Understanding Transformers through Implementation

Mehdi Ben Barka

June 20, 2025

Contents

Preface

It is clear that large language models are one of humanity's most powerful inventions—amplifying our cognitive abilities and opening up new frontiers in how we work, create, and learn. For anyone aspiring to enter the field of artificial intelligence, understanding how these models work is no longer optional—it's foundational.

I believe the best way to truly understand this technology is to build it. That's what motivated me to write this report: as a personal learning project, and as a resource to help others follow the same path. My goal is to offer a clear, hands-on walkthrough of the Transformer decoder architecture, enriched with many of the improvements that have emerged since the original "Attention is All You Need" paper by Vaswani *et al.* (2017).

This work builds on Andrej Karpathy's "Let's build GPT from scratch, in code, spelled out" video, but goes further by integrating modern techniques: RMSNorm, RoPE, SwiGLU, token-level regularization, cosine LR schedules, mixed-precision training, and more.

If you read this and have suggestions, corrections, or ideas for techniques I may have missed, I would be sincerely grateful for your feedback.

Also, please note that this report is the first version of a document I intend to update as I continue learning more about LLMs, as well as about writing papers and conducting research in the machine learning field.

1 Foundations

If you're already familiar with the Transformer architecture, you can skim this section as a quick review. However, if you're new to the topic, I highly recommend watching the 3 Blue 1 Brown series on Deep Learning before diving into the material that follows.

1.1 Decoder-Only Transformer Recap

The decoder-only Transformer architecture is a stack of identical layers, each composed of two main submodules:

- A multi-head masked self-attention mechanism, which allows each token to attend to all previous tokens.
- A position-wise feed-forward network (FFN) applied independently to each token position.

Each of these modules is wrapped in residual connections and preceded or followed by a form of normalization (e.g., LayerNorm or RMSNorm). Positional encodings are added to token embeddings at the input layer to inject order information.

1.2 Causal Self-Attention

In decoder-only models, self-attention is *causal*—meaning that each token can only attend to itself and previous tokens, never to future ones. This is implemented via a causal mask that prevents the model from accessing future information during training.

The scaled dot-product attention for a single head is computed as

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{1}$$

where

- Q, K, V are the query, key, and value matrices;
- d_k is the dimension of the keys.

1.3 Positional Encodings

Because Transformers are permutation-invariant by design, they require positional encodings to encode the order of tokens. There are multiple strategies:

- **Sinusoidal encodings**: fixed functions of token position;
- **Learnable embeddings**: trainable position vectors;
- **Rotary Positional Embeddings (RoPE)**: inject position via rotations in query/key space.

1.4 Feed-Forward Network (FFN)

In addition to self-attention, each Transformer block contains a position-wise feed-forward network (FFN) that independently transforms the representation at each token position. This sub-network allows the model to apply non-linear transformations and project token embeddings to a higher-dimensional space before bringing them back to the original size.

The FFN consists of two linear layers with a non-linearity in between:

$$\text{FFN}(x) = \text{Dropout}\big(W_2\,\phi(W_1 x)\big), \tag{2}$$

where

- W_1 expands the embedding dimension (typically by a factor of 4);
- ϕ is a non-linear activation such as ReLU or SwiGLU;
- W_2 projects the output back to the original embedding dimension;
- Dropout: is a regularization technique used during training to prevent overfitting. It works by randomly setting a fraction of the neurons in a layer to zero with a certain probability, called the *dropout rate*. This prevents the model from becoming overly dependent on any specific neurons and encourages it to learn more robust and generalized features.

1.5 Residual and Normalization Layers

Each submodule in the Transformer is wrapped with residual connections, which involve adding the input of a layer directly to its output. This approach allows the model to more easily propagate gradients during training, facilitating deeper networks and improving learning by preserving important features from the original input.

- The canonical Transformer uses post-layer normalization:

$$x \leftarrow x + \text{Sublayer}\big(\text{Norm}(x)\big), \tag{3}$$

- But many modern implementations use *pre-normalization* for better stability during training:

$$x \leftarrow \mathrm{Norm}(x) + \mathrm{Sublayer}\big(\mathrm{Norm}(x)\big). \tag{4}$$

2 Data & Tokenisation

2.1 Shakespeare Dataset

The "tiny–Shakespeare" corpus[1] is a compact text dataset containing approximately 1 MB of dialogue extracted from Shakespeare's plays. With its distinctive 16th-century style, rich punctuation, and compact size, it offers a perfect playground for training and debugging autoregressive models. Its limited vocabulary and manageable sequence lengths make it ideal for experimenting with architectural variants and optimization schedules without requiring large-scale compute.

