Building a tinyGPT from scratch

Composed by ZhenAn.

January 18, 2026

Contents

Chapter 1
Introduction to Language Model

1.1 The core idea of language modeling

At its core, a **Language Model (LM)** is a probabilistic distribution over sequences of tokens. The objective of language modeling is to estimate the joint probability distribution $P(\mathbf{w})$ over a sequence of discrete symbols $\mathbf{w} = (w_1, w_2, \ldots, w_T)$ drawn from a finite vocabulary $\mathcal{V}$. According to *the chain rule of probability*, any joint distribution can be factorized into a product of conditional probabilities:

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

In the context of generative modeling, this is referred to as Autoregressive Modeling. The model is tasked with predicting the probability of the next token w_t given the preceding sequence, known as the prefix or context $w_{<t}$.

> Given the vocabulary $\mathcal{V} = \{\text{The, a, cat, dog, sat, mat}\}$ and the sequence $\mathbf{w} = (\text{The, cat, sat})$, the joint probability is calculated as follows:

Let $w_1 = \text{The}$, $w_2 = \text{cat}$, and $w_3 = \text{sat}$. Applying the chain rule of probability:

$$P(w_1, w_2, w_3) = P(w_1) \cdot P(w_2 \mid w_1) \cdot P(w_3 \mid w_1, w_2)$$

Substituting the specific tokens from $\mathcal{V}$:

$$P(\text{The, cat, sat}) = P(\text{The}) \times P(\text{cat} \mid \text{The}) \times P(\text{sat} \mid \text{The, cat})$$

Each term represents the model's prediction at time step t given the preceding context $w_{<t}$. Based on the probability distribution presented in table 1.1, the model does not simply output a list of numbers; it must execute a decoding strategy to select the most appropriate discrete token for the sequence. The choice between candidates like "on" or "mat" depends on the specific objective of the generation:

+ **Greedy Decoding**: The model consistently selects the token with the highest probability:

  $$\hat{w}_t = \text{argmax}_{v \in \mathcal{V}} P(v \mid w_{<t})$$

  In this case, the model would select "on" ($P = 0.5012$) to continue the sequence as "The dog sat on..."

+ **Stochastic Sampling**: To introduce variance and avoid repetitive "robotic" text, the model may sample from the distribution.

  Strategies like top-k or top-p (Nucleus) sampling allow the model to occasionally choose lower-probability candidates like "mat" if they fall within a certain threshold.

Table 1.1: Probability distribution for w_4 given $w_{<4}$ = (The, dog, sat)

Index	**Token** (v)	**Logit** (z)	$P(w_4 = v \mid \textbf{prefix})$
1	on	4.12	0.5012
2	mat	3.89	0.3985
3	a	1.45	0.0347
4	the	1.18	0.0265
5	cat	0.82	0.0185
6	dog	0.54	0.0140
7	sat	-0.12	0.0066
Total			**1.0000**

1.2 From logits to softmax

The raw outputs from the final linear layer of a language model are known as logits ($\mathbf{z}$). These values are unbounded, ranging from $(-\infty, +\infty)$, making them unsuitable for direct probabilistic interpretation. To transform these scores into a valid probability distribution where each value $P_i \in [0, 1]$ and $\sum P = 1$, we apply the Softmax function.

For a vocabulary $\mathcal{V}$ of size K, the probability of the i-th token is defined as:

$$P(w_t = v_i \mid w_{<t}) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)}$$

The exponential function $\exp(\cdot)$ ensures that all resulting values are positive, while the denominator (the partition function) normalizes the vector.

Referring back to the empirical values in Table 1.1, we can observe how the model amplifies the gap between candidates. Even though the logit for "on" (4.12) and "mat" (3.89) are numerically close, the exponentiation creates a clear distinction in probability mass.

Derivation 1.4: Softmax Calculation

Let us calculate the probability for $v_1 =$ "on" and $v_2 =$ "mat" using their respective logits $z_1 = 4.12$ and $z_2 = 3.89$.
First, we compute the exponentials:

- $\exp(4.12) \approx 61.56$
- $\exp(3.89) \approx 48.91$

Assuming the sum of exponentials for the entire vocabulary $\sum \exp(z_j) \approx 122.82$, the resulting probabilities are:

$$P(\text{on}) = \frac{61.56}{122.82} \approx 0.5012, \quad P(\text{mat}) = \frac{48.91}{122.82} \approx 0.3982$$

This mapping demonstrates how the Softmax function acts as a "winner-take-all" mechanism, aggressively rewarding higher logits while suppressing lower scores.

