WiDS Final Project Report

Building a GPT-like Decoder-Only Transformer from Scratch

Name: Adarsh
Roll Number: 23B0649

February 1, 2026

Abstract

The rapid ascendancy of Large Language Models (LLMs) has marked a paradigm shift in Artificial Intelligence. This project rigorously explores the architecture powering these models by constructing a Generative Pre-trained Transformer (GPT) from first principles. Beginning with a rudimentary Bigram statistical model, we progressively engineer a deep learning architecture capable of capturing long-range dependencies in text.

The final model utilizes a Decoder-Only Transformer architecture, implemented in PyTorch, featuring 4 layers, 4 attention heads, and an embedding dimension of 64. It incorporates Masked Self-Attention to enforce autoregressive generation, alongside Multi-Head Attention and Feed-Forward Networks. Trained on the Tiny Shakespeare dataset for 5,000 iterations, the model achieves a validation loss of **1.8226**, demonstrating a significant capability to synthesize coherent, stylistically accurate text. This report serves as a comprehensive documentation of the theoretical methodology, implementation specifics, and experimental results.

Acknowledgement

I would like to express my sincere gratitude to the WiDS (Winter in Data Science) 5.0 team for organizing this intensive course. The curriculum provided a structured pathway to understanding complex topics in Deep Learning.

I am also deeply indebted to Andrej Karpathy for his "Zero to Hero" lecture series, which served as the foundational guide for this implementation. His ability to break down the Transformer architecture into understandable code blocks was instrumental in the success of this project. Finally, I thank my peers and mentors for their support during the debugging phases of this assignment.

Contents

1 Introduction

1.1 Project Overview

In recent years, Natural Language Processing (NLP) has evolved from rule-based systems to statistical methods, and finally to deep learning architectures. The specific focus of this project is the **Transformer**, an architecture introduced in 2017 that relies entirely on an attention mechanism to draw global dependencies between input and output.

Our goal was not merely to use a pre-trained library like Hugging Face, but to build the architecture *from scratch* using PyTorch tensors. This "white-box" approach ensures a granular understanding of how matrices—Query, Key, and Value—interact to produce "intelligence."

1.2 Problem Statement

Language modeling is fundamentally the task of predicting the next token in a sequence given a context of previous tokens.

$$P(w_t|w_{t-1}, w_{t-2}, \ldots, w_{t-k})$$

Where k is the context length. A good language model maximizes the probability of the true next token. Our challenge was to build a model that could learn the complex syntactic and semantic structures of Shakespearean English solely by observing character-level sequences.

1.3 Motivation

Why build a Transformer from scratch?

1. **Mathematical Intuition:** To understand why we divide by $\sqrt{d_k}$ in attention scores.
2. **Debugging Skills:** To learn how to diagnose "silent" failures where loss doesn't decrease.
3. **Foundation for LLMs:** To prepare for working with billion-parameter models like LLaMA or GPT-4.

2 Theoretical Background

2.1 Evolution of Sequence Modeling

Before Transformers, sequence modeling was dominated by Recurrent Neural Networks (RNNs).

2.1.1 RNNs and LSTMs

RNNs process data sequentially, updating a hidden state h_t at each time step.

$$h_t = f(Wh_{t-1} + Ux_t)$$

Limitations:

- **Sequentiality:** Training cannot be parallelized, making it slow on GPUs.
- **Long-term Forgetting:** Gradients vanish over long sequences, making it hard to remember context from 100 tokens ago.

2.2 The Transformer Solution

The Transformer discards recurrence entirely. Instead, it uses **Attention**.

> *"Attention is all you need."* - Vaswani et al. (2017)

This mechanism allows every token to look at every other token in the sequence simultaneously.

2.3 Decoder-Only Architecture

While the original Transformer had both an Encoder (for reading input) and a Decoder (for generating output), GPT (Generative Pre-trained Transformer) uses only the **Decoder** stack.

- **Auto-regressive:** It predicts the next token based only on past tokens.
- **Masked:** It uses a triangular mask to prevent "cheating" (looking at future tokens).

