# From bigram Models to GPT: Building a Transformer-Based Language Model

**Sayan Dewanjee**

May, 2025

# Contents

# 1 Abstract

This project investigates the evolution of language modeling from basic statistical techniques to powerful neural architectures, with a focus on bridging the conceptual gap between a simple bigram model and a transformer-based GPT-style model. These models, while historically important and computationally simple, are inherently constrained by their inability to capture long-range dependencies in text. To overcome these constraints, we transition to a neural approach by introducing embedding layers that transform discrete tokens into dense vector representations, followed by the integration of self-attention mechanisms and the architectural principles of transformers. The GPT-style model we build incorporates key components such as multi-head self-attention, position encodings, residual connections, feed-forward networks, and layer normalization. Each addition is motivated by the need to enhance the model's capacity to learn complex patterns in sequential data and improve its generalization ability. We compare both models through quantitative metrics such as cross-entropy loss and perplexity, and qualitatively evaluate their ability to generate coherent text. The transformer-based model significantly outperforms the bigram model in both accuracy and expressive power. This project highlights not only the technical improvements but also the educational value of incrementally building from a classical probabilistic model to a modern deep learning system, offering a grounded understanding of the principles that underlie today's state-of-the-art NLP systems.

# 2 Introduction

Language modeling is one of the fundamental problems in Natural Language Processing (NLP), aiming to predict the likelihood of sequences of words or tokens. A language model learns the structure of a language by estimating the probability distribution over sequences, making it essential for a wide range of downstream applications, including machine translation, speech recognition, text summarization, and conversational AI. Traditional statistical models such as bigrams assume that the probability of a word in a sentence depends only on the previous word, a simplification known as the Markov assumption. While these models are easy to implement and interpret, they are limited in their ability to capture longer-range syntactic and semantic dependencies, especially in more complex language generation tasks. Moreover, they suffer from sparsity issues when dealing with unseen word pairs, unless explicit smoothing techniques are used. To address these shortcomings, the field has shifted toward neural models that use distributed representations and learn complex functions over sequences.

In this project, we begin by implementing a character-level bigram model and studying its behavior, strengths, and weaknesses. This model serves as a foundation for understanding sequential prediction tasks and probabilistic modeling. From there, we enhance the model by introducing key components of the transformer architecture—most notably, the self-attention mechanism that allows the model to weigh the relevance of different positions in a sequence when making predictions. We incrementally build up to a GPT-style model that uses multiple attention heads, position embeddings, and stacked transformer blocks to capture richer contextual dependencies. The use of layer normalization and residual connections ensures more stable and efficient training. By comparing these two models—the bigram and the transformer—we aim to demonstrate how modern deep learning techniques can vastly improve the capacity of language models to generate fluent and contextually relevant text. This progression not only underscores the practical benefits of neural architectures but also deepens our theoretical understanding of why and how these methods outperform traditional approaches. Our report reflects this journey from simplicity to sophistication, revealing how each architectural innovation contributes to the power of modern language models.

# 3 Conceptual Framework

## 3.1 Language Modeling

Have you ever wondered how your phone predicts your next word while typing, or how chatbots like ChatGPT seem to "understand" language? Behind these capabilities lies a fundamental concept in Natural Language Processing (NLP) called language modeling.

At its core, a language model is a probability distribution over sequences of words. It tries to answer the question:

> "Given a few words, what is the most likely next word?"

For example, given the phrase "The cat sat on the", a language model might predict that "mat" is more likely to follow than "spaceship". This seemingly simple task underpins a wide range of applications—text generation, translation, speech recognition, autocomplete, and more.

Language models learn from text data by identifying patterns and structures. The better a model understands these patterns, the more coherent and meaningful its predictions become. Over time, language models have evolved from simple statistical methods like n-gram models to powerful neural architectures like Transformers, which use mechanisms such as self-attention to capture deeper context and longer-range dependencies.

In this section, we will explore the fundamental ideas behind language modeling, starting from traditional approaches and building up to modern deep learning-based methods that power today's most advanced AI systems.

### 3.1.1 Definition

Language modeling is the task of estimating the joint probability of a sequence of words $w_1, w_2, ...., w_T$ in a language. Formally, a language model defines the probability:

$$P(w_1, w_2, \ldots, w_T) = \prod_{t=1}^{T} P(w_t \mid w_1, w_2, \ldots, w_{t-1})$$

where $P(w_t \mid w_1, w_2, \ldots, w_{t-1})$ is the conditional probability of the word $w_t$ given all the previous words in the sequence. Since directly modeling these high-dimensional conditional probabilities is challenging, classical approaches like n-gram models approximate this by considering only the previous $n-1$ words, applying the Markov assumption:

$$P(w_t \mid w_1, w_2, \ldots, w_{t-1}) = P(w_t \mid w_{t-(n-1)}, \ldots, w_{t-1})$$

More advanced language models, including neural networks and transformers, learn to estimate these conditional probabilities by encoding richer context representations, enabling them to capture long-range dependencies and complex linguistic patterns.

### 3.1.2 Application

Natural Language Processing (NLP) plays a vital role in enabling machines to understand and generate human language. Some of its key applications include:

- **Machine Translation:** Automatically translating text from one language to another, e.g., Google Translate or DeepL.
- **Text Summarization:** Generating concise summaries of longer documents using extractive or abstractive methods.
- **Sentiment Analysis:** Determining the sentiment (positive, negative, neutral) expressed in text, widely used in social media and product reviews.
- **Chatbots and Virtual Assistants:** Powering conversational agents like Siri, Alexa, and customer service bots through dialogue understanding and generation.
- **Speech Recognition:** Converting spoken language into text, enabling voice commands, dictation, and accessibility features.
- **Named Entity Recognition (NER):** Identifying and classifying entities like names, organizations, dates, and locations in text.
- **Text Classification:** Categorizing text into predefined classes, such as spam detection, topic labeling, or language identification.
- **Question Answering:** Building systems that can directly answer user queries based on a given context, e.g., in search engines or reading comprehension tasks.

## 3.2 N-gram Models

### 3.2.1 What is an n-gram

Consider the sentence

`The monsoon air over the ghats is so beautifully ....`

One can easily conclude the next word will be thick or dense or mystical. But it will not be train. An n-gram model assigns a probability to all such possible words using last $n-1$ words.

Let $P(w \mid h)$ denote the probability of word $w$ given some history $h$. In our context we want to find the probability that next word will be thick

$$P(\text{thick} \mid \text{The monsoon air over the ghats is so beautifully })$$

Now with the definition of probability we can write

$$P(\text{thick} \mid \text{The monsoon air over the ghats is so beautifully }) = \frac{\text{count of } \textit{The monsoon air over the ghats is so beautifully thick}}{\text{count of } \textit{The monsoon air over the ghats is so beautifully}}$$

But as we have a finite text corpus we will not get a good estimates of these probabilities. In any literature we can build new sentence all the time. And hence we can't get the accurate count of such large objects like sentence.