Chapter 2
The evolution of language model

In modern artificial intelligence, while the underlying architectures have transitioned from simple statistical counts to massive neural networks, the fundamental objective remains rooted in the principles of sequence estimation, that is primarily defined by how the conditional distribution $P(w_t \mid w_{<t})$ is parameterized.

2.1 Statistical N-gram models

Early approaches relied on the k-th order Markov sssumption, which posits that the probability of a token depends only on a local window of the preceding $k-1$ tokens:

$$P(w_t \mid w_1, \ldots, w_{t-1}) \approx P(w_t \mid w_{t-k+1}, \ldots, w_{t-1})$$

2.1.1 Bi-gram models

In a Bigram approach, the Markov assumption simplifies the joint probability so that each token is conditioned only on the single immediate predecessor. The joint probability $P(\text{The, dog, sat})$ is factorized as:

$$P(\text{The, dog, sat}) \approx P(\text{The}) \times P(\text{dog} \mid \text{The}) \times P(\text{sat} \mid \text{dog})$$

The transition probabilities are calculated by counting occurrences within a training corpus. For instance, to calculate $P(\text{sat} \mid \text{dog})$, the model computes:

$$P(\text{sat} \mid \text{dog}) = \frac{\text{Count(dog, sat)}}{\text{Count(dog)}}$$

2.1.2 Tri-gram models

A Trigram model ($k = 3$) represents a significant step up in contextual awareness from the Bigram approach. By expanding the Markov window to include the two preceding tokens, the model begins to capture more complex linguistic structures, such as subject-verb-object relationships.In a Trigram model, the conditional probability is defined as:

$$P(w_t \mid w_1, \ldots, w_{t-1}) \approx P(w_t \mid w_{t-2}, w_{t-1})$$

For the sequence $\mathbf{w} = (\text{The, dog, sat})$, the joint probability is decomposed into a product of trigram components. Note that the first two tokens utilize lower-order models (Unigram and Bigram) to initialize the sequence:

$$P(\text{The, dog, sat}) \approx P(\text{The}) \times P(\text{dog} \mid \text{The}) \times P(\text{sat} \mid \text{The, dog})$$

2.1.3 N-gram models

The N-gram model is the generalized form of the Markov chain approach to language modeling. While Bi-grams and Tri-grams look at one or two preceding tokens, an N-gram model looks at $n-1$ previous tokens to predict the n-th token.

In an N-gram model, we assume that the probability of a word depends only on the $n-1$ words immediately preceding it. This is known as a Markov chain of order $n-1$. The conditional probability is defined as:

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

The joint probability of an entire sequence $\mathbf{w} = (w_1, \ldots, w_T)$ is therefore factorized as:

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

The transition probabilities are estimated using Maximum Likelihood Estimation (MLE) from a large corpus by calculating the ratio of sequence frequencies:

$$P(w_t \mid w_{t-n+1}, \ldots, w_{t-1}) = \frac{\text{Count}(w_{t-n+1}, \ldots, w_t)}{\text{Count}(w_{t-n+1}, \ldots, w_{t-1})}$$

2.2 Traditional deep learning approaches

Traditional deep learning for NLP shifted the focus from counting word frequencies to learning continuous vector representations. These models process tokens sequentially, maintaining an internal state that evolves over time.

2.2.1 Recurrent Neural Networks

The Recurrent Neural Network (RNN) represents a shift from fixed-window counting to a dynamic system capable of processing variable-length sequences. Unlike N-grams, which explicitly look back n steps, an RNN attempts to compress the entire history of a sequence into a single, fixed-size hidden vector $\mathbf{h}_t$.

Modeling conditional probabilities

From a probabilistic perspective, an RNN models the joint probability of a sequence by approximating the conditional probability of each token. It assumes that the probability of the current token w_t is conditioned on a latent representation of all previous tokens:

$$P(w_t \mid w_{t-1}, w_{t-2}, \ldots, w_1) \approx P(w_t \mid \mathbf{h}_{t-1})$$

where $\mathbf{h}_{t-1}$ is the hidden state summarizing the prefix $w_{<t}$.

