From Language Models to Transformers: Deconstructing ChatGPT

1 Introduction: What is a Language Model?

At its core, ChatGPT is an advanced language model. A language model is a system that has been trained to understand the statistical patterns and structures of human language. Its fundamental task is sequence completion: given an initial piece of text (a prompt), it predicts the most likely continuation, generating text one piece at a time, from left to right.

This process is probabilistic, not deterministic. For the same prompt, the model can generate slightly different outputs, as it samples from a distribution of possible next words or characters. This is why you can get varied responses to identical queries.

Listing 1: Prompt and outputs

```
// Prompt: "Write a haiku about understanding AI
   ."

// Output 1:

AI knowledge

Brings prosperity for all.

Embrace its power.
```

```
// Output 2:

AI's power to grow,

Ignorance holds us back, learn.

Prosperity waits.
```

The model's capability extends far beyond simple poetry, enabling it to perform a vast range of text-based tasks, from explaining complex topics in simple terms to writing humorous articles, all by treating the task as a sequence completion problem.

2 The Core Technology: The Transformer

The neural network architecture that powers models like ChatGPT is the Transformer, introduced in the landmark 2017 paper, "Attention Is All You Need." The "T" in GPT stands for Transformer, with the full acronym being Generatively Pre-trained Transformer.

Interestingly, the original paper focused on machine translation and didn't fully anticipate that this architecture would become the foundation for nearly all state-of-the-art AI models in the subsequent years. The model we will build is a direct descendant of this work.

Our goal is not to replicate the massive, production-grade ChatGPT. Instead, we will build a "toy" model from scratch to understand the fundamental principles. We will train a character-level Transformer language model on the "Tiny Shakespeare" dataset—a 1MB text file containing the complete works of Shakespeare.

3 Step 1: Data Preparation and Tokenization

3.1 Building the Vocabulary & Encoder/Decoder

First, we identify every unique character in our text to form our vocabulary (65 unique characters for Tiny Shakespeare). Then, we create a mapping from each character to a unique integer and vice-versa.

Encoder: Converts a string to a list of integers.

Decoder: Converts a list of integers back to a string.

$$\texttt{'hello'} \longrightarrow [46, 43, 50, 50, 53] \longrightarrow \texttt{'hello'}$$

This character-level approach is simple but results in very long sequences. Large models like GPT use subword tokenization (like BPE) to create a larger vocabulary of word fragments, resulting in much shorter sequences.

3.2 Preparing Data Tensors and Splits

The tokenized text is converted into a single, long PyTorch Tensor. To monitor for overfitting (where the model memorizes the training data instead of learning general patterns), we split the data:

Training Set (90%): Used to train the model's parameters.

Validation Set (10%): Held back to evaluate how well the model generalizes to unseen data.

4 Step 2: Batches, Block Size, and Targets

We train the Transformer on small, random chunks of data.

Block Size (Context Length): The maximum number of preceding tokens the model can look at to predict the next one (e.g., block_size = 8).

Inputs (X) and Targets (Y): From a sequence of 9 characters, we can

extract 8 individual training examples.

```
Context (Input X)          Target (Y)

[18]                       47
[18, 47]                   56
...
[18, 47, 56, 57, 58, 1, 58, 59]    1
```

Batch Size: To efficiently use the GPU, we process multiple independent chunks in parallel. A batch is a stack of these chunks (e.g., a tensor of shape (4, 8) for a batch_size of 4).

5 A Simple Start: The Bigram Language Model

We begin with a Bigram Model, which predicts the next character based only on the current character. It uses a token embedding table (a lookup matrix) to get logits (raw scores) for the next character. A Cross-Entropy Loss function measures the error, and an optimizer (like AdamW) adjusts the model's parameters to minimize this loss. This model is a simple but necessary starting point before tackling token communication.

6 The Core of the Transformer: Self-Attention

Self-attention allows tokens to look at other tokens in the context and decide which ones are most important.

6.1 The Intuition: A Journey to Weighted Aggregation

How can tokens communicate? Let's build up the idea step-by-step, just as one would when designing the mechanism from scratch.

Version 1: The Naive for Loop. The simplest idea is for each

token to be the average of itself and all the tokens that came before it. We could write this with nested for loops that iterate through each token and calculate a running average. This works, but it's a "bag-of-words" approach that loses positional information and is incredibly inefficient in Python.