The **Markov assumption** posits that the probability of a word depends only on a limited history—in the simplest case, just the immediately preceding word. Markov models are a class of probabilistic models that leverage this assumption, enabling prediction of future outcomes based on a fixed-length context, rather than the entire past. A bigram model, for instance, considers only the previous word, while a trigram looks two words back. This naturally extends to n-gram models, which condition the probability of a word on the preceding $n-1$ words.

### 3.2.2 bigram Model Implementation

The bigram model is one of the simplest and most foundational probabilistic language models. It estimates the probability of a word based on the immediately preceding word, embodying the first-order Markov assumption. This approach enables basic predictive capabilities and provides insight into the statistical structure of natural language.

Suppose $w_1, w_2, .....w_T$ are words or tokens in our language modeling context. In bigram model we calculate the conditional probability

$$P(w_n \mid w_{n-1}) = \frac{\text{count}(w_{n-1}, w_n)}{\text{count}(w_{n-1})}$$

To find this probability we use a text corpus as training set. We use relative frequency to estimate of occurrence of every bigram $(w_i, w_j)$. Now consider we know that the present token is $w_j$. We want to predict the next token. We assume that given present token is $w_j$,probability distribution of next token follows multinomial distribution. By maximum-likelihood approach we can show mle of probability of next token is $w_k$ is given by relative frequency of $(w_j, w_k)$. Now to generate next token we estimate parameter of multinomial distribution with corresponding maximum likelihood estimator and we consider a sample from corresponding multinomial distribution.

## 3.3 Neural Language Models

### 3.3.1 Why move beyond n-grams

N-gram models have long served as foundational tools in natural language processing, offering a simple and intuitive way to model the probability of word sequences. By assuming that the probability of a word depends only on the preceding $n-1$ words, these models capture local linguistic patterns efficiently. However, despite their utility and simplicity, n-gram models exhibit significant limitations that motivate the need for more advanced modeling techniques.

N-gram models suffer from several significant limitations. They can only capture short-range dependencies due to their fixed context window, making them incapable of modeling long-term relationships in language. As the order $n$ increases, they face severe data sparsity, since many word combinations may never appear in the training corpus, resulting in unreliable probability estimates. These models also generalize poorly, treating semantically or morphologically related words (e.g., "run" and "jog") as entirely distinct tokens. Additionally, n-gram models scale poorly with vocabulary size and sequence length, leading to storage inefficiency and a combinatorial explosion of parameters as $n$ grows. These limitations significantly restrict their performance in real-world language tasks.

To overcome the above limitations, modern NLP has shifted toward neural language models that leverage distributed representations and deep learning architectures:

- Word Embeddings like Word2Vec or GloVe capture semantic similarity and generalize better across vocabulary.
- Recurrent Neural Networks (RNNs) and LSTMs maintain memory across longer sequences, enabling learning from context beyond a fixed window.
- Transformers with self-attention mechanisms allow models to dynamically attend to all positions in the input sequence, efficiently modeling both local and global dependencies.

These approaches not only address the sparsity and context issues but also provide a scalable and robust framework for a wide range of downstream NLP tasks.

### 3.3.2 Neural networks for language modeling

Language modeling is the task of predicting the next word in a sequence, given its context. While traditional approaches like $n$-gram models rely on counting word occurrences, neural network-based models learn to represent and generalize from language data using dense vector representations and nonlinear functions. This section explores key neural architectures that have advanced language modeling.

- **Feedforward Neural Language Models**

  The first neural language model proposed by Bengio et al. (2003) was based on a feed-forward network. It takes a fixed-size context (e.g., the previous $n-1$ words), converts them into word embeddings (continuous-valued vectors), concatenates these vectors, and passes them through a hidden layer to predict the next word:

  $$P(w_t \mid w_{t-(n-1)}, \ldots, w_{t-1}) = \text{Softmax}(f(\mathbf{e}_{t-(n-1)}, \ldots, \mathbf{e}_{t-1}))$$

  This model overcomes the sparsity of $n$-grams by learning word similarities through embeddings. However, it still uses a fixed-size window and cannot dynamically adjust to longer contexts.

- **Recurrent Neural Networks (RNNs)**

  RNNs solve the fixed-window limitation by introducing a hidden state that is passed along the sequence, maintaining a form of memory:

  $$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{e}_t)$$

  $$P(w_{t+1} \mid w_1, \ldots, w_t) = \text{Softmax}(\mathbf{W}_o \mathbf{h}_t)$$

  This enables the model to use information from the entire past sequence. However, basic RNNs have difficulty preserving information over long distances due to the vanishing gradient problem.

- **LSTMs and GRUs**

  To improve the ability to capture long-range dependencies, Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs) introduce gates that control the flow of information:

  - The **forget gate** decides what information to discard.
  - The **input gate** selects what new information to store.
  - The **output gate** determines the hidden state to pass forward.

  These mechanisms allow LSTMs/GRUs to retain relevant context over much longer sequences compared to standard RNNs, making them widely used in NLP tasks before the rise of transformers.

- **Transformers**

  Transformers revolutionized language modeling by completely removing recurrence. Instead, they use self-attention, allowing each word to directly attend to every other word in the input, regardless of distance:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

  Here, $Q$, $K$, and $V$ are learned projections of the input embeddings. This architecture enables better modeling of both local and global dependencies and allows for massive parallelization during training. Models like BERT and GPT, built on transformer blocks, have achieved state-of-the-art results in virtually every NLP benchmark.

  Neural language models offer several advantages over traditional methods:

  - They generalize better by learning semantic representations (e.g., ”king” and ”queen” are related).
  - They handle variable-length context effectively (RNNs, LSTMs, Transformers).
  - They scale well with data and compute, improving performance with larger corpora.
  - They are end-to-end trainable, requiring no manual feature engineering.

  Neural networks have transformed language modeling from a statistical pattern-matching task into a robust, learnable function capable of understanding and generating human language.

## 3.4 Transformers and GPT

### 3.4.1 Self-attention Mechanism

The self-attention mechanism is a powerful technique that allows a model to weigh the importance of different words in a sequence when encoding a particular word. Unlike recurrent models, which process inputs sequentially, self-attention considers all positions simultaneously, capturing both local and global dependencies more effectively. This mechanism forms the core of transformer architectures and enables them to model long-range context with high efficiency and flexibility.

In simple words,self-attention is a mechanism to gather relational information among words to understand the context precisely. The best thing to accommodate this understanding is to take a weighted average of embedding vectors of our tokens.

Consider we have $x_1, x_2, \ldots, x_N$ as inputs each of dimenson $D \times 1$. Now these inputs represent word or word fragment in context of language modeling. At first *values* are computed from inputs which are also of dimension $D \times 1$. Values are computed in following way :

$$v_m = \beta_v + \Omega_v x_m$$

where $\beta_v$ and $\Omega_v$ is of dimension $D \times 1$ and $D \times D$.

n-th output of self-attention block will be

$$sa_n[x_1, \ldots, x_N] = \sum_{m=1}^{N} a[x_m, x_n] v_m$$

Hence this is a weighted sum of values. In other words $a[x_m, x_n]$ is the attention paid by $x_n$ to $x_m$. Hence $a[\bullet, x_n]$ is non-negative and will be summed to one.