The recursive update rule

At each time step t, the model performs a recursive transformation. It consumes the current input vector $\mathbf{x}_t$ (the embedding of w_t) and the previous hidden state $\mathbf{h}_{t-1}$ to compute the next state:

$$\mathbf{h}_t = \sigma(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h)$$

where:

- $\mathbf{W}_{hh}$ is the recurrent weight matrix (captures temporal dependencies).
- $\mathbf{W}_{xh}$ is the input weight matrix (captures current token features).
- σ is a non-linear activation function, typically tanh or ReLU.

The vanishing gradient problem

Despite their theoretical ability to maintain infinite memory, RNNs face severe practical limitations during training via **Backpropagation Through Time (BPTT)**. Because the same weight matrix $\mathbf{W}_{hh}$ is multiplied repeatedly at every step, the gradient of the loss with respect to early inputs involves a product of many Jacobian matrices.

If the eigenvalues of $\mathbf{W}_{hh}$ are small, the gradients decay exponentially ($0.9^{100} \approx 0$). This problem means the model effectively "forgets" distant context, reducing it to a short-term memory model that struggles with long-range dependencies, such as subject-verb agreement across long sentences.

2.2.2 Long-short Term Memory (LSTM)

While standard RNNs struggle to maintain context, the Long Short-Term Memory (LSTM) architecture preserves the conditional probability $P(w_t \mid w_{<t})$ by replacing simple nodes with sophisticated memory cells. The core innovation is the Cell State (c_t), a long-term highway that evolves via a selective additive update rather than forced multiplication.

By using a forget gate (f_t) to prune the past and an input gate (i_t) to scale new data, the cell performs the update $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$. This additive logic prevents the "condition" from decaying over time, allowing the model to ground its predictions in distant context that standard recurrent layers would otherwise lose.

The gating mechanism and update in cell

The core innovation of the LSTM is its use of gates, that are the neural layers that regulate the flow of information. These gates use a sigmoid activation (σ) to output values between 0 (completely close) and 1 (completely open).

1. **The forget gates**

   This gate looks at the previous hidden state and the current input to decide which information from the past is no longer relevant.

   $$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

2. **The input gate**

   This gate determines which new information from the current token is worth storing in our long-term memory.

   $$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

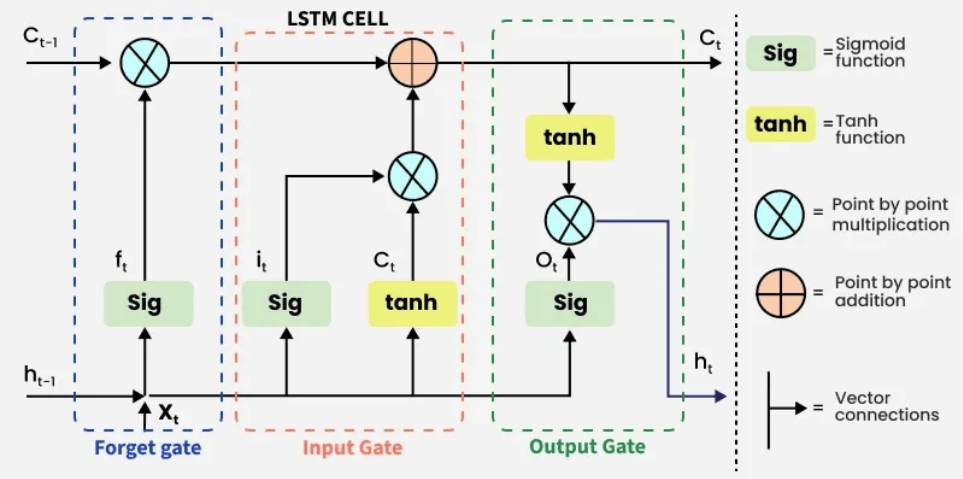

Figure 2.1: An LSTM cell architecture

3. **The output gate**

   Finally, this gate decides which part of the long-term cell state should be filtered out to become the hidden state (the "short-term memory") used for the current prediction.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

In conclusion, the interplay between these gating units allows the LSTM to overcome the vanishing gradient bottleneck inherent in standard recurrent architectures. By strategically utilizing the **forget gate** to discard obsolete information and the **input gate** to integrate novel features, the model maintains a stable and persistent **cell state** (c_t).

