batch_size = 64 # how many independent sequences will we process in parallel?
block_size = 256 # what is the maximum context length for predictions?
max_iters = 10000
eval_interval = 500
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'cpu'
eval_iters = 200
n_embd = 384
n_head = 6
n_layer = 6
dropout = 0.2
```

The above code snippet defines various hyperparameters for training a model. Let's break down what each hyperparameter means:

1. `batch_size`: This parameter determines how many independent sequences will be processed in parallel during training. It affects the number of samples processed in each iteration of training.

2. `block_size`: It specifies the maximum context length for predictions. During training, the model will generate predictions based on context windows of this size.

3. `max_iters`: This parameter sets the maximum number of iterations (or training steps) that the training process will run for.

4. `eval_interval`: It determines the frequency at which evaluation of the model's performance will be conducted during training. In this case, evaluation will occur every 500 iterations.

5. `learning_rate`: This parameter sets the learning rate for the optimizer, which controls the size of the steps taken during gradient descent. It affects the rate at which the model parameters are updated during training.

6. `device`: This parameter specifies whether to use a GPU ('cuda') or CPU ('cpu') for training. It checks if CUDA (NVIDIA's parallel computing platform) is available and selects the appropriate device.

7. `eval_iters`: It specifies the number of iterations used for evaluation during training. This parameter determines how many iterations are used to evaluate the model's performance at each evaluation interval.

8. `n_embd`: This parameter defines the dimensionality of the embedding layer, which transforms input tokens into dense vectors. It affects the richness of the learned representations.

9. `n_head`: It determines the number of attention heads in the multi-head attention mechanism. Each attention head allows the model to focus on different parts of the input sequence simultaneously.

10. `n_layer`: This parameter specifies the number of layers in the model. It determines the depth of the neural network architecture, which affects its capacity to capture complex patterns in the data.

11. `dropout`: This parameter sets the dropout probability, which controls the rate at which randomly selected neurons are ignored during training. Dropout helps prevent overfitting by regularizing the model.

These hyperparameters play crucial roles in determining the behavior and performance of the model during training. Adjusting these parameters can affect the model's learning dynamics, convergence speed, and generalization ability.

Next, we provide the function to generate data batches for training and validating the model.

### 4.4. Function to generate data batches

```
# data loading
def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y
```

This code snippet defines a function `get_batch(split)` responsible for generating batches of input-output pairs for training or validation purposes. Here's an explanation of what the function does:

1. `data = train_data if split == 'train' else val_data`: Depending on the value of the `split` parameter ('train' or 'val'), the function selects either the training dataset (`train_data`) or the validation dataset (`val_data`).
2. `ix = torch.randint(len(data) - block_size, (batch_size,))`: This line generates random indices (`ix`) within the range of the data, ensuring that the selected indices leave enough room for the block size (`block_size`) window.
3. `x = torch.stack([data[i:i+block_size] for i in ix])`: For each index in `ix`, a sequence of input data (`x`) of length `block_size` is extracted from the dataset and stacked into a tensor. This forms the input batch.
4. `y = torch.stack([data[i+1:i+block_size+1] for i in ix])`: Similarly, for each index in `ix`, a sequence of target data (`y`) shifted by one position from the input sequence is extracted and stacked into a tensor. This forms the target batch.
5. `x, y = x.to(device), y.to(device)`: Both the input and target batches are transferred to the computational device specified by the `device` variable (usually GPU if available, otherwise CPU).
6. Finally, the function returns the input (`x`) and target (`y`) batches.

This function facilitates the loading of batches of input-output pairs from the dataset for training or validation, enabling the model to learn and evaluate its performance iteratively during training.

Next, we give you the function to evaluate the loss during training and validation.

### 4.5. Function to evaluate loss