In GPT architecture we process tokens in a sequential manner.
Consider the sentence
`The monsoon air over the ghats is so beautifully...`
The transformer will take follwoing tokens in a sequential manner:
It takes `The` as input and `monsoon`
It takes `The monsoon` as input and `air`
It takes `The monsoon air` as input and `over`
In such architecture the transformer will look upon past words only and do not know about future words. Hence when we will pass `The monsoon air` weights for `(ghats,monsoon)` or `(beautifully,monsoon)` will be zero as our transformer only encounters `The monsoon air`. It knows nothing about `beautifully`. We will see the attention matrix in this case will be a lower triangular matrix.

### 3.4.2 Computing Attention Weights

Consider $x_1, x_2, \ldots, x_N$ as inputs of dimension $D \times 1$. Now we will compute two more projections from these inputes - *keys* $k_m$ and *queries* $q_m$, both of dimension $D_h \times 1$.

Note :
In self-attention mechanism

- **Query** is what we are currently focusing on — it represents the word we are trying to understand in the current position.
- **Key** is the reference information from all words in the sequence — what other words "offer" for comparison.

So if we take dot product between a key and query will get the attention current word (query) should pay to each of the other words (keys).

Queries and keys are computed in following ways :

$$q_n = \beta_q + \Omega_q x_n$$

$$k_m = \beta_k + \Omega_k x_m$$

Then we compute dot-product and pass it through soft-max function to obtain attention weights $a[x_m, x_n]$.

$$a[x_m, x_n] = \frac{exp(k_m^T q_n)}{\sum_{i=1}^{N} exp(k_i^T q_n)}$$

It is called *dot-product self-attention*.

In matrix notation, if the $N$ inputs $x_n$ forms the column of input matrix $X$ then value,key and query will be given by :

$$V = \beta_v 1^T + \Omega_v X$$
$$K = \beta_k 1^T + \Omega_k X$$
$$Q = \beta_q 1^T + \Omega_q X$$

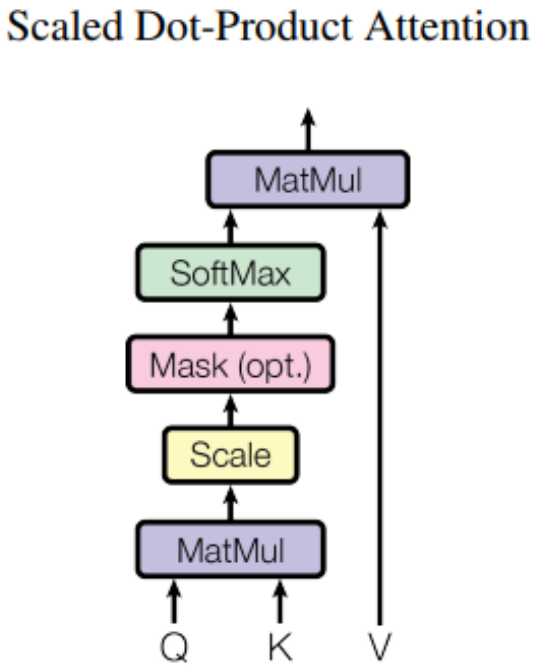
And self-attention is given by :

$$SA = Softmax(QK^T)V$$

**Note:**
In the paper *Attention is all you need* the dot product is standardized by scaling it.

$$SA = Softmax(\frac{K^T Q}{\sqrt{D_h}})V$$

For large values of $D_h$ dot-products grow large in magnitude , making convergence of optimization function hard. To handle this we have to scaled down our dot-product attention in appropriate manner.




*Reference : Attention is all you need*

### 3.4.3 Multihead Attention

Till now we have computed only a single head of self-attention. Now if we apply multiple heads, then different heads can learn different aspects of the input like subject-verb agreement,semantic similarity, long-range dependencies etc.

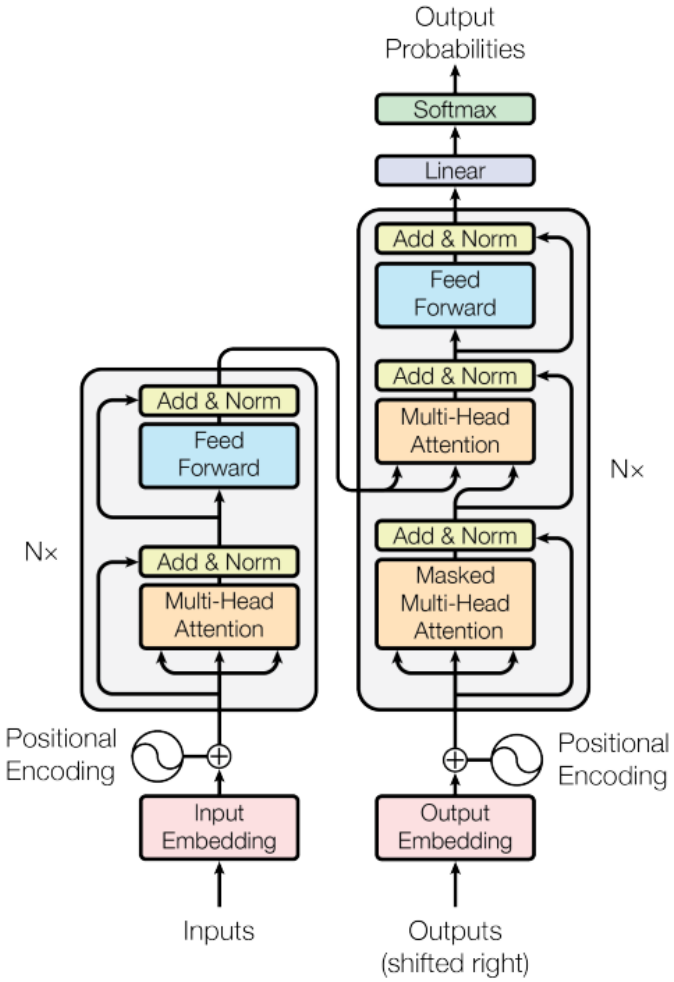
Typically, if the dimension of input is $D$ and we are using $H$ different heads, then the dimension of value, key and query will be $D/H$. For input $X$ outputs from $H$ different heads are $SA_1, \ldots, SA_H$ each of dimension $D/H$. Then we concatenate all those outputs vertically and will take a final linear projection.

$$\text{MHSA} = \Omega_c[SA_1^T, \ldots, SA_H^T]^T$$

### 3.4.4 Transformer Architecture

The Transformer is a deep learning architecture introduced by Vaswani et al. in 2017, which revolutionized the field of natural language processing. Unlike traditional recurrent models, Transformers rely entirely on attention mechanisms to model relationships between words, enabling efficient parallelization and superior performance on long sequences. The core idea is to allow each word in a sentence to attend to every other word, capturing contextual dependencies more effectively and flexibly. This design has become the foundation for modern language models such as BERT, GPT, and T5.