Solving the vanishing gradient

From a probabilistic and mathematical standpoint, the LSTM avoids vanishing gradients through its Cell State Update. Unlike the RNN's multiplicative update, the LSTM updates the cell state additively:

$$\mathbf{c}_t = f_t \odot \mathbf{c}_{t-1} + i_t \odot \tilde{\mathbf{c}}_t$$

Because the gradient can flow through this additive path without being repeatedly multiplied by a weight matrix, the LSTM can maintain a "gradient highway." This allows the model to learn that a subject at the start of a paragraph is still relevant to a verb twenty tokens later.

Chapter 3
Start building a tinyGPT

3.1 Model backbone

3.1.1 Causal self-attention

Linear projections and semantic mapping

The fundamental operation of the Transformer decoder involves mapping an input sequence $X \in \mathbb{R}^{T \times d_{\text{model}}}$ into three distinct functional representations. This is achieved through learned affine transformations parameterized by the weight matrices $W_Q, W_K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ and $W_V \in \mathbb{R}^{d_{\text{model}} \times d_v}$. The projections yield the Query (Q), Key (K), and Value (V) matrices:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V \tag{3.1}$$

These projections allow the model to decouple the specific role each token plays during the retrieval process: Q represents the current search criteria, K denotes the characteristics against which queries are matched, and V contains the substantive information to be propagated.

Variance stabilization and numerical stability

The raw affinity between tokens at positions i and j is quantified by the dot product $q_i \cdot k_j^T$. From a probabilistic perspective, consider the components of q_i and k_j as independent random variables with $\mathbb{E}[q_{im}] = \mathbb{E}[k_{jm}] = 0$ and $\text{Var}(q_{im}) = \text{Var}(k_{jm}) = 1$. The variance of their dot product is given by:

$$\text{Var}(q_i \cdot k_j) = \sum_{m=1}^{d_k} \text{Var}(q_{im} k_{jm}) = d_k \tag{3.2}$$

As the dimensionality d_k increases, the magnitude of these dot products grows, effectively pushing the input to the softmax function into regions of extreme saturation. In these regions, the gradient of the softmax function, $\nabla\sigma$, approaches zero, precipitating the vanishing gradient problem during backpropagation. To ensure the preservation of unit variance and maintain numerical stability, the scores are attenuated by a scaling factor of $1/\sqrt{d_k}$.

The attention operator

The final contextualized representation Z is computed by aggregating the value vectors according to the normalized attention weights. To maintain the autoregressive property required for generative tasks, a causal mask $\mathcal{M}$ is introduced to prevent information leakage from future tokens. The operator is defined as:

$$\text{Attention}(Q, K, V) = \sigma\left(\frac{QK^T + \mathcal{M}}{\sqrt{d_k}}\right)V \tag{3.3}$$

where $\sigma(\cdot)$ denotes the row-wise softmax activation:

$$\sigma(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{T}\exp(z_j)} \tag{3.4}$$

The mask $\mathcal{M} \in \{0, -\infty\}^{T\times T}$ is a lower-triangular matrix where $\mathcal{M}_{ij} = -\infty$ for all $j > i$. This additive identity ensures that the softmax distribution assigns zero probability to subsequent positions, strictly constraining the representation at index i to be a function of the prefix sequence $X_{\leq i}$.

3.1.2 Transformer block

The `Block` class serves as the fundamental unit of the GPT backbone, encapsulating two distinct operations: inter-token communication and individual token computation. By stacking these blocks, the model can iteratively refine its understanding of the sequence.

The Pre-LN architecture and the residual stream

In modern generative models, the arrangement of components follows the Pre-Layer Normalization (Pre-LN) convention. Unlike the original Transformer, which normalized signals after the residual addition, Pre-LN applies `LayerNorm` before each sub-layer.

This creates a "clean" residual stream (the identity path), allowing gradients to flow unimpeded from the output back to the input layers. The forward pass through a single block is defined by the following system of equations:

$$x_{mid} = x_{in} + \text{Attention}(\text{LN}_1(x_{in})) \tag{3.5}$$

$$x_{out} = x_{mid} + \text{MLP}(\text{LN}_2(x_{mid})) \tag{3.6}$$

By stabilizing the variance of activations before they enter the high-variance attention and feed-forward operations, Pre-LN facilitates the use of higher learning rates and significantly improves training stability in deep architectures.

