Building Large Language Models

From Scratch

A Comprehensive Technical Guide

Understanding Transformers, Attention Mechanisms, and Modern LLM Architecture

Author: Wreetojyoti Ray

Copyright Information

First Edition: February 2026

Table of Contents

Chapter 1: Introduction to Large Language Models

1.1 What Are Large Language Models?

Large Language Models (LLMs) are neural networks trained on vast amounts of text data to understand and generate human-like text. These models have revolutionized natural language processing by demonstrating remarkable capabilities in tasks ranging from translation and summarization to code generation and reasoning.

At their core, LLMs are statistical models that learn the probability distribution of language. Given a sequence of tokens (words or subwords), they predict the most likely next token. This seemingly simple objective, when scaled to billions of parameters and trained on trillions of tokens, enables emergent capabilities that weren't explicitly programmed.

1.2 Historical Context and Evolution

The journey to modern LLMs began with simple statistical language models like n-grams, which predicted the next word based on the previous n-1 words. These models were limited by their inability to capture long-range dependencies and their exponential growth in size with vocabulary.

Neural language models emerged in the early 2000s, using feedforward and recurrent neural networks. The breakthrough came with Long Short-Term Memory (LSTM) networks and Gated Recurrent Units (GRUs), which could better handle sequential dependencies. However, these models still struggled with very long sequences and parallelization during training.

The transformer architecture, introduced in the 2017 paper 'Attention Is All You Need' by Vaswani et al., fundamentally changed the landscape. By replacing recurrence with self-attention mechanisms, transformers could process sequences in parallel and capture dependencies regardless of distance. This led to the development of BERT, GPT, and eventually modern LLMs like GPT-3, GPT-4, and Claude.

1.3 Key Concepts and Terminology

1.3.1 Tokens and Vocabulary

Tokens are the basic units of text that language models process. Unlike words, tokens can be subwords, characters, or even bytes. Modern LLMs typically use subword tokenization algorithms like Byte-Pair Encoding (BPE) or SentencePiece, which balance vocabulary size with the ability to represent rare words and handle out-of-vocabulary terms.

1.3.2 Context Window

The context window is the maximum number of tokens a model can process at once. Early GPT models had context windows of 512-1024 tokens, while modern models like GPT-4 and Claude support 128,000 or more tokens. A larger context window allows the model to maintain coherence over longer texts and access more information when generating responses.

1.3.3 Parameters

Parameters are the learnable weights in a neural network. LLMs are called 'large' because they have billions or even trillions of parameters. GPT-3 has 175 billion parameters, while some models exceed 500 billion. More parameters generally allow the model to store more knowledge and capture more complex patterns, though they also require more compute and data to train effectively.

1.3.4 Pretraining and Fine-tuning

Pretraining is the initial training phase where a model learns from massive amounts of unlabeled text data, typically using a self-supervised objective like next-token prediction. Fine-tuning is the subsequent phase where the pretrained model is adapted to specific tasks using labeled data or reinforcement learning from human feedback (RLHF).

1.4 Applications and Capabilities

Modern LLMs demonstrate a wide range of capabilities that emerge from their training. These include natural language understanding and generation, translation between languages, summarization of long documents, question answering, code generation and debugging, mathematical reasoning, creative writing, and even multimodal understanding when combined with vision models.

Importantly, many of these capabilities emerge without explicit training. For example, GPT-3 could perform arithmetic despite not being explicitly trained on math problems. This emergent behavior is one of the most fascinating aspects of scaling language models and suggests that larger models develop more sophisticated internal representations of the world.

1.5 Challenges and Limitations

Despite their impressive capabilities, LLMs face several challenges. They can generate plausible-sounding but incorrect information (hallucinations), struggle with precise arithmetic and logical reasoning, may reflect biases present in their training data, have high computational costs for training and inference, and lack true understanding or grounding in the physical world.