2.2 Character vs BPE Tokenisation

Tokenisation refers to the process of dividing raw text into units that serve as input tokens for the model. Two common approaches are character-level and subword-level tokenisation:

Character Tokenisation. Each individual character—letter, digit, punctuation, or whitespace—is treated as its own token. This results in a small, fixed vocabulary (usually around 65–100 tokens) and eliminates the problem of out-of-vocabulary words. The main downside is inefficiency: sentences become long sequences of tokens, and the model must learn word structures and patterns from scratch. This is the method used in Karpathy's original implementation.

Byte–Pair Encoding (BPE). BPE tokenisation starts with characters and iteratively merges the most frequent adjacent pairs into subword units, producing vocabularies ranging from 10 000 to 50 000 tokens. This reduces sequence length and accelerates learning by capturing common patterns such as prefixes, suffixes, and frequent words. BPE is especially effective when training on diverse or morphologically rich corpora.

In this project, we implemented BPE tokenisation to improve training efficiency and better align with modern large language models. Compared to character-level tokenisation, BPE allows the model to process more information per forward pass, converge faster, and generalise more effectively to unseen text, particularly in settings where token length matters.

2.3 The `tiktoken` Package

To handle BPE tokenisation, we use `tiktoken`, a fast and memory-efficient tokenizer developed by OpenAI. It implements the same encoding scheme used in GPT-2 and GPT-3.

`tiktoken` also makes it easy to process large text files in streaming mode, which is particularly useful when working with custom corpora or limited RAM environments.

[1] Originally prepared by Andrej Karpathy for illustrating character-level RNNs.

3 Karpathy's methods and my additions

This project builds upon the clear and educational foundation laid out by Karpathy, while extending it with modern techniques that improve efficiency, stability, and performance. Below is a comparative overview of his original implementation and the enhancements introduced in this project.

Normalization: LayerNorm vs. RMSNorm

- **Karpathy:** Applies `LayerNorm` before each sub-layer (pre-normalization), which centers the activations and reduces their variance to stabilize training.

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sigma} \cdot \gamma + \beta \tag{5}$$

  Where:

  - x is the input vector,.
  - μ is the mean of x,
  - σ is the standard deviation of x,
  - γ is a learnable scaling parameter,
  - β is the bias term.

- **This work:** Replaces `LayerNorm` with `RMSNorm`, a lighter alternative that normalizes based on the root-mean-square (RMS) of the activations without centering them:

$$\text{RMSNorm}(x) = \frac{x}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \varepsilon}} \cdot \gamma \tag{6}$$

  - x is the input vector,
  - d is the dimension of the vector,
  - ε is a small constant added for numerical stability (to prevent division by zero),
  - γ is the learnable scaling parameter,

- **Benefits:** Fewer operations, and more stable convergence when strong regularization techniques are used.

Positional Encoding: Learnable Embeddings vs. RoPE

- **Karpathy:** Uses learned positional embeddings, assigning each absolute position a trainable vector.
- **This work:** Uses **Rotary Positional Embeddings (RoPE)** which are a type of positional encoding used in transformers, particularly designed to improve the model's handling of long sequences and to better encode the relative positions of tokens. RoPE works by directly incorporating positional information into the attention mechanism (rather than adding it separately to the input embeddings as in traditional positional encodings like sinusoidal or learned embeddings). It does this by applying trigonometric transformations to the queries (Q) and keys (K) in the attention mechanism.

- **Benefits:**
  - Generalizing better to longer sequences.
  - Implicitly modeling relative positions without the need for additional explicit encoding.
  - Better handling of causal attention, which is essential for autoregressive models like GPT.

Attention Mechanism: Baseline vs. Optimized

- **Karpathy:** Implements standard scaled dot-product self-attention with a causal mask recomputed at each forward pass.
- **This work:** Keeps the same formulation but stores the **causal mask once** via `register_buffer`, eliminating redundant computation in every forward call and marginally reducing memory traffic.