Explain more detailed about LN, MLP & GELU

The internal logic of the Transformer block is governed by three critical components that manage stability and expressivity.

a) Layer normalization

To facilitate the training of deep architectures, Layer Normalization is utilized to standardize the distribution of hidden states. For a given input vector $\mathbf{x}$, the operation is defined as:

$$\text{LN}(x) = \gamma \odot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{3.7}$$

where μ and σ are the mean and standard deviation of the features, while γ and β are learnable affine parameters.

b) MLP with GELU activation

The MLP sub-layer provides the model's computational capacity. It utilizes a bottleneck structure to project tokens into a higher-dimensional manifold for feature extraction:

$$\text{MLP}(\mathbf{x}) = \text{GELU}(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2 \tag{3.8}$$

In this implementation, $\mathbf{W}_1 \in \mathbb{R}^{d_{model} \times 4d_{model}}$, providing a four-fold expansion of the latent space.

The **Gaussian Error Linear Unit (GELU)** introduces non-linearity by weighting the input by its cumulative distribution function:

$$\text{GELU}(x) = x \cdot \Phi(x) = \frac{x}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right] \tag{3.9}$$

The smoothness of GELU allows for more nuanced gradient updates compared to the piecewise-linear ReLU function.

3.1.3 The tinyGPT

The final architecture integrates the modular components into a cohesive autoregressive model. The process begins with the fusion of semantic and temporal information.

Positional Integration

Since the self-attention mechanism is permutation-invariant, we must inject positional information. We utilize a **Learned Positional Embedding** matrix $W_p \in \mathbb{R}^{T \times C}$. The input to the first Transformer block is defined as:

$$\mathbf{x}_0 = \text{Embedding}_{token}(idx) + \text{Embedding}_{pos}(pos) \tag{3.10}$$

The Transformer stack and head for each loop

The vector $\mathbf{x}_0$ propagates through a sequence of L blocks, where each block performs the *Attention* and *MLP* operations. The output of the final block, $\mathbf{x}_L$, represents a fully contextualized hidden state that captures the relationships between all preceding tokens.

To derive a prediction, the model must map this continuous vector back into the discrete vocabulary space. First, a final **LayerNorm** is applied to stabilize the cumulative variance from the stack. Then, the **Language Modeling Head** (a linear projection) computes the *Logits*:

$$\text{Logits} = \text{LayerNorm}(\mathbf{x}_L)\mathbf{W}_{vocab}^T \tag{3.11}$$

In an autoregressive loop, the token corresponding to the highest logit is appended to the input sequence. This updated sequence then serves as the input for the subsequent forward pass. This iterative cycle allows the model to generate text token-by-token, where each new prediction is conditioned on the entire history of previously generated tokens.

The inference mechanism for sequence generation

The generation of long-form text is achieved through an **autoregressive feedback loop**. This mechanism ensures that each generated token is conditioned on the entire preceding context.

Given an initial prompt $S = \{t_1, t_2, \ldots, t_n\}$, the model performs a forward pass to compute the probability distribution for the next token t_{n+1}:

$$P(t_{n+1}|t_1, \ldots, t_n) = \text{Softmax}(\text{tinyGPT}(S)) \tag{3.12}$$

Once t_{n+1} is sampled, it is appended to S, and the updated sequence $S' = \{t_1, \ldots, t_{n+1}\}$ is used for the subsequent iteration. This cycle continues until a predefined maximum length or an `<EOS>` token is reached.

Due to the quadratic complexity of the self-attention mechanism, the input sequence is constrained by the `block_size`. If the generated text exceeds this limit, the model utilizes a **sliding window approach**, retaining only the most recent T tokens. This ensures that the positional embeddings and attention masks remain consistent with the architecture's design limits.

3.2 Dataset

3.3 Training

3.3.1 Configuration

```
from dataclasses import dataclass

@dataclass
class GPTConfig:
    block_size: int = 256     # Maximum context length
    vocab_size: int = 50257   # Size of the dictionary
    n_layer: int = 12         # Number of Transformer blocks
    n_head: int = 12          # Number of attention heads
    n_embd: int = 768         # Embedding dimension

config = GPTConfig()
model = tinyGPT(config)

# 3. Prepare dummy input (B=4 batch size, T=8 sequence length)
# Example: "The dog sat on the mat" encoded as integers
idx = torch.randint(0, config.vocab_size, (4, 8))

# 4. Forward pass
logits = model(idx)
print(logits.shape) # Should output: torch.Size([4, 8, 50257])
```

3.3.2 Training

3.3.3 Evaluation

3.4 Model deployment