Understanding these limitations is crucial when building and deploying LLMs. Throughout this guide, we'll explore not just how to build these models, but also how to address their shortcomings and improve their reliability.

Chapter 2: Understanding Text Data and Tokenization

2.1 From Text to Numbers

Neural networks operate on numerical data, not raw text. The process of converting text into numbers that neural networks can process is fundamental to building language models. This chapter explores the various approaches to text representation and why modern methods work best for LLMs.

2.2 Tokenization Approaches

2.2.1 Character-Level Tokenization

The simplest approach is to treat each character as a token. This has several advantages: the vocabulary is small (typically under 100 characters including uppercase, lowercase, digits, and punctuation), there are no out-of-vocabulary issues, and it requires minimal preprocessing.

However, character-level tokenization has significant drawbacks for LLMs. Sequences become very long, requiring more computation and memory. The model must learn to compose characters into meaningful units, which is inefficient compared to providing those units directly. Additionally, the model must learn spelling rules before it can learn higher-level language patterns.

2.2.2 Word-Level Tokenization

Word-level tokenization splits text on whitespace and punctuation, treating each word as a token. This is more efficient than character-level tokenization and aligns with how humans think about language. Sequences are shorter, and the model can immediately work with semantic units.

The main challenge is vocabulary size. English has hundreds of thousands of words, and a comprehensive vocabulary would be enormous. Common approaches limit the vocabulary to the most frequent words and mark rare words as 'unknown', but this loses information. Additionally, word-level tokenization struggles with morphological variants (run, running, ran) and completely fails with new or misspelled words.

2.2.3 Subword Tokenization

Subword tokenization is the modern solution that balances the advantages of character and word-level approaches. The key insight is that words can be decomposed into meaningful subunits. For example, 'unhappiness' might tokenize as ['un', 'happiness'] or ['un', 'happy', 'ness'], allowing the model to understand both the root word and its modifications.

This approach offers several benefits: vocabularies of 30,000-50,000 tokens can represent virtually any text, rare words are broken into common subwords rather than marked as unknown, morphological relationships are preserved, and different languages can share subword units in multilingual models.

2.3 Byte-Pair Encoding (BPE)

Byte-Pair Encoding is one of the most popular subword tokenization algorithms. Originally a data compression technique, BPE has become the de facto standard for many LLMs, including GPT-2,

GPT-3, and GPT-4.

2.3.1 The BPE Algorithm

BPE works by iteratively merging the most frequent pair of characters or character sequences. The algorithm starts with a vocabulary of individual characters. It then repeatedly finds the most frequent adjacent pair of symbols in the training corpus and merges them into a single symbol, adding this merged symbol to the vocabulary. This process continues for a predefined number of iterations or until the vocabulary reaches a target size.

Example BPE Process:

```
Starting corpus: low low low low lower lower newest newest newest widest widest

After character splitting: l o w _ l o w _ l o w _ l o w _ l o w e r _ l o w e r _
...

Most frequent pair: ('l', 'o') -> Merge into 'lo'

Next iteration: ('lo', 'w') -> Merge into 'low'

Continuing until vocabulary size is reached...
```

The beauty of BPE is that it automatically discovers common subwords in the training data. Frequent words are kept whole, while rare words are broken into meaningful parts. The algorithm is deterministic and reversible - you can always decode tokens back to the original text.

2.3.2 Implementing BPE

A basic BPE implementation requires two components: training the merge rules and applying them to tokenize new text. During training, we count all adjacent pairs in the corpus, find the most frequent pair, merge it everywhere in the corpus, add the merged pair to our vocabulary, and repeat until we reach the desired vocabulary size.

To tokenize new text, we split it into characters, then iteratively apply the learned merge rules in the order they were learned. The result is a sequence of tokens that can be converted to integer IDs for input to the neural network.