Here is how a typical *transformer* block looks like:




*Reference : Attention is all you need*

The rectangle shaped box on the right-hand side duagram is a typically *transformer* block. It consists 3 parts :

- A masked Multi-head attention block which takes sum of token embedding andpositional encoding as input.

- Another block of multi-head attention which is cross-attained.
- A final position-wise feed forward neural network which is applied to each position separately and identically. In feed-forward block it consists of two linear layer with a ReLU activation in between.

$$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$$

There are 2 more important tools in this block. They are :

- **Layer normalization** : Layer normalization improves training stability and speed in Transformers by normalizing activations within each layer. It ensures consistent input distribution, enhances gradient flow, and works well with variable-length sequences, making it ideal for deep NLP models.
- **Residual Connection** : A residual layer is used before multi-head attention and feed forward layer to help preserve information and improve gradient flow during training. By adding the input of a layer to its output (output = x + f(x)), they allow the model to learn modifications to the input rather than the full transformation, making it easier to train deep networks. This helps prevent vanishing gradients, stabilizes learning, and enables better reuse of features across layers.

The whole process can be written as :

$$\begin{aligned} X &\leftarrow X + \text{MHSA}(X) \\ X &\leftarrow \text{Layer norm}(X) \\ x_n &\leftarrow x_n + \text{FFN}(x_n) \\ X &\leftarrow \text{Layer norm}(X) \end{aligned}$$

**Note**:

- We use Drop-out layers also as this model is pretty dense.
- In the paper the dimension of layers in positional forward networks is $512 \times 2048$.

### 3.4.5 GPT-style Autoreegressive Model

Generative Pre-trained Transformers (GPT) represent a class of autoregressive language models that have significantly advanced the field of natural language processing (NLP). Developed initially by OpenAI, the GPT family employs transformer architectures to model the probability distribution of text sequences, enabling highly fluent and context-aware language generation.

Consider the sentence : `It is very bright.` Let's consider here each individual word as a token.Then probability of sentence can be written as

$P(\text{It is very bright})) =$
$P(\text{It})\times$
$P(\text{is} \mid \text{It})\times$
$P(\text{very} \mid \text{It is})\times$
$P(\text{bright} \mid \text{It is very})$

An autoregressive model computes such conditional probability of current token conditioned on presence of all prior tokens. It computes

$$P(t_1, t_2, \ldots, t_N) = P(t_1) \prod_{i=2}^{N} P(t_i \mid t_1, t_2, \ldots, t_{i-1})$$

This formulation allows the model to generate text one token at a time, with each token conditioned on all previously generated tokens.

In GPT architecture we use a special type of self-attention,namely masked self-attention. Masked self-attention is a variant of the self-attention mechanism used in autoregressive models like GPT, where attention is restricted such that each token can only attend to previous tokens and itself, but not to any future tokens in the sequence. This is achieved by applying a causal mask—a triangular matrix that sets the attention weights for future positions to negative infinity before the softmax operation. The masking preserves the autoregressive property, preventing information leakage from future tokens and allowing the model to be trained in parallel while still simulating sequential token prediction. Hence when it takes input It is,model does not get any information from very bright.

# 4 Data Preparation

- **Source :** We have used `tiny shakespeare` dataset.
- **Cleaning and Preprocessing :** We have implemented a character-level model. Here we have considered all lowercase letters,uppercase letters and special characters.
- **Tokenization :** To build a character-level model, we have considered every character as a token.
- **Vocabulary Construction :** To create our vocabulary we have considered all unique characters from our text corpus. We have $65$ unique characters in our text corpus. Hence our vocabulary size is $65$. Our vocabulary consists -
  - Special characters like `!$',-.3:;?`
  - All uppercase letter like `ABCDEFGHIJKLMNOPQRSTUVWXYZ`
  - All lowercase letter like `abcdefghijklmnopqrstuvwxyz`

  We have assigned token id $1$ to first special character, $2$ to second special character and so on. Hence token id of `z` is $65$.
- **Input-Target Sequence Construction :** Consider $[w_1, w_2, \ldots, w_{n+1}]$ is a block of size $n+1$. We have $n$ inputs from such block. When $w_1$ is input,target is $w_2$. If $[w_1, w_2]$ is input target is $w_3$. In this way,we have inputs from one block.
- **Batching :** To train our models efficiently, we divided the input data into smaller groups called batches. Instead of processing the entire dataset at once, which can be computationally expensive, the model is trained on mini-batches.
- **Train-Validation Split :** We considered the first $90\%$ of the text as training data, and the remaining $10\%$ as validation data.

An excerpt from `tiny shakespeare` dataset:

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You
```

# 5 Bigram Model Implementation

We implement a bigram language model that learns the probability of a word given the previous word. This simple model captures local word dependencies and enables basic text generation. We use pytorch module for that.

## 5.1 Architectural Overview

We will explain our *character-level Bigram model* in a step by step manner.

- **Step 1 :** `nn.Embedding` is used to create embedding layer. Each input character (as an index) is mapped directly to a logits vector over the entire vocabulary. It gives us the look-up table
- **Step 2 :** We pass tokens of size `block_size` batch by batches.
- **Step 3 :** We calculated logits and losses. We have used `cross_entropy` loss function.
- **Step 4 :** To generate new tokens, we pass the starting token and considered the corresponding logits. We pass the logits through `softmax` to obtain probabilities. Then we take a sample from multinomial distribution with obtained probabilities to generate new token.

Here is the code of a simple bigram model:

```
# bigram model
class BigramModel(nn.Module):

  def __init__(self,vocab_size):
    super().__init__()
    self.embedding_table = nn.Embedding(vocab_size,vocab_size)

  def forward(self,id,targets=None):
      # calculating logits and losses
      # id is a tensor of shape (B,T)
      logits = self.embedding_table(id)

      if targets is None :
        losses = None
      else:
        B,T,C = logits.shape
        logits = logits.view(B*T,C)
        targets = targets.view(B*T)
        losses = F.cross_entropy(logits,targets)

      return logits,losses
```

```
def generator(self,id,new_tokens):
    # here we are starting with a new id with dimension B*T
    for a in range(new_tokens):

      logit,loss = self(id)
      logit = logit[:,-1,:]
      probs = F.softmax(logit,dim=1)
      newid = torch.multinomial(probs,num_samples=1)
      id = torch.cat((id,newid),dim=1)