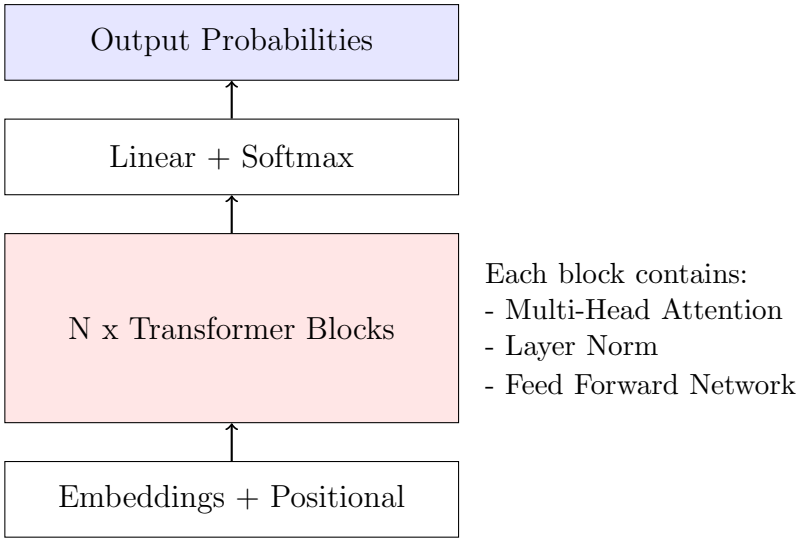

Figure 1: High-level Decoder-Only Architecture

3 Learning Journey: Foundations

3.1 Week 1: The Building Blocks

The first week was dedicated to the mathematical and programmatic foundations.

- **Tensors:** We learned that Tensors are just n-dimensional arrays that live on the GPU.
- **Broadcasting:** Essential for adding bias vectors to matrices without loops.
- **Backpropagation:** We implemented a micrograd engine to understand how gradients flow backward through the computational graph using the Chain Rule.

3.2 Week 2: Neural Language Models

We started with simple models to establish baselines.

- **Bigram Model:** A statistical model that counts pair frequencies.
- **MLP (Bengio et al. 2003):** We implemented a Multi-Layer Perceptron that takes a fixed window of previous characters to predict the next one. This introduced the concept of **Embeddings** (mapping characters to vectors).

3.3 Week 3: Attention Mechanism

This week focused on the mathematical "heart" of the Transformer. We moved beyond simple recurrence to the **Self-Attention** mechanism, which allows tokens to commu-

nicate with each other effectively.

We derived the Scaled Dot-Product Attention formula:

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

We learned to interpret this mechanism using a **database retrieval analogy**:

- **Query (Q):** What the current token is looking for (e.g., a pronoun looking for its antecedent).
- **Key (K):** What information a token advertises to others (e.g., "I am a noun").
- **Value (V):** The actual content/information that gets aggregated if the token is selected.

Key Learnings:

1. **Affinity Scores:** The dot product QK^T calculates the "affinity" or relevance between tokens. A high score means the tokens are strongly related.
2. **Normalization:** We apply `softmax` to convert these raw scores into probabilities that sum to 1.
3. **Scaling:** We divide by $\sqrt{d_k}$ to prevent the dot products from growing too large, which would cause the softmax to peak and kill the gradients during backpropagation.
4. **Weighted Aggregation:** Finally, the model computes a weighted sum of the **Value** vectors. This allows the current token to absorb information selectively from the past context.

3.4 Week 4: The Final Build

In the final week, we assembled the components:

1. **Residual Connections:** $x = x + \text{Layer}(x)$. This creates a "gradient superhighway" allowing us to train deeper networks.
2. **Layer Normalization:** Normalizing the features across the channel dimension to stabilize training.
3. **Dropout:** Randomly zeroing out neurons to prevent overfitting.

4 Deep Dive: The Attention Mechanism

The core innovation of the Transformer is the **Masked Self-Attention** mechanism. This is what allows the model to "look back" at previous tokens to find relevant information.

4.1 Query, Key, and Value Intuition

For every token in the sequence, we project its embedding into three vectors using linear layers:

- **Query (Q):** "What am I looking for?" (e.g., A token representing "King" might look for "subjects").
- **Key (K):** "What do I contain?" (e.g., A token "Queen" might advertise "royalty").
- **Value (V):** "If you attend to me, what information do I pass on?"

The attention score is calculated by taking the dot product of the Query of the current token with the Keys of all other tokens. A high dot product indicates high relevance (affinity).