```
@torch.no_grad()
def estimate_loss():
    out = {}
    model.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)           # we are trying to compute the
    loss for eval_iters number of batches.
            logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model.train()
    return out
```

The above code defines a function `estimate_loss()` that computes the average loss for both the training and validation sets over a specified number of iterations. Here's an explanation of what each part of the function does:

1. `@torch.no_grad()`: This is a decorator indicating that the subsequent function should be executed without computing gradients. It's commonly used during evaluation to save memory and computation.
2. `out = {}`: Initializes an empty dictionary to store the computed losses for both the training and validation sets.
3. `model.eval()`: Puts the model in evaluation mode. This affects the behavior of certain layers like dropout and batch normalization, ensuring they operate differently during evaluation compared to training.
4. `for split in ['train', 'val']`: Iterates over the training and validation sets.
5. `losses = torch.zeros(eval_iters)`: Initializes a tensor to store the losses computed over multiple iterations for each split.
6. `for k in range(eval_iters)`: Iterates over the specified number of evaluation iterations.
7. `X, Y = get_batch(split)`: Calls the `get_batch()` function to obtain a batch of input-output pairs for the current split.
8. `logits, loss = model(X, Y)`: Forward pass through the model with the input `X` and target `Y` to obtain the predicted logits and compute the loss.
9. `losses[k] = loss.item()`: Stores the computed loss for the current iteration in the `losses` tensor.
10. `out[split] = losses.mean()`: Computes the mean loss over all iterations for the current split and stores it in the `out` dictionary.
11. `model.train()`: Puts the model back in training mode to resume training. This reverts the effects of `model.eval()`.
12. `return out`: Returns the dictionary containing the average losses for both the training and validation sets.

Overall, this function provides a way to estimate the loss of the model on both training and validation data without computing gradients or updating model

parameters, which is useful for monitoring the model's performance during training.

Next, we define a python class for single head of self-attention, this will be used in multi-head attention.

### 4.6. Python class for single head of self-attention

```
class Head(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        self.register_buffer('tril', torch.tril(torch.ones(block_size,
    block_size)))

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B,T,C = x.shape
        k = self.key(x)   # (B,T,C)
        q = self.query(x) # (B,T,C)
        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * C**-0.5 # (B, T, C) @ (B, C, T) -> (B,
     T, T)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T,
     T)
        wei = F.softmax(wei, dim=-1) # (B, T, T)
        wei = self.dropout(wei)
        # perform the weighted aggregation of the values
        v = self.value(x) # (B,T,C)
        out = wei @ v # (B, T, T) @ (B, T, C) -> (B, T, C)
        return out
```

This code defines a class `Head` which represents one head of the self-attention mechanism. Here's an explanation of what each part of the class does:

1. `def __init__(self, head_size):`: This is the constructor method for the `Head` class. It initializes the parameters and layers of the self-attention head.

2. `self.key = nn.Linear(n_embd, head_size, bias=False)`: This line initializes a linear transformation layer (`nn.Linear`) for computing the query vectors. The layer transforms input vectors of size `n_embd` to output vectors of size `head_size`. The `bias=False` argument indicates that no bias term is added to the transformation.

3. `self.query = nn.Linear(n_embd, head_size, bias=False)`: Similar

to the previous line, this initializes a linear transformation layer for computing the query vectors.

4. `self.value = nn.Linear(n_embd, head_size, bias=False)`: This line initializes a linear transformation layer for computing the value vectors.
5. `self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))`: This line creates a lower triangular matrix (`tril`) of size `block_size x block_size` filled with ones using `torch.ones()`. Then, it registers the matrix as a buffer using `register_buffer()`. Buffers are persistent and will be saved alongside parameters, but not updated by gradients during training.
6. `self.dropout = nn.Dropout(dropout)`: This initializes a dropout layer with dropout probability specified by the `dropout` parameter.
7. `def forward(self, x)`:: This method defines the forward pass of the self-attention head.
8. `B,T,C = x.shape`: Here, `x` is the input tensor representing a batch of sequences. `B` is the batch size, `T` is the sequence length, and `C` is the number of features in each element of the sequence.
9. `k = self.key(x)`: This line computes the query vectors by applying the linear transformation defined by `self.key` to the input `x`.
10. `q = self.query(x)`: Similarly, this line computes the query vectors by applying the linear transformation defined by `self.query` to the input `x`.
11. `wei = q @ k.transpose(-2,-1) * C**-0.5`: This line computes the attention scores ("affinities") by performing a dot product between the query and key vectors, followed by scaling by the square root of the feature dimension `C`.
12. `wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))`: Here, the lower triangular part of the attention scores matrix is masked to prevent attending to future tokens. The elements in the lower triangular part are set to negative infinity, which ensures they are effectively ignored during softmax computation.
13. `wei = F.softmax(wei, dim=-1)`: This line applies softmax activation to normalize the attention scores along the last dimension, ensuring they sum up to 1 and represent valid probabilities.
14. `wei = self.dropout(wei)`: Dropout is applied to the attention scores to prevent overfitting and encourage robustness.

15. `v = self.value(x)`: This line computes the value vectors by applying the linear transformation defined by `self.value` to the input `x`.

16. `out = wei @ v`: Finally, the weighted aggregation of the values is performed by computing the dot product between the attention scores (`wei`) and the value vectors (`v`). This produces the output tensor representing the attended values for each position in the input sequence.

Overall, this class encapsulates the functionality of one head of the self-attention mechanism, which computes the attention scores and performs the weighted aggregation of values based on these scores. This is a fundamental component of GPT models for sequence processing tasks.

Next, we define masked multi-head attention, which is a key component of GPT block.

### 4.7. Python class for Multi-Head Attention