Feed-Forward Layer: ReLU vs. SwiGLU

- **Karpathy:** Uses a classic two-layer MLP with `ReLU` activation and a hidden dimension expanded by a factor of 4.

$$\mathrm{ReLU}(x) = \max(0, x) \tag{7}$$

- **This work:** Replaces the activation function with `SwiGLU`, where two separate linear transformations are applied to the input x:

$$\mathrm{SwiGLU}(a, b) = \mathrm{SiLU}(a) \cdot b \tag{8}$$

  Here:
  - $a = W_1 x + b_1$ is the output from the expansion layer, where W_1 has a larger number of units (increased dimension),
  - $b = W_2 x + b_2$ is the output from the reduction layer, where W_2 has a smaller number of units (reduced dimension).

  The SiLU activation function is applied to a as follows:

$$\mathrm{SiLU}(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}}, \tag{9}$$

  and then the result is multiplied element-wise by b to produce the final output.

- **Benefits:** Smoother gradients, higher expressiveness at constant parameter count, and improved convergence on mid-sized datasets. We reduce the expansion factor to $2\times$ instead of $4\times$ to lower computational cost without compromising performance.

Transformer Block Structure

- **Karpathy:** `LayerNorm` → Causal `Self-Attention` → MLP with `ReLU` → Residual connection.
- **This work:** `RMSNorm` → Causal `Self-Attention + RoPE` → `SwiGLU` → Residual connection + precomputed causal mask.

Embedding and Output Projection

- **Karpathy:** Maintains separate matrices for the input token embedding and the output linear head.
- **This work:** Employs *weight tying* by reusing the embedding matrix for the unembedding matrix (output projection).
- **Benefits:** Reduces the total parameter count, slightly improves perplexity (weight tying acts as a form of regularization), and simplifies checkpoint size.

Model Scale

- **Karpathy:** 6 layers, 384-dimensional embeddings, 6 attention heads, ~10.8 M parameters.
- **This work:** 3 layers, 256-dimensional embeddings, 4 attention heads. Despite the smaller hidden size, the parameter count is ~14.83 M because of a much larger BPE vocabulary (~50k tokens vs. ~65 characters).
- **Benefits:** Fewer layers reduce training time and memory footprint, while the larger vocabulary plus modern regularisation maintains comparable performance.

4 Training Pipeline

The training loop diverges from Karpathy's minimalist baseline by incorporating a set of modern engineering practices—each aimed at faster convergence, better generalisation, or higher computational efficiency. Table 1 contrasts the key choices.

Table 1: Baseline versus SOTA training pipeline.

Topic	Baseline (Karpathy)	This work
Optimiser	AdamW	AdamW + decoupled weight-decay (0.25)
LR schedule	Constant	Warm-up → cosine decay; `ReduceLROnPlateau` guard
Regularisation	Dropout 0.20	Dropout 0.35; label-smoothing 0.10; token-dropout 0.08
Precision	FP32	BF16 autocast & `GradScaler`
Gradient tricks	—	Grad-clip 1.0; `zero_grad(set_to_none=True)`

Optimizer:

- `AdamW`: Adam with weight decay, an optimizer that combines adaptive learning rates with regularization. **Weight decay** is a technique where a penalty proportional to the square of the model's weights is added to the loss function, helping to prevent the model from overfitting by discouraging large weights. This is implemented through the L2 regularization method.
- `AdamW + decoupled weight-decay (0.25)`: In this variant, weight decay is decoupled from the gradient updates, which leads to better generalization and reduces overfitting. The weight decay parameter is set to 0.25, meaning that a penalty is applied to large weights during training, encouraging the model to keep weights smaller.

Learning Rate (LR) Schedule:

- `Constant`: The learning rate remains fixed throughout the training.
- `Warm-up` → `cosine decay`:
  - `Warm-up`: The learning rate starts from a low value and increases gradually for the first few training steps to avoid large updates at the beginning.
  - `Cosine decay`: After warm-up, the learning rate decreases following a cosine function. **Cosine decay** gradually reduces the learning rate over time in a smooth and non-linear manner, which helps the model converge better and prevents overshooting during the final stages of training.
  - `ReduceLROnPlateau guard`: This method reduces the learning rate when the validation loss stops improving, ensuring better convergence by allowing more fine-tuning of the model in later stages.