    return id
```

## 5.2 Implementation Details

Hyperparameters of our model are given by :

- **Block size** = $8$
- **Batch size** = $4$
- **Maximum training iterations** = $5000$
- **Evaluation iterations** = $50$
- **patience** = $7$
- **Learning rate** = $0.001$

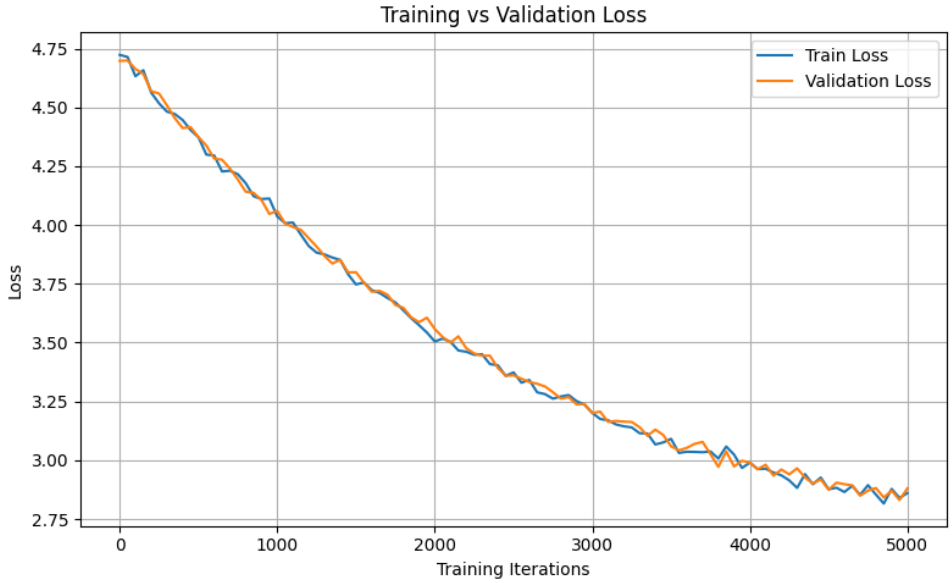
We have used `AdamW`(Adam optimizer with weight decay) optimizer. We have implemented four differrent model :

- Model 1 : `block_size` = $8$ , `batch_size` = $4$ with learning rate = $0.001$
- Model 2 : `block_size` = $8$ , `batch_size` = $32$ with learning rate = $1e-1$ and $1e-3$
- Model 3 : `block_size` = $16$ , `batch_size` = $32$ and $64$ with learning rate = $1e-1$
- Model 4 : `block_size` = $256$ , `batch_size` = $64$ with learning rate = $1e-1$

## 5.3 Training Process

### Model 1

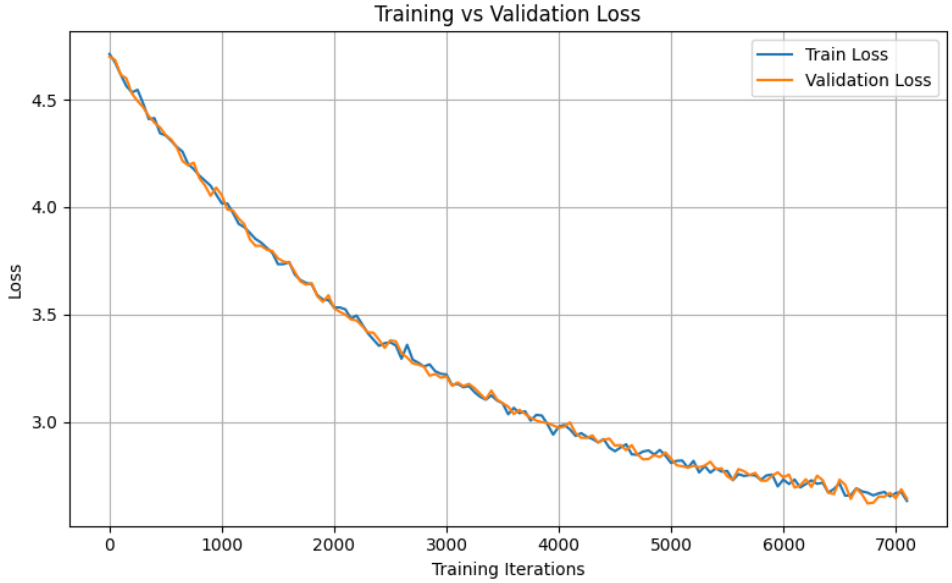
We have implemented a training loop with $5000$ iterations. We have evaluated at every $50$ steps. We have also used a early-stopping. We stopped our training if in $7$ consecutive evaluation steps validation loss does not improve.




```
step 4999: train loss 2.8604, val loss 2.8802
No improvement for 1 evaluations.
```

Both training loss and validation loss decreases steadily which indicates that the model is learning and optimizing correctly. As both the curves are decreasing and early stopping is not triggerred we should train this model for more iteration. The gap between both loss curves are very small,indicating the model generalizes well to unseen (validation) data.

Now we trained our model by setting `max_iters` 10000.

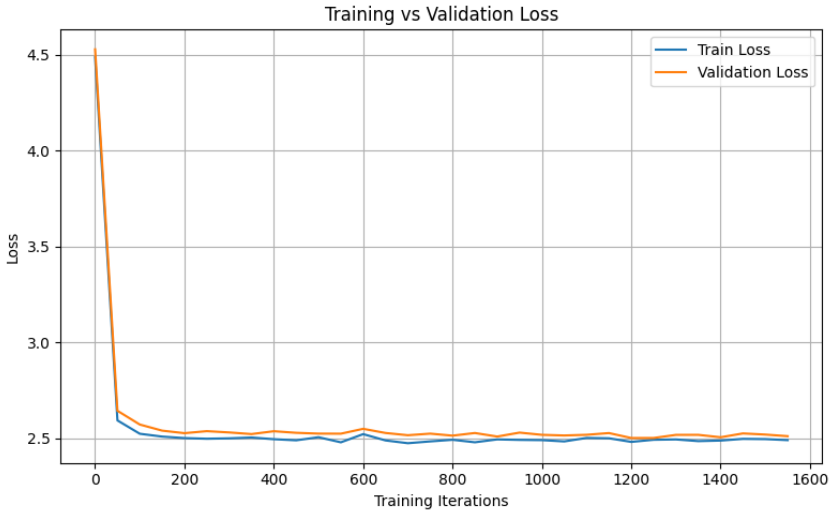