```
class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)
    ])
        self.proj = nn.Linear(n_embd, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out
```

This code defines a class `MultiHeadAttention` representing multiple heads of the self-attention mechanism operating in parallel. Here's an explanation of what each part of the class does:

1. `def __init__(self, num_heads, head_size):`: This is the constructor method for the `MultiHeadAttention` class. It initializes the parameters and layers of the multi-head self-attention mechanism.

2. `self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])`: This line creates a list of `num_heads` instances of the `Head` class, each representing one head of the self-attention mechanism. These heads operate in parallel and are stored in a `ModuleList`, which ensures that their parameters are properly registered and tracked by PyTorch.

3. `self.proj = nn.Linear(n_embd, n_embd)`: This line initializes a linear transformation layer (`nn.Linear`) for projecting the concatenated outputs of the attention heads back to the original feature dimension `n_embd`.
4. `self.dropout = nn.Dropout(dropout)`: This initializes a dropout layer with dropout probability specified by the `dropout` parameter.
5. `def forward(self, x)`:: This method defines the forward pass of the multi-head self-attention mechanism.
6. `out = torch.cat([h(x) for h in self.heads], dim=-1)`: This line applies each head in the `heads` list to the input tensor `x` and concatenates their outputs along the last dimension (`dim=-1`). This results in a tensor where each head's output is stacked together.
7. `out = self.dropout(self.proj(out))`: The concatenated outputs are passed through a linear projection layer (`self.proj`) followed by dropout regularization.
8. `return out`: Finally, the processed output tensor is returned.

Overall, this class encapsulates the functionality of multiple heads of the self-attention mechanism operating in parallel, followed by a linear projection and dropout regularization. This is a key component of GPT models for capturing complex dependencies in sequential data.

Next, we define python class for the feed-forward neural networks used in the GPT model.

### 4.8. Python class for fully connected feed-forward network

```
class FeedFoward(nn.Module):
    """ a simple linear layer followed by a non-linearity """