Version 2: The Matrix Multiplication Trick. We can achieve the same result far more efficiently. By creating a lower-triangular matrix of ones (using torch.tril) and normalizing its rows, we can perform a weighted average of all past tokens for every position simultaneously with a single matrix multiplication. This is a huge leap in efficiency and elegance, but it's still limited—every past token is weighted equally (e.g., 0.2, 0.2, 0.2, 0.2, 0.2).

Version 3: The General softmax Method. We need the weights to be data-dependent. A token might need to pay more attention to a verb three positions ago than a determiner one position ago. We achieve this by replacing our fixed averaging matrix with one whose values are learned. The final, powerful implementation is:

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

The scaling factor (d_k) is crucial for stabilizing training by controlling the variance of the dot products, preventing softmax from producing overly sharp distributions with poor gradients.

6.2 Scaled Dot-Product Attention: Keys, Queries, and the "Private vs. Public" Analogy

To make the aggregation data-dependent, we introduce three vectors for each token, derived from its initial embedding (which includes positional embeddings to give it a sense of order).

Query (Q): A vector representing what this token is looking for.

Key (K): A vector representing what this token contains or offers.

Value (V): A vector representing the information this token will pass on.

The dot product of a token's Query with another token's Key produces the attention score or affinity. But what is the Value vector for? This is where the brilliant "private vs. public information" analogy comes in:

You can think of a token's initial vector x (its token and positional embedding) as its private information—everything it knows about its identity and location. For the purpose of communicating in an attention head, it doesn't just broadcast all this private data. Instead, it creates a specific message. The Value vector, V, is that message. It's the token's public information, a curated piece of data it's willing to share. It's the token effectively saying, "Of all the things I know about myself, if you find me interesting (based on my Key matching your Query), this is the specific information I will communicate to you."

7 Building a Full Transformer Block

A complete Transformer is a stack of repeating blocks that intersperse this communication with computation.

7.1 Multi-Head Attention & Feed-Forward Networks

Multi-Head Attention: Instead of one large attention mechanism, we run multiple smaller "attention heads" in parallel and concatenate their results. This allows the model to simultaneously focus on different types of relationships (e.g., syntactic, semantic).

Feed-Forward Network (FFN): After attention (the communication step), each token needs to "think" about the information it just gathered. This is the computation step, performed by a small neural network applied to each token's vector independently.

7.2 Residual Connections and the "Aha!" Moment of Layer Norm

To train very deep networks effectively, two components are critical:

Residual Connections: We add the input of a layer to its output

$(x + \text{Layer}(x))$. This creates a "gradient superhighway" that dramatically improves training stability.

Layer Normalization: This stabilizes training by normalizing the features for each token. It's often compared to Batch Normalization, and the speaker provides a powerful "aha!" moment about their relationship:

Batch Norm normalizes across the batch dimension (i.e., it normalizes each column of a data batch to have zero mean and unit variance). The incredible insight is that Layer Norm does the exact same operation—mean-centering and scaling—but across the feature dimension (i.e., it normalizes each row). To turn a Batch Norm implementation into a Layer Norm implementation, you can often just change a single argument specifying the dimension of normalization (e.g., from dim=0 to dim=1 in PyTorch). This reveals they aren't alien concepts; they are two sides of the same normalization coin, applied along different axes of the data.

A single Transformer Block therefore consists of:

Layer Normalization → Multi-Head Self-Attention → Residual Connection

Layer Normalization → Feed-Forward Network → Residual Connection

8 Scaling Up and the Path to ChatGPT

By stacking these blocks and increasing hyperparameters (block_size, embedding_dimension, number_of_heads, etc.), we create a powerful model. Dropout is also added as a regularization technique to prevent overfitting.

8.1 The Full Picture: Encoder-Decoder vs. Decoder-Only

The original Transformer was an encoder-decoder model for machine translation. The encoder processes a source sentence (e.g., in French), and the decoder generates a target sentence (e.g., in English) while also paying cross-attention to the encoder's output.

GPT models are decoder-only. They don't have an encoder because

their task is simply to continue a given sequence, not translate from another.

8.2 From Pre-training to ChatGPT

Training ChatGPT is a two-stage process:

Pre-training: This is what we implemented, but at an astronomical scale (e.g., GPT-3 used 175 billion parameters trained on 300 billion tokens). The result is a powerful "document completer."

Fine-Tuning and Alignment: This crucial stage turns the base model into a helpful assistant using techniques like Reinforcement Learning from Human Feedback (RLHF), where the model is optimized to align with human preferences for helpfulness and safety.