```
step 7100: train loss 2.6321, val loss 2.6431
 No improvement for 7 evaluations.
 Early stopping triggered.
```

Validation loss is decreased by $0.2$ and early stopping triggerred at $7100$ steps. There are no signs of overfitting.

### Model 2

We have changed `batch_size` to $32$ and kept all other hyperparameters same. We implemented $2$ different learning rate of $1e-1$ and $1e-3$. Training and validation loss is given by :




(a) learning rate = $1e-1$


(b) learning rate = $1e-3$

Figure : Changing learning rate

The model trained with a learning rate of $1e-3$ shows smooth and stable convergence, with both training and validation loss gradually decreasing over time and reaching a lower final loss. In contrast, the model with a $1e-1$ learning rate exhibits a rapid initial drop in loss but quickly

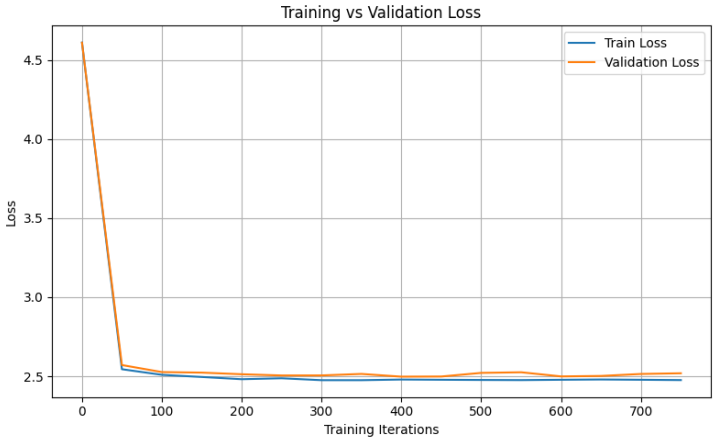
plateaus and fluctuates, indicating potential instability or overshooting during optimization. While $1e-1$ leads to faster initial learning, $1e-3$ provides more reliable and consistent training performance, ultimately achieving better generalization.

In both the cases validation loss is $2.5$. The only difference is first model optimized at $1550$ steps and second model optimized at $6000$ steps.

From now on we will fix our learning rate at $1e-1$.

**Model 3**

We changed $block\_size$ to $16$ and $lrate$ to $1e-1$ and implemented two model with $batch\_size$ $32$ and $64$. Training and validation loss is given by :




(a) batch size= $32$


(b) batch size = $64$

Figure : Changing batch size

Both models demonstrate quick initial convergence, but the second graph achieves a lower validation loss and thus better perplexity, indicating superior generalization despite a slightly wider

gap between train and validation losses. In contrast, the first graph shows tighter alignment between the two losses, suggesting stability but with slightly higher final validation loss. Overall, the second model performs better in terms of validation accuracy and perplexity, even if the training-validation gap is a bit more pronounced.

So,increasing `batch_size` helps to reduce validation loss. Another point is that in almost all cases validation loss will be around $2.50$.

### Model 4

We used a batch size of $64$ and a block size of $256$, which are standard choices commonly adopted in training transformer-based language models.


The loss curve shows a sharp initial drop followed by a long plateau, indicating quick convergence in the early stages of training. Both training and validation losses remain low and closely aligned throughout, suggesting stable learning with no signs of overfitting. The model appears to have reached convergence efficiently, with consistent performance across both datasets.Validation loss for this model is given by $2.4926$.

Despite varying hyperparameters, the model consistently converges to a validation loss of approximately $2.50$, indicating that this is likely the optimal performance achievable with the current model architecture, dataset, and objective (e.g., negative log-likelihood).

- When the learning rate was too small, convergence was slow but stable.
- When the learning rate was large, convergence was fast but risked instability.
- Increasing the batch size led to smoother and slightly better generalization, but only marginally.

- Regardless of the path taken, the final validation loss remained near 2.50, suggesting that the model is capacity-limited or that the task/dataset has inherent complexity which constrains further improvement.

## 5.4 Evaluation

We have used perplexity metric to evaluate our models. Model 4 achieves the lowest perplexity

| **Model** | **Specifications** | **Perplexity** |
|---|---|---|
| Model 1 | max iterations = 5000 | 16.99 |
| | max iterations = 10000 | 14.19 |
| Model 2 | Learning rate = $1e-1$ | 12.34 |
| | Learning rate = $1e-3$ | 12.59 |
| Model 3 | batch size = 32 | 12.22 |
| | batch size = 64 | 12.09 |
| Model 4 | $\times$ | 12.05 |

Table 1: Perplexity for different models

of 12.05, making it the best-performing configuration overall. However, Model 3 with batch size 64 also performs nearly as well with a perplexity of 12.09, indicating that increasing batch size contributes significantly to improved performance. Both models show strong generalization ability.

## 5.5 Text generation

Here we are giving a text generated by `Model 4`:

```
VIfr'ld
Wotree ill oure yow, IULLUENTh shein s.
AByouro auireat nd y mepayoftld I at and h omorikerbo wonderores:

Wed avethalltus mang t
BRomy wns, lo't wndr t:

Ap

Whays.
Foonkn ss the ngethet pe.
Co iththetisus E fin

cotl the CES: IUK:
HOLous ta feyor ublupore e, ouen
ONCOREEROu tor d t.
```

```
VI'dat fat inangrotadd s ne,
FRUMu w; y arshaith thaleror w ISppimise,
I go h fenan thay f st shtho
AK:

Hour l lo thepengigontca wooven ode ma s
Shene pespar I oblo farerl lurdeirifimbied t thinourd s bar
```

The model demonstrates an ability to generate plausible word fragments, punctuation usage, and sentence-like flows, despite the lack of meaningful content. This reflects the typical behavior of character-level language models trained on limited data or with shallow architectures — they can capture local dependencies and stylistic patterns, but struggle with long-range coherence and grammatical correctness.

# 6 Transformer Based Model Implementation

## 6.1 Architectural Overview

We have implemented a step-by-step procedure to update our bigram model.

- **Step 1:** We replaced the one-hot encoding of tokens with a learnable embedding of dimension 384, instead of the original `vocab_size`-dimensional vectors. We also introduced a positional embedding to retain sequence order, and added it element-wise to the token embeddings to form the input tensor `x`.
- **Step 2:** We constructed self-attention blocks and combined them into multi-head attention mechanism, allowing the model to attend to different parts of the input sequence in parallel.
- **Step 3:** A full transformer block was built using:
  - A layer normalization applied to the input `x`.
  - Multi-head self-attention layers, followed by a residual connection with the original input.
  - Another layer normalization and a position-wise feed-forward network.
  - A second residual connection to preserve input information and stabilize training.
- **Step 4:** The input `x` is passed through the transformer block a predefined number of times (i.e., stacked layers). Finally, we apply one more layer normalization followed by a linear projection to the vocabulary space, producing the logits used for prediction.

This modular upgrade from a bigram model introduces the key components of modern language models—namely embeddings, self-attention, normalization, and residual connections. With these enhancements, the model gains the ability to capture long-range dependencies and nuanced language patterns that simple n-gram models cannot handle.

## 6.2 Implementation Details

We have used following hyperparameters :-

- **Batch size:** 64
- **Block size:** 256
- **Number of heads:** 6

- **Head size:** $64$
- **Embedding length:** $6 \times 64 = 384$
- **Number of transformer layers:** $6$
- **Dropout rate:** $0.2$
- **Evaluation iterations:** $50$
- **Maximum training iterations:** $5000$
- **Learning rate:** $3 \times 10^{-4}$
- **Device:** `cuda` if available, else `cpu`
- **Patience :** $7$

Our model has $11387969$ parameters.

We have used `AdamW`(Adam optimizer with weight decay) optimizer and`cross_entropy` as loss function. We have also used a early-stopping. We stopped our training if in $7$ consecutive evaluation steps validation loss does not improve. We have implemented a transformer block with $6$ transformers layers.

## 6.3 Training Process

The model was trained using `cuda`, with each set of $50$ training steps requiring roughly $1$ minute.

Last 7 training and validation losses are given by :