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )

    def forward(self, x):
        return self.net(x)
```

This code defines a class `FeedForward` representing a simple feedforward neural network layer followed by a non-linearity. Here's what each part of the class does:

1. `def __init__(self, n_embd)`:: This is the constructor method for the `FeedForward` class. It initializes the parameters and layers of the feedforward network.
2. `self.net = nn.Sequential(...)`: This line initializes a sequence of neural network layers using `nn.Sequential`, which allows defining a sequence of layers to be applied sequentially.
3. `nn.Linear(n_embd, 4 * n_embd),`: This adds a linear transformation layer (`nn.Linear`) to the sequence, mapping the input dimension `n_embd` to 4 times the input dimension. This increases the dimensionality of the input space.
4. `nn.ReLU(),`: This adds a rectified linear unit (ReLU) activation function to the sequence, introducing non-linearity to the network.
5. `nn.Linear(4 * n_embd, n_embd),`: This adds another linear transformation layer to the sequence, mapping the 4 times expanded input dimension back to the original input dimension. This reduces the dimensionality back to the original space.
6. `nn.Dropout(dropout),`: This adds a dropout layer to the sequence with dropout probability specified by the `dropout` parameter.
7. `def forward(self, x)`:: This method defines the forward pass of the feedforward network.
8. `return self.net(x)`: This line applies the sequential layers defined in `self.net` to the input tensor `x`, producing the output tensor.

Overall, this class encapsulates the functionality of a simple feedforward neural network layer followed by a non-linearity and dropout regularization. This type of layer is commonly used in deep learning models for learning complex mappings between input and output spaces.

Next, we'll now present the implementation of a single GPT (Generative Pretrained Transformer) block utilizing the multi-head attention mechanism and the feed-forward network that we previously defined.

### 4.9. Python class for single GPT block

```
class Block(nn.Module):
    """ Transformer block: communication followed by computation """

    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd like
```

```
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x
```

This code defines a class `Block` representing a single transformer block. Transformer blocks are fundamental building blocks of transformer-based architectures for sequence processing tasks like natural language understanding and generation. Here's an explanation of what each part of the class does:

1. `def __init__(self, n_embd, n_head)`:: This is the constructor method for the `Block` class. It initializes the parameters and layers of the transformer block.

2. `head_size = n_embd // n_head`: This line calculates the size of each attention head based on the total embedding dimension (`n_embd`) and the number of attention heads (`n_head`).

3. `self.sa = MultiHeadAttention(n_head, head_size)`: This line initializes a multi-head self-attention module (`MultiHeadAttention`) with the specified number of heads (`n_head`) and head size. This module captures the dependencies between different elements in the input sequence.

4. `self.ffwd = FeedForward(n_embd)`: This line initializes a feedforward neural network module (`FeedForward`) with the specified embedding dimension (`n_embd`). This module processes each position in the input sequence independently.

5. `self.ln1 = nn.LayerNorm(n_embd)`: This line initializes a layer normalization module (`nn.LayerNorm`) to normalize the input embeddings before applying the self-attention mechanism.

6. `self.ln2 = nn.LayerNorm(n_embd)`: Similarly, this line initializes another layer normalization module to normalize the output of the feedforward neural network before adding it back to the input.

7. `def forward(self, x)`:: This method defines the forward pass of the transformer block.

8. `x = x + self.sa(self.ln1(x))`: This line applies layer normalization to the input `x`, passes it through the multi-head self-attention module

(`self.sa`), and adds the output to the original input `x`. This step captures dependencies between different positions in the input sequence.

9. `x = x + self.ffwd(self.ln2(x))`: Similarly, this line applies layer normalization to the intermediate result, passes it through the feedforward neural network module (`self.ffwd`), and adds the output to the intermediate result. This step processes each position in the input sequence independently.

10. `return x`: Finally, the processed output tensor is returned.

Overall, this class encapsulates the functionality of a single transformer block, which consists of a multi-head self-attention mechanism followed by a feedforward neural network layer, with layer normalization applied before and after each sub-module. These blocks are stacked to form the encoder and decoder of a transformer-based model.

Next, we will proceed to implement our final GPT model, incorporating all the components and definitions we have established thus far.

### 4.10. Python class for GPT model

```
class GPT(nn.Module):

    def __init__(self):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd) # final layer norm
        self.lm_head = nn.Linear(n_embd, vocab_size)

    def forward(self, idx, targets=None):
        B, T = idx.shape

        # idx and targets are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)
        x = self.blocks(x) # (B,T,C)
        x = self.ln_f(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
```