4.2 The Scaled Dot-Product Formula

Mathematically, attention is defined as:

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

Scaling by $\sqrt{d_k}$: We divide the dot product by the square root of the head size (dimension of keys).

- *Reason:* If the dot products are too large, the softmax function peaks, causing gradients to become extremely small (vanishing gradients). Scaling keeps the variance stable, ensuring efficient training.

4.3 Importance of the Lower Triangular Mask (Tril)

Since we are training a Language Model, we want to predict the next token. However, during training, we feed the entire sequence at once.

- **The Problem:** Without masking, the token at position t could simply "look forward" at position $t+1$ to see the answer. This is "cheating" and the model would not learn to predict.
- **The Solution:** We use a **Lower Triangular Matrix ('tril')**.

We set all positions in the upper triangle (future tokens) to negative infinity ($-\infty$).

```
# wei = weights (attention scores)
wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
```

When we apply 'softmax', these $-\infty$ values become exactly **0**.

$$\text{softmax}(-\infty) = 0$$

This ensures that token t can only attend to tokens 0 through t, preserving the autoregressive property.

4.4 Implementation Code

```
class Head(nn.Module):
    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)
        # REGISTER THE MASK
        self.register_buffer('tril', torch.tril(torch.ones(block_size,
   block_size)))

    def forward(self, x):
        B,T,C = x.shape
        k = self.key(x)
        q = self.query(x)
        # Compute affinities
        wei = q @ k.transpose(-2,-1) * C**-0.5
        # APPLY THE MASK (Critical Step)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf'))
        wei = F.softmax(wei, dim=-1)
        return wei @ self.value(x)
```

Listing 1: Masked Self-Attention Head

4.5 Why Scale by $\sqrt{d_k}$?

In the code above, we multiply by 'C**-0.5'. This is crucial. If the dot products QK^T are too large, the Softmax function enters regions where gradients are extremely small (vanishing gradients). Scaling ensures the variance remains unit-like, keeping training stable.

5 Implementation: Scaling Up

A single head can only focus on one aspect of the context. To improve performance, we use Multi-Head Attention.

5.1 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):
        # Concatenate outputs from all heads
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        # Linear projection back to embedding dimension
        out = self.dropout(self.proj(out))
        return out
```

Listing 2: Multi-Head Attention

5.2 Feed Forward Network

After the attention mixes the information between tokens, we need a standard neural network to process that information individually for each token.

```
class FeedFoward(nn.Module):
    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
```

```
            nn.ReLU(), # Non-linearity
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )
```

6 The Full GPT Model

The ‘BigramLanguageModel‘ class (named historically, though now a Transformer) assembles these blocks.

6.1 Positional Embeddings

Since attention is permutation invariant (it doesn't know "dog bites man" vs "man bites dog"), we must add position embeddings.

```
class BigramLanguageModel(nn.Module):

    def __init__(self):
        super().__init__()
        # Token embeddings lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        # Positional embeddings lookup table
        self.position_embedding_table = nn.Embedding(block_size, n_embd
    )