2.4 Special Tokens

Beyond regular text tokens, LLMs use special tokens to mark boundaries and convey structural information. Common special tokens include:

```
<BOS> or <s>: Beginning of sequence, marks the start of a text <EOS> or </s>: End
of sequence, signals the model to stop generating <PAD>: Padding token, used to
make sequences the same length in batches <UNK>: Unknown token, for rare or
out-of-vocabulary terms <MASK>: Mask token, used in models like BERT for masked
language modeling
```

These special tokens are added to the vocabulary and have their own learned embeddings. They help the model understand text structure and control generation behavior.

2.5 Vocabulary Size Considerations

Choosing the right vocabulary size involves tradeoffs. A larger vocabulary means shorter sequences, faster processing, and better representation of common words. However, it also means a larger embedding matrix, more parameters to train, and potentially worse representations for rare tokens due

to less frequent updates.

Most modern LLMs use vocabularies between 32,000 and 100,000 tokens. GPT-2 uses 50,257 tokens, while GPT-3 and GPT-4 use around 100,000. The optimal size depends on the languages supported, domain specificity, and computational constraints.

2.6 Practical Tokenization Pipeline

A complete tokenization pipeline involves several steps beyond just applying BPE. First, text is normalized by converting to lowercase (optional), handling Unicode normalization, and managing whitespace. Then, pre-tokenization splits the text on whitespace and punctuation boundaries. BPE is applied to each pre-token independently. Finally, tokens are mapped to integer IDs, and special tokens are added as needed.

Modern libraries like HuggingFace Tokenizers or SentencePiece handle these details efficiently, providing both training and inference capabilities. They also support advanced features like byte-level BPE (which handles any Unicode character) and dynamic token dropout for regularization.

Chapter 3: Neural Network Fundamentals

3.1 Building Blocks of Neural Networks

Before diving into transformers and LLMs, we need to understand the fundamental components that make up neural networks. These building blocks - neurons, layers, activations, and optimizers - form the foundation of all deep learning models.

3.2 Embeddings: From Tokens to Vectors

After tokenization, we have sequences of integer IDs. Neural networks require dense, continuous representations, which we obtain through embeddings. An embedding is a learned mapping from discrete tokens to continuous vectors in a high-dimensional space.

3.2.1 Embedding Layer

An embedding layer is essentially a lookup table. For a vocabulary of size V and embedding dimension d, we have a matrix of shape (V, d). Each row corresponds to one token in the vocabulary. To embed a token with ID i, we simply retrieve the i-th row of this matrix.

For example, with vocabulary size 50,000 and embedding dimension 768, our embedding matrix has 38.4 million parameters. This matrix is learned during training, starting from random initialization. Tokens that appear in similar contexts will have similar embedding vectors, encoding semantic relationships.

3.2.2 Positional Embeddings

Unlike recurrent networks, transformers process all tokens in parallel and have no inherent notion of sequence order. To inject positional information, we add positional embeddings to token embeddings. There are two main approaches: learned positional embeddings and sinusoidal positional encodings.

Learned positional embeddings are simply another embedding matrix, this time of shape (max_seq_len, d), where max_seq_len is the maximum sequence length the model can handle. For position p, we add the p-th row to the token embedding.

Sinusoidal encodings use sine and cosine functions at different frequencies to encode positions. For position pos and dimension i, the encoding is: sin(pos / 10000^(2i/d)) for even dimensions and cos(pos / 10000^(2i/d)) for odd dimensions. This approach allows the model to extrapolate to longer sequences than seen during training.

3.3 Layer Normalization

Layer normalization is critical for training deep transformers. It normalizes activations across the feature dimension for each example in a batch, stabilizing training and allowing deeper models.

Given an input x of shape (batch_size, seq_len, d_model), layer normalization computes the mean and variance across the d_model dimension for each position in each example. It then normalizes: x_norm = (x - mean) / sqrt(variance + epsilon), where epsilon is a small constant for numerical stability. Finally, it applies learned scale and shift parameters: output = gamma * x_norm + beta.