```
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx
```

This code defines a simple GPT (Generative Pre-trained Transformer) model. GPT is a type of transformer-based model used for various natural language processing tasks, including text generation. Below is an explanation of each part of the model:

1. **Initialization** (`def __init__(self):`):
    - The constructor initializes the parameters and layers of the GPT model.
    - It defines an embedding layer (`self.token_embedding_table`) to map input tokens to continuous vector representations.
    - It also defines another embedding layer (`self.position_embedding_table`) to incorporate positional information into the embeddings.
    - The GPT model consists of a stack of transformer blocks (`self.blocks`), with each block composed of self-attention and feedforward layers.
    - After the transformer blocks, layer normalization (`self.ln_f`) is applied to the output embeddings.
    - Finally, a linear layer (`self.lm_head`) maps the embeddings to logits over the vocabulary size.
2. **Forward Pass** (`def forward(self, idx, targets=None):`):
    - The forward method takes input token indices (`idx`) and optional target token indices (`targets`) for training.
    - It first retrieves token embeddings (`tok_emb`) and positional embeddings (`pos_emb`) for the input tokens and adds them together.

- The combined embeddings are passed through the transformer blocks (`self.blocks`) to capture dependencies between tokens.
- Layer normalization is applied to the output embeddings (`x`), followed by linear transformation to obtain logits (`logits`) over the vocabulary.
- If target indices are provided, the method calculates the cross-entropy loss between the predicted logits and the target indices.

3. **Text Generation** (`def generate(self, idx, max_new_tokens):`):
   - The generate method takes a sequence of token indices (`idx`) and the maximum number of new tokens to generate (`max_new_tokens`).
   - It iteratively generates new tokens based on the input sequence.
   - At each step, the model predicts the next token by sampling from the probability distribution over the vocabulary given by the softmax of the logits.
   - The sampled token indices are concatenated to the input sequence for the next iteration.
   - This process continues until the desired number of new tokens is generated.

Overall, this GPT model leverages transformer architecture to learn contextual representations of input tokens and generate coherent text based on the learned representations. It can be fine-tuned on specific tasks or used as a standalone generative model.

Next, we'll provide you with the code to instantiate the GPT model that we defined earlier.

#### 4.11. Creating an instance of GPT model

```
model = GPT()
m = model.to(device)
# print the number of parameters in the model
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters')
```

```
Output: 10.798157 M parameters
```

The above code block performs the following tasks:

1. **Instantiate GPT Model**: It creates an instance of the GPT model by calling the `GPT()` constructor. This initializes the model with the specified architecture and initializes its parameters with random values.

2. **Move Model to Device**: It moves the instantiated model (`model`) to the specified device (`device`). This is typically done to utilize hardware accelerators like GPUs for faster computation. The `to(device)` method is a PyTorch function that moves all model parameters and buffers to the specified device.

3. **Print Number of Parameters**: It calculates and prints the total number of parameters in the model. This is achieved by iterating over all parameters of the model (`m.parameters()`) and summing up the number of elements in each parameter tensor (`p.numel()`). The result is divided by 1 million (`1e6`) to convert the number of parameters to millions (M) for easier readability. Finally, the result is printed along with the label '`M parameters`'.

Overall, this code is useful for understanding the size and complexity of the GPT model, which can help in managing memory usage and optimizing performance during training and inference.

Next, we train our model.

### 4.12. Code for training the model

```
    # create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

# Initialize empty lists to store training and validation losses
train_loss = []
val_loss = []

# Iterate over the training process
for iter in range(max_iters):

    # Evaluate the loss on train and val sets at regular intervals
    if iter % eval_interval == 0 or iter == max_iters - 1:
        losses = estimate_loss()  # Assuming this function estimates losses
        train_loss.append(losses['train'].item())  # Append training loss to train_loss list
        val_loss.append(losses['val'].item())      # Append validation loss to val_loss list
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}")