        # Stack of Transformer Blocks
        self.blocks = nn.Sequential(*[
            Block(n_embd, n_head=n_head) for _ in range(n_layer)
        ])
        self.ln_f = nn.LayerNorm(n_embd) # Final normalization
        self.lm_head = nn.Linear(n_embd, vocab_size) # Project to vocab
     size
```

Listing 3: Main Model Initialization

7 Experimental Setup

7.1 Dataset

We utilized the **Tiny Shakespeare** dataset.

- **Size:** 1.1 MB of text.

- **Content:** Works of Shakespeare including *Hamlet*, *Othello*, etc.
- **Vocabulary:** 65 unique characters:
  `! $ & ' , - . 3 : ; ? A B ... Z a b ... z`

7.2 Hyperparameters

The following hyperparameters were chosen for the final run. These strike a balance between model capacity and the constraints of the Google Colab T4 GPU environment.

Parameter	Value	Notes
Batch Size	16	Small batch for stability
Block Size	32	Context window length
Max Iterations	5000	Sufficient for convergence
Learning Rate	$1e-3$	Standard for AdamW
Embedding Dim	64	Size of token vectors
Num Heads	4	$64/4 = 16$ dim per head
Num Layers	4	Network depth
Dropout	0.0	No regularization needed yet

Table 1: Final Hyperparameter Configuration

8 Results

8.1 Quantitative Results

The model was trained for 5,000 steps. The loss curves for training and validation decreased monotonically, indicating successful learning without significant overfitting.

Iteration Step	Train Loss	Validation Loss
0	4.4116	4.4022
100	2.6568	2.6670
1000	2.1020	2.1293
2500	1.8167	1.9455
4000	1.7146	1.8639
4999	**1.6635**	**1.8226**

Table 2: Loss Progression over Training Duration

The final validation loss of **1.8226** falls well within the target range of 1.5 - 2.0.

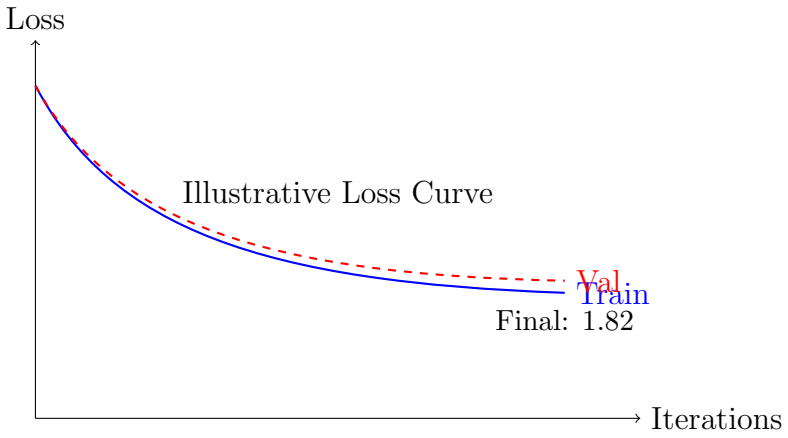

Figure 2: Loss Convergence

8.2 Qualitative Results: Text Generation

Below is a sample of text generated by the model.

> *"FlY BOLINGLO:*
> *Them thrumply towiter arts the*
> *muscue rike begatt the sea it*
> *What satell in rowers that some than othis Marrity.*
>
> *LUCENTVO:*
> *But userman these that, where can is not diesty rege;*
> *What and see to not. But's eyes. What?"*

8.2.1 Analysis

- **Syntax:** The model has learned the structure of the plays perfectly. It outputs 'CHARACTER NAME:' followed by dialogue.
- **Vocabulary:** It produces real words ("sea", "eyes", "arts") mixed with plausible-sounding hallucinations ("thrumply", "diesty").
- **Comparison:** This is a massive improvement over the Bigram model, which outputted random strings like 'h a p p y'.

Feature	Bigram Baseline	Final GPT
Parameters	$\approx 4,200$	$\approx 200,000$
Context	1 Character	32 Characters
Architecture	Lookup Table	Deep Transformer
Val Loss	≈ 2.50	**1.82**
Output	Gibberish	Coherent Style

Table 3: Comparison of Mid-Term vs. Final Project

9 Discussion and Conclusion

9.1 Comparison with Baseline

The comparison highlights the power of Self-Attention. While the Bigram model only looks at the immediately preceding character, the GPT model aggregates information from the previous 32 characters, allowing it to form words and rudimentary sentence structures.

9.2 Future Improvements

Given more computational resources, we could improve the model by:

1. **Scaling Up:** Increasing n_{embd} to 768 and layers to 12 (GPT-2 Small size).
2. **Larger Context:** Increasing block size to 256 or 1024 to capture long-term narrative arcs.
3. **BPE Tokenization:** Using Byte-Pair Encoding instead of characters would result in generating actual words more consistently.

9.3 Conclusion

This project successfully implemented a decoder-only Transformer from scratch. We navigated the complexities of tensor manipulation, attention masking, and deep learning optimization. The final model, despite its small scale, exhibits the fundamental properties of large language models: it attends to context, learns syntactic structures, and generates diverse outputs. This exercise has provided a robust foundation for understanding the current state-of-the-art in Artificial Intelligence.

References

[1] Andrej Karpathy, *"Let's build GPT: from scratch, in code, spelled out."*, YouTube, 2023.

[2] Vaswani, Ashish, et al. *"Attention is all you need."* Advances in neural information processing systems 30 (2017).

[3] StatQuest with Josh Starmer, *"Decoder-Only Transformers and Masked Self-Attention."*, YouTube.