```
step 4100: train loss 1.1161, val loss 1.4859
 No improvement for 1 evaluations.
step 4150: train loss 1.1130, val loss 1.4888
 No improvement for 2 evaluations.
step 4200: train loss 1.1097, val loss 1.4875
 No improvement for 3 evaluations.
step 4250: train loss 1.1072, val loss 1.4941
 No improvement for 4 evaluations.
step 4300: train loss 1.1024, val loss 1.4870
 No improvement for 5 evaluations.
step 4350: train loss 1.0980, val loss 1.4907
 No improvement for 6 evaluations.
step 4400: train loss 1.0969, val loss 1.4807
 No improvement for 7 evaluations.
 Early stopping triggered.
```

Our best bigram model had validation loss around $2.49$. But this model reduced it almost by $1$. It took $4400$ steps for convergence.

The plot shows that training loss steadily decreases, indicating effective learning, while validation loss initially drops sharply and then plateaus around 1.5 after approximately 2000 iterations. The growing gap between training and validation loss suggests mild overfitting, but not to a concerning degree. Since the validation loss shows little improvement beyond a certain point, further training may yield diminishing returns.

## 6.4 Evaluation

We have used perplexity metric and achieved perplexity score of $4.39$ which is almost $\frac{1}{3}$rd of our previous perplexity score.

## 6.5 Text generation

This is the text generated by our model of $2000$ characters:

```
Stars for.

RICHARD:
Sor, oftellow as doubt ristings low, with nothing,
Dorset sorr's bound only princess in love:
So, with the glass men on this three son win.

EDWARD:
Outle, the very colour's good in the eagle.
```

```
Go, Margare, for; thou shalt without help.

Nurse:
Here pither Lord of Warwick instruce,
And hath see up footen use to your grace
Till half him to childin, who sway, kill their
Enrich, how our father defenced to make
I contune, sir sensely with opprison! cannot do now
A older fellow: that then are tell it.

LEONTES:
if you indeed, famous battle tonguely thy sons
That show thou bid a fear and man:
Dear thou hath, lipply; and sauth us me not.

DORSET:
With all my old Mariana,
The measure the order limps on the show,
Where do or nothing!

HENRY BOLINGHASOR:
Unless far Aufidius fatal grief lain.
But, Belingbroke, like not myself.

LEONTES:
Then be come banished of good atons;
For one light of the highest forswearful of great.
I'llow you not go show with grief, and myselfsar
Have vanget by one was punish'd where? wishconten night
To hear not for a man! thou better outgness,
And in his stealed endiness received him honour!
If you hope, for his overtake highness thrift;
And, out made the stig, were angry an one,
Or breath, cold Be against the sun's enter'd's naturate,
Which whom False was hate puppose his chatle, Baptistis,
To mean at Tybalt, my Time ward, wearse
Cannot me bold to cock of a tomb; wenched him then
Dores as your gage the coursey will fast you.
But, I seed.

RUTLAN:
That makes boldly, have took'n him meet he knicks:
Why, 'tis a brave death. he is advensed
With his thousations, he's wise.