    # Sample a batch of data
    xb, yb = get_batch('train')

    # Evaluate the loss
    logits, loss = model(xb, yb)
    optimizer.zero_grad(set_to_none=True)  # Zero out the gradients
    loss.backward()                        # Backpropagate the loss
    optimizer.step()                       # Update the model parameters using the optimizer

```

```
# Save the model's parameters
torch.save(m.state_dict(), "/storage/sridinesh/MTP/model_parametes.pkl")
```

The above code block trains our GPT model as explained below:

1. **PyTorch Optimizer Initialization**: The code initializes a PyTorch optimizer using the AdamW optimizer. It is used to optimize the parameters of the model during training. The optimizer is initialized with the parameters of the `model` and a learning rate (`lr`) specified by the `learning_rate` variable.
2. **Initialization of Lists**: Two empty lists `train_loss` and `val_loss` are initialized to store the training and validation losses, respectively.
3. **Training Loop**: The code iterates over the training process for a specified number of iterations (`max_iters`).
   - Within each iteration, it checks whether the current iteration is a multiple of `eval_interval` or if it is the last iteration. If true, it estimates the losses on both the training and validation sets using the `estimate_loss()` function. The obtained losses are then appended to the `train_loss` and `val_loss` lists.
   - Next, it samples a batch of data (`xb`, `yb`) from the training set using the `get_batch()` function.
   - The model is then called with the input batch (`xb`, `yb`) to compute the logits and the loss.
   - Gradients are zeroed out using `optimizer.zero_grad(set_to_none=True)` to reset the gradients from the previous iteration.
   - The loss is then backpropagated through the model using `loss.backward()` to compute gradients.
   - Finally, the optimizer updates the model parameters using the computed gradients via `optimizer.step()`.
4. **Model Parameters Saving**: Once training is completed, the code saves the trained model's parameters to a file using `torch.save()`. The model parameters are saved in the specified file path

   `/storage/sridinesh/MTP/model_parametes.pkl`.

Overall, this code block performs model training using the AdamW optimizer, monitors training and validation losses, and saves the trained model's parameters for future use.

Next, we will present a graph for analyzing the training loss and validation loss in relation to the number of steps (epochs).

### 4.13. Graph to interpret training and validation loss

```
import plotly.graph_objects as go
steps = list(range(0, 11000, eval_interval))
# Create traces
fig = go.Figure()
fig.add_trace(go.Scatter(x=steps, y=train_loss, mode='lines', name='Train Loss'))
fig.add_trace(go.Scatter(x=steps, y=val_loss, mode='lines', name='Validation Loss'))