5 Evaluation & Logging

Bits-per-Character vs. Perplexity

For character-level models the natural metric is *bits-per-character* (bpc), whereas sub-word or word models are typically reported in *perplexity* (PPL). Both derive from the cross-entropy (CE) loss:

$$\mathrm{PPL} = e^{\mathrm{CE}}, \tag{10}$$

$$\mathrm{bpc} = \frac{\mathrm{CE}}{\ln 2}. \tag{11}$$

Because our baseline uses character tokens and the upgraded model uses BPE, we convert cross-entropy to bpc for an apples-to-apples comparison. A bpc difference of 0.01 already reflects a noticeable change in sample quality for tiny-Shakespeare.

Summary of BPC Calculations:

- **Karpathy Model:**
  - **Step 0:** bpc ≈ 6.1
  - **Step 4999:** bpc ≈ 2.25
- **Your Model:**
  - **Step 0:** bpc ≈ 15.64
  - **Step 1599:** bpc ≈ 8.06

Training Time and BPC Comparison:

In **1600 steps and 11 minutes**, my model achieved a reduction of **7.58 points in bpc**. However, I had to stop at step 1600 to avoid overfitting. On the other hand, the **Karpathy model** required **5000 steps** to lower the bpc to **3.85**, demonstrating that my model converged much faster in terms of **bpc** reduction.

Both models were trained on a **T4 GPU from Kaggle**, and my model was trained in **11 minutes**, while the Karpathy model took **51 minutes** to complete the training process. This shows that my model was more efficient in terms of time and computational resources, achieving similar or better results in a shorter period.

Generation Comparison:

Here's a comparison of the text generated by both models:

My Model:

> Ay, sir, as gentlemen, Edward's Harm;; It is taken, ONTo France you good in this rudeons, Yet shall publications alone looks to see him to commend him Nicola to the? O's Wil Cube what she dardverts?
>
> CORIOLANUS: So, doth pure heart: And shall I are.
>
> ogleULIET: scill shall have my p subduits, entertain your would you to, That came your't.
>
> CLARURENCE: assaulting is little doubtful Chomsky such a gentleman to a train, [sons take fertility accident and in det fencedll, So I do I man can have next: , I will Icar and read of our nose 'tis. Love where sing rate his light as he came an dread May look in my live, my blood, nay, thy manner, I'll tell me that, Un brisk from my heart thrwans, here sit him, And when the petition of too plian man'st come: Here with Reflectness great aid

Karpathy's Model:

> But with prison, I will steal for the fimker.
>
> KING HENRY VI: To prevent it, as I love this country's cause.
>
> HENRY BOLINGBROOKE: I thank bhop my follow. Walk ye were so?
>
> NORTHUMBERLAND: My lord, I hearison! Who may love me accurse Some chold or flights then men shows to great the cur Ye cause who fled the trick that did princely action? Take my captiving sound, althoughts thy crown.

Future Work:

While my model showed promising results in terms of training efficiency and bpc reduction, the performance was somewhat limited due to the small size of the training dataset. The primary factor preventing the model from fully "shining" is the lack of diverse and large-scale data. To improve the model's generation quality, I intend to train it on a much larger dataset, with the goal of seeing if the model can produce better results with more extensive exposure to various linguistic patterns.

As a cinema fan, I plan to train the model on **Quentin Tarantino's** movie scripts. This decision stems from a few reasons:

- **Limited number of movies**: Tarantino has made only 10 movies, which makes it easier to collect a complete dataset.

- **Relatively high quality**: All of Tarantino's films are considered relatively good, providing high-quality text to learn from.
- **Distinct writing style**: Tarantino has a unique and easily identifiable writing style, which would help the model capture and replicate his voice more effectively.
- **Script ownership**: Tarantino writes all of his own scripts, ensuring consistency and a rich dataset for training the model.

The goal of training on Tarantino's scripts is to see if the model can generate a **complete opening scene** in his style, capturing the distinct elements of his writing, such as character dialogues, pacing, and tone. By training on this specific, high-quality dataset, I hope to push the boundaries of the model's ability to generate coherent and stylistically accurate text.