First Citizen:
What, or slander head!
```

This text demonstrates strong learning of structural patterns, punctuation, line breaks, and archaic syntax. While the generated content lacks coherent meaning or grammar, it convincingly mimics the texture of Elizabethan English, including invented but plausible Shakespeare-sounding words. The model excels at surface-level imitation due to its character-level training but struggles with semantic consistency or narrative logic, which is expected given its limited context window and lack of word-level understanding. Overall, a perplexity of $4.39$ aligns well with the quality of this syntactically rich but semantically nonsensical output.

# 7 Comparison

We implemented two language models for comparison: the best bigram model(Model 4) and a Transformer-based architecture. The evaluation focuses on loss, perplexity, and the quality of generated text.

In terms of loss,the bigram model's loss quickly plateaus around 2.5, indicating limited learning capacity and early convergence. In contrast, the Transformer model shows a steady decline in both training and validation loss over many iterations, reflecting its ability to learn richer patterns and generalize better. This demonstrates the Transformer's superior modeling capability over the simpler bigram approach.

Perplexity, a measure of how well a model predicts a sample, further highlights the performance gap. The bigram model achieved a perplexity of approximately $12.05$, whereas the Transformer model reached a much lower perplexity of $4.39$. This represents a significant improvement, suggesting that the Transformer assigns higher confidence to the correct sequences and learns more meaningful patterns in language data.

The difference in generation quality is also evident. The bigram model generates locally plausible but short-sighted and often repetitive text. Its reliance on just one preceding word limits its fluency and coherence. On the other hand, the Transformer model produces text that is more fluent, though contextually inconsistent but grammatically structured, owing to its self-attention mechanism and ability to consider longer contexts.

In conclusion, while the bigram model serves as a simple and interpretable baseline, the Transformer architecture significantly outperforms it across all metrics. It demonstrates the advantages of deep neural networks and attention mechanisms in modeling natural language, making it a far more powerful tool for modern language modeling tasks.

# 8 Discussion

## 8.1 Insights Learned

Throughout this project, we have explored and compared two distinct approaches to language modeling: the classical bigram model and the modern Transformer-based architecture. The following insights summarize our key observations and learning outcomes:

- The bigram model is easy to implement and train, but it lacks the ability to capture long-range dependencies in text.
- Perplexity achieved by the bigram model was significantly higher, indicating poorer predictive performance.
- The Transformer model, with self-attention and deeper architecture, learns richer contextual relationships.
- Training loss and validation loss in the Transformer decrease steadily, reflecting better generalization.
- Use of techniques like multi-head attention, positional encoding, and residual connections contribute to the model's stability and expressiveness.
- Perplexity dropped to nearly one-third with the Transformer, demonstrating a substantial performance improvement.
- Transformer-generated text is more coherent, context-aware, and structurally richer compared to the output from the bigram model.
- Training Transformers is computationally more intensive but yields far better results for language modeling tasks.

## 8.2 Improvement of Architectural Components

In this section, we highlight possible enhancements and architectural refinements that could further improve the performance of both the bigram and Transformer-based models. These suggestions are based on known limitations and current best practices in language modeling.

### Possible Improvements for the bigram Model

- **Add Smoothing Techniques:** Applying methods like Laplace or Kneser-Ney smoothing can reduce zero-probability issues and improve generalization, especially for rare word pairs.
- **Increase Context Window:** Extending to tri-gram or n-gram models would allow the model to utilize longer histories for prediction.
- **Incorporate POS Tags or Word Features:** Adding part-of-speech tags or word categories can inject linguistic information into the model.

### Possible Improvements for the Transformer Model

- **Advanced Tokenization:** Replacing simple character-level tokenization with tools like `tiktoken` or SentencePiece can lead to more meaningful and compact representations.
- **Hyperparameter Tuning:** Experimenting with the number of layers, head sizes, and embedding dimensions can lead to more optimal architectures.
- **Weight Sharing and Layer Dropout:** These techniques can help reduce model size and overfitting while maintaining performance.
- **Use of Pretrained Embeddings:** Incorporating pretrained embeddings like GloVe or Word2Vec can provide better initialization.
- **Curriculum Learning or Scheduled Sampling:** These training strategies can help the model learn more robustly, especially in early stages.

# 9 Conclusion

In this report, we have documented the process of building a language model pipeline starting from a simple bigram model and extending it toward a Transformer-based architecture inspired by GPT. Each stage of development introduced new concepts, from probabilistic modeling and embeddings to multi-head self-attention and residual learning. This progression offered a hands-on perspective on how modern language models are constructed and why they outperform simpler approaches.

One of the core insights gained is the significance of contextual depth in language modeling. While the bigram model offered a basic understanding of local dependencies, it struggled with long-range structure and semantic coherence. In contrast, the Transformer model, through its attention mechanisms, was able to capture relationships across the entire input sequence, leading to significantly lower perplexity and more natural text generation.

However, this journey was not without challenges. Designing the architecture from scratch required careful tuning of hyperparameters, efficient batching strategies, and an understanding of both training dynamics and model scaling. Furthermore, implementing components such as layer normalization, residual connections, and positional embeddings from first principles enhanced our understanding of their individual contributions.

Looking ahead, there are several promising directions for improvement. Applying advanced tokenization strategies like Byte-Pair Encoding (BPE) or `tiktoken` could help compress the input space and improve generalization. Pretraining the model on larger corpora before fine-tuning could significantly enhance fluency and robustness. Techniques such as learning rate scheduling, weight sharing, or sparse attention mechanisms may help scale the model efficiently while maintaining accuracy.

Ultimately, this project served as a valuable exercise in bridging traditional statistical modeling with modern deep learning techniques. It opens up pathways for exploring even more sophisticated architectures and contributes to our broader understanding of natural language generation systems.

# 10 References

**Research Paper :**


1. Attention Is All You Need, Ashish Vaswani et al.
2. Dropout Reduces Underfitting, Zhuang Liu et al.
3. Language Models are Few-Shot Learners,Tom B. Brown et al.


**Blog posts and Tutorials :**


1. Code your GPT2 Architecture from Scratch in PyTorch! By Srijit Mukherjee
2. Let's build GPT: from scratch, in code, spelled out. By Andrej Karpathy
3. Yes you should understand backprop
4. The Simplest Language Model: Character Level Bigram
5. Deep Dive into AI: Building a Bigram Language Model and Practicing Patience!
6. N-Gram Language Modelling with NLTK
7. Perplexity for LLM Evaluation
8. What is self-attention?


**Books :**


1. Understanding Deep Learning - Book by Simon J. D. Prince
2. Speech and Language Processing by Daniel Jurafsky  James H. Martin

# 11 Appendices

## Full Model Architecture

### bigram Model:

```
# bigram model
class BigramModel(nn.Module):

  def __init__(self,vocab_size):
    super().__init__()
    # embedding vectors over all vocabulary
    self.embedding_table = nn.Embedding(vocab_size,vocab_size)


  def forward(self,id,targets=None):
      # calculating logits and losses
      # id is a tensor of shape (B,T)

      logits = self.embedding_table(id)
      # stacking logits based on those id hence shape is (B,T,vocab_size)

      if targets is None :
        losses = None
      else:
        B,T,C = logits.shape
        logits = logits.view(B*T,C)
        targets = targets.view(B*T)
        losses = F.cross_entropy(logits,targets)

      return logits,losses

  def generator(self,id,new_tokens):

      # here we are starting with a new id with dimension B*T

      for a in range(new_tokens):

        logit,loss = self(id)
        # logit is obtained for new id with shape (B,T,C)
        logit = logit[:,-1,:]
        # considered the last id from every batch hence shape (B,C)
        probs = F.softmax(logit,dim=1)
        # turning logits into probability
        newid = torch.multinomial(probs,num_samples=1)
        # sampling from multinomial distribution
        id = torch.cat((id,newid),dim=1)

      return id

model = BigramModel(vocab_size)
```

**Transformer based Model:**

```
# creating final gpt model with transfer block
class final_GPTModel(nn.Module):

  def __init__(self,vocab_size):
    super().__init__()
    # embedding vectors over all vocabulary
    self.token_embedding = nn.Embedding(vocab_size,embed_length)
    self.position_embedding = nn.Embedding(block_size,embed_length)
    # block_size is maximum sequence length in a batch
    self.sa_multihead = multihead(num_head,head_size)
    self.blocks = nn.Sequential(*[transformer(num_head,head_size) for _ in range(n_layer
                                nn.LayerNorm(embed_length))
    self.feed_forw_1 = nn.Linear(embed_length,vocab_size)
    self.final_norm = nn.LayerNorm(embed_length)

  # weight initialization
  def _init_weights(self, module):
      if isinstance(module, nn.Linear):
          torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
          if module.bias is not None:
              torch.nn.init.zeros_(module.bias)
      elif isinstance(module, nn.Embedding):
          torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

  def forward(self,id,targets=None):
      # calculating logits and losses
      # id is a tensor of shape (B,T)

      B,T = id.shape
      # batch size * block size

      tk = self.token_embedding(id)
      # B * T * embed_dim
      pos = self.position_embedding(torch.arange(T,device=device))
      # T * embed_dim
      x = tk + pos
      # broadcasting over two tensors
      x = self.blocks(x)
      # B * T * embed_dim
      x = self.final_norm(x)
      logits = self.feed_forw_1(x)
      # B * T * vocab_size


      if targets is None :
        losses = None
      else:
        B,T,C = logits.shape
        logits = logits.view(B*T,C)
        targets = targets.view(B*T)
        losses = F.cross_entropy(logits,targets)

      return logits,losses

  def generator(self,id,new_tokens):
```

```
    # here we are starting with a new id with dimension B*T

    for a in range(new_tokens):

      id_cond = id[:, -block_size:]

      logit,loss = self(id_cond)
      # logit is obtained for new id with shape (B,T,C)
      logit = logit[:,-1,:]
      # considered the last id from every batch hence shape (B,C)
      probs = F.softmax(logit,dim=1)
      # turning logits into probability
      newid = torch.multinomial(probs,num_samples=1)
      # sampling from multinomial distribution
      id = torch.cat((id,newid),dim=1)

    return id
```