# Edit layout
fig.update_layout(title='Train and Validation Loss',
                  xaxis_title='Epochs',
                  yaxis_title='Loss',
                  template='plotly_dark',  # You can choose different templates here for the theme
                 )
# Show plot
fig.show()
```

Output:

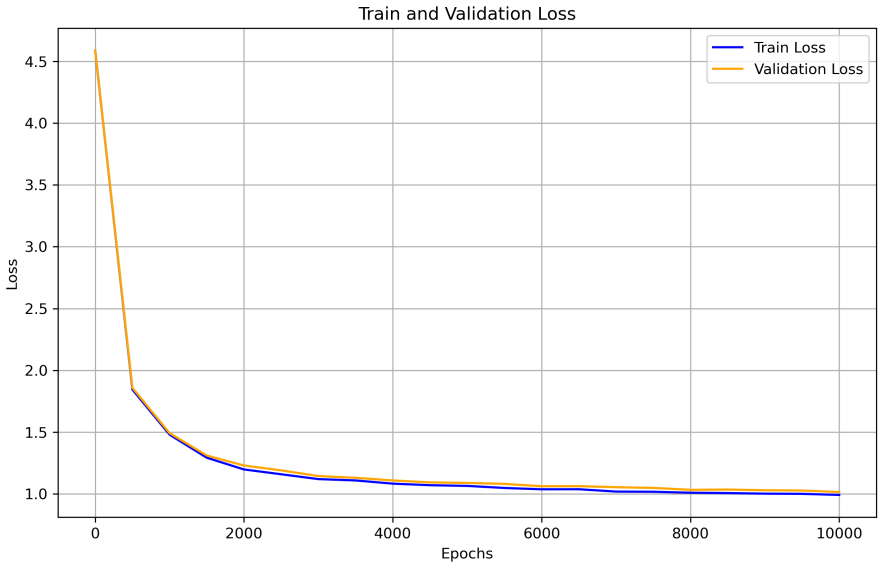


Figure 8: Training and Validation loss Vs No.of Epochs

The provided Python code generates a plot using Plotly to visualize the training and validation loss over epochs. Let's break down the graph:

**X-axis:** The x-axis represents the number of epochs or steps. In this case, steps represents the epochs from 0 to 11000 with a step size of 500. Each point on the x-axis corresponds to one training epoch.

**Y-axis:** The y-axis represents the loss value. It shows how well the model is performing during training and validation. Lower values indicate better performance.

**Data points:**

– The blue line represents the training loss (`train_loss`), showing how the loss decreases over epochs during training.

– The orange line represents the validation loss (`val_loss`), indicating how the loss changes over epochs during validation.

**Interpretation:**

– Initially, both training and validation loss are high, which is expected as the model hasn't learned much.

– As training progresses (i.e., as epochs increase), the training loss decreases gradually, indicating that the model is learning to fit the training data better.

– Similarly, the validation loss also decreases initially, indicating that the model is generalizing well to unseen data.

– The plot allows you to monitor the training process and make decisions on when to stop training to avoid overfitting or underfitting.

**Layout:**

– The title of the plot is "Train and Validation Loss".

– The x-axis label is "Epochs", indicating the number of training iterations.

– The y-axis label is "Loss", representing the value of the loss function.

Overall, the plot provides a visual representation of the training and validation loss trends, which is useful for monitoring and analyzing the performance of the neural network model during training.

Next, we'll utilize our trained GPT to generate new text that reflects the style of the dataset on which it was trained.

### 4.14. Text generation using the trained GPT model (Final Result)

First, we will generate a new song by using the 'new line' character as the starting character. The Output is shown below the code block.

```
# generate from the model
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(m.generate(context, max_new_tokens=800)[0].tolist()))
```

Output:

```
armony, thought overflows legs...count legs..., movy, will legs.
I will look at the ground in the again...
All the time I feed
When armony, the seeds are thy tonight I know now.
Armony, the seeds are wisered
All these wounds are thy tonight

Louino

I need to reap, bet to hold our snow, man, ma.
I never walked in it again.

CRUSPR)
Break it child, break it all right around,
Break it right down tonight, break it, breat it happens I'll inside your life
Happy that's right, happy!

Guker thanks all my tonight
Coffee, oh!
Oh!
Gukililiang, guar.
Whremila islang, whistling, you will
Who won to see.

[Chorus]

Staw thisle on ten arguage
Cinda, action, he'll see when he accement!
They not never be be going on by
Nigg
```

The above code snippet performs text generation using a pre-trained model.

Here's a breakdown:

1. **Context Initialization**:
   - `context = torch.zeros((1, 1), dtype=torch.long, device=device)`: It initializes the context tensor with zeros. The context tensor is a tensor of shape (1, 1) containing a single zero value. This tensor is created using PyTorch's `torch.zeros()` function. The data type of the tensor is set to `torch.long`, indicating it contains integer values. The `device` variable specifies the device (CPU or GPU) on which the tensor should be created.
2. **Generation**:
   - `m.generate(context, max_new_tokens=800)`: This line generates text using the pre-trained model `m`. The `generate` method takes the `context` tensor as input and generates text based on it. The `max_new_tokens` parameter specifies the maximum number of tokens (words or characters) to generate.
3. **Decoding**:
   - `m.generate(context, max_new_tokens=800)[0].tolist()`: This line generates text using the model and converts the output tensor to a Python list. The `[0]` index is used to access the first element of the output tensor, which contains the generated text. The `tolist()` method is then applied to convert the tensor to a Python list containing token indices.
4. **Printing**:
   - `print(decode(...))`: This line prints the decoded text. The `decode` function (not shown in the provided code snippet) is assumed to be a custom function that converts token indices into human-readable text. The generated text is passed as an argument to the `decode` function, and the resulting decoded text is printed to the console.

Overall, this code snippet initializes a context tensor, generates text using a GPT model, decodes the generated text, and prints the decoded text to the console.

Next, we generate a song with "City of Stars" as the beginning of the song, utilizing the GPT model.The Output is shown below the code block.