Unlike batch normalization, layer normalization is independent across examples in a batch. This makes it suitable for sequence models where batch items may have different lengths and the statistics should not mix across examples.

3.4 Feed-Forward Networks

Each transformer layer includes a position-wise feed-forward network. This is a simple two-layer neural network applied independently to each position. Despite its simplicity, it contains most of the parameters in a transformer model.

3.4.1 Architecture

The feed-forward network consists of two linear transformations with an activation function between them. If the model dimension is d_model and the inner dimension is d_ff (typically 4 * d_model), the transformations are: FFN(x) = activation(x * W1 + b1) * W2 + b2, where W1 has shape (d_model, d_ff), W2 has shape (d_ff, d_model), and activation is typically GELU or ReLU.

This expands the representation to a higher dimension (d_ff), applies non-linearity, then projects back to the model dimension. The expansion ratio of 4x is a hyperparameter that can be tuned, though 4 has proven effective across many models.

3.4.2 Activation Functions

The activation function introduces non-linearity, essential for learning complex patterns. The Gaussian Error Linear Unit (GELU) is most common in modern LLMs. GELU is defined as: GELU(x) = x * Φ(x), where Φ(x) is the cumulative distribution function of the standard Gaussian distribution.

GELU provides a smooth, non-monotonic function that allows both positive and negative values to pass through, with the amount modulated by the input magnitude. This has proven more effective than older activation functions like ReLU for language modeling.

3.5 Residual Connections and Layer Normalization Placement

Residual connections (skip connections) are crucial for training deep networks. They provide a direct path for gradients to flow backward, mitigating the vanishing gradient problem. In transformers, every sub-layer (attention and feed-forward) is wrapped with a residual connection.

The original transformer used post-layer normalization: x = LayerNorm(x + Sublayer(x)). However, modern LLMs typically use pre-layer normalization: x = x + Sublayer(LayerNorm(x)). This configuration is more stable for training very deep networks and allows training without learning rate warmup.

3.6 Dropout for Regularization

Dropout randomly sets a fraction of activations to zero during training, preventing overfitting by forcing the network to learn redundant representations. In transformers, dropout is applied after attention weights, after feed-forward layers, and to embeddings.

The dropout rate is a hyperparameter, typically 0.1 (10%) for large models. Higher rates provide more regularization but can hurt performance if too high. Dropout is only active during training; during inference, all neurons are active but their outputs are scaled by (1 - dropout_rate) to maintain expected magnitudes.

3.7 Weight Initialization

Proper weight initialization is critical for training deep networks. Poor initialization can lead to vanishing or exploding gradients, making training unstable or impossible. For transformers, we typically use Xavier/Glorot initialization or a scaled variant.

Xavier initialization draws weights from a distribution with variance 2/(fan_in + fan_out), where fan_in and fan_out are the number of input and output units. This keeps the variance of activations roughly constant across layers. For deep transformers, weights may be additionally scaled by 1/sqrt(num_layers) to account for the accumulation of residual branches.

Chapter 4: The Transformer Architecture

4.1 Overview of the Transformer

The transformer architecture, introduced in 'Attention Is All You Need', fundamentally changed how we process sequential data. Unlike recurrent networks that process tokens one at a time, transformers process entire sequences in parallel using self-attention mechanisms. This enables much faster training and better capture of long-range dependencies.

The original transformer was designed for sequence-to-sequence tasks like machine translation and consisted of an encoder and decoder. Modern LLMs like GPT use only the decoder portion, configured as an autoregressive language model. We'll focus on this decoder-only architecture as it's the foundation of most current LLMs.

4.2 High-Level Architecture

A decoder-only transformer consists of a stack of identical layers. Each layer has two main components: a multi-head self-attention mechanism and a position-wise feed-forward network. Both components are wrapped with residual connections and layer normalization.