```
input = encode('City of stars')
l =len(input)
context = torch.zeros((1,l),dtype=torch.long, device = device)
context[0] = torch.tensor(inp)
print(decode(m.generate(context, max_new_tokens=800)[0].tolist()))
```

```
Output:
City of stars
And then clance for shell
Like all that, mily, will like become
The sure that I've been waiting for
All through the morning
Makes me sch me, No
A moment have known

As I walk about the world
The truth of time
Every distance (Will ustoce) yes, yes.
It speaks of day, we gosn't make it worder.
We'll be strong, I feels be no way, no
Big mall in a bridge

We'll raise on the Black rain
And your madness your system
You, my happiness you'll find your way, no
You'll be my head in the Marlous

Ooh Son hones, I had in your thender
Railor all. No, when I Hear soul just so good

Your happiness, if she do you go away
You know love will always be alive
Your gonessings yes, if it's worth the time to stay.
```

## 5. Hardware and Schedule

Our model was trained on a single machine equipped with an NVIDIA GeForce GPU of version 11.7. The total space consumed by our program and data amounts to 4312 megabytes. The duration of model training spanned a period of 2 hours, employing hyperparameter values as delineated in Section 4.

## 6. Results on Tiny Shakespeare dataset and Conclusion

We also trained our GPT model using the Tiny Shakespeare dataset, which we obtained from the following URL: `https://www.kaggle.com/datasets/kaushaltiwari/tiny-shakespeare`. The Tiny Shakespeare dataset is loaded

into our program using the following code snippet:

```
with open('/storage/sridinesh/MTP/tiny_shakespeare.txt', 'r', encoding='utf-8') as f:
    text = f.read()
```

After training the GPT model on the Tiny Shakespeare dataset using all of the parameters specified in section 4, the following output is produced. To generate the output, we gave 'new line' as the initial character in this case. Here we generated 800 tokens (characters).

```
Output:

That CAPULET:
Good false and in outrage, dispatch you;
And your voices: get I'll ear: let not.

BALTHASAR:
Now, no, sice more confessive than you will non.

FRIAR LAURENCE:
A sister than all la-wench'd elky insume stone,
Or enough, for your fatal double plane.

CAPULET:
You writ? and you have very danceless an epeath;
'Tis honest-faced is creased to be so done:
In 'serity that for 'em; 'tis exacule.'

BRARDINE:
It rises more such a leasure of this five: my good a
cold Cominius, blame up, justice, or oddinate, king
To fardely: seem what dreply to blaw'd the world,
But to cut that make on my father
Will not you do look his blow: the seat are
Proseasant widing with your service, if thou vile,
Speed hence to repel, and, were not sacrable persons
The hatch'd with sullest afor shall consul
```

Likewise, we have the capability to train our GPT model on various datasets. By augmenting the dataset's size and adjusting the hyperparameter values accordingly, we can attain performance akin to that of a Large Language Models (LLMs) with our GPT-based language model.

## References


1. Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 27, 3104-3112.

2. Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *Advances in Neural Information Processing Systems*, 28, 3104-3112.

3. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, 30, 5998-6008.

4. Howard, J., & Ruder, S. (2018). Universal Language Model Fine-tuning for Text Classification. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1641-1650.

5. Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving Language Understanding by Generative Pre-training. Retrieved from OpenAI website: `https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf`

6. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Retrieved from arXiv website: `https://arxiv.org/abs/1810.04805`

7. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15, 1929-1958.

8. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

9. Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer Normalization. arXiv preprint arXiv:1607.06450.