The input sequence flows through the following stages: First, tokens are embedded and positional information is added. Then, the combined embeddings pass through N transformer layers (typically 12-96 layers for modern LLMs). Each layer transforms the representation while preserving the sequence length. Finally, a language modeling head (linear layer) projects from the model dimension to vocabulary size, producing logits for next token prediction.

4.3 Input Processing

The first step is converting input tokens into dense vectors. For a batch of sequences with shape (batch_size, seq_len), we perform token embedding to get shape (batch_size, seq_len, d_model), add positional embeddings with the same shape, and apply dropout for regularization.

The model dimension d_model is a key hyperparameter. GPT-2 small uses 768, GPT-2 medium uses 1024, GPT-2 large uses 1280, and GPT-2 XL uses 1600. Larger models like GPT-3 use dimensions up to 12,288. This dimension remains constant throughout all transformer layers.

4.4 The Transformer Layer

Each transformer layer applies the following operations in sequence. First comes pre-layer normalization of the input. Then multi-head self-attention processes the normalized input. The attention output is added to the input via a residual connection. Next comes another pre-layer normalization. Then the position-wise feed-forward network processes the normalized input. Finally, another residual connection adds the FFN output to its input.

This can be expressed mathematically as: x1 = x + Attention(LayerNorm(x)) and x2 = x1 + FFN(LayerNorm(x1)), where x is the input to the layer and x2 is the output.

4.5 Stacking Layers

Modern LLMs stack many transformer layers to build deep networks. The number of layers is another critical hyperparameter. GPT-2 variants use 12, 24, 36, or 48 layers. GPT-3 uses 96 layers. Very large models may have over 100 layers.

Deeper networks can learn more complex representations and capture more abstract patterns. However, they also require more careful training, including proper initialization, normalization, and gradient clipping. The residual connections are essential for training these deep models, providing direct paths for gradient flow.

4.6 Output and Language Modeling Head

After passing through all transformer layers, we have representations of shape (batch_size, seq_len, d_model). To predict the next token, we apply a final layer normalization, then a linear projection to vocabulary size: logits = LayerNorm(x) * W_lm + b_lm, where W_lm has shape (d_model, vocab_size).

These logits represent unnormalized log probabilities for each token in the vocabulary at each position. We typically apply softmax to convert them to probabilities: P(token) = softmax(logits). During training, we compute cross-entropy loss against the true next tokens. During inference, we sample from these probabilities or take the argmax for greedy decoding.

4.7 Causal Masking

For autoregressive language modeling, each position should only attend to previous positions, not future ones. This is enforced through causal masking (also called attention masking). We create a mask matrix where position i can attend to positions 0 through i, but not i+1 onwards.

The mask is typically implemented by adding negative infinity (-inf) to attention scores for disallowed positions before the softmax. After softmax, these positions have zero attention weight, effectively preventing information flow from future tokens. This ensures the model can be used autoregressively at inference time, generating one token at a time based only on previous tokens.

4.8 Model Size Scaling

Transformer models scale along several dimensions. The model dimension (d_model) controls the representational capacity of each position. The number of layers (depth) allows learning more abstract and compositional features. The number of attention heads enables attending to different aspects simultaneously. The feed-forward dimension (d_ff) provides additional capacity for non-linear transformations.

Research has shown that scaling these dimensions, along with data and compute, leads to consistent improvements in performance. The scaling laws describe predictable relationships between model size, data size, compute budget, and final performance. This has driven the trend toward ever-larger models.

Chapter 5: Attention Mechanisms Deep Dive

5.1 The Concept of Attention

Attention mechanisms allow models to focus on relevant parts of the input when processing each position. The core idea is to compute a weighted sum of values based on the compatibility between a query and keys. This enables flexible, content-based information flow rather than fixed patterns.

In the context of language modeling, when processing each token, the model attends to all previous tokens (in autoregressive models) or all tokens in the sequence (in bidirectional models). The attention weights determine how much each token contributes to the current token's representation.

5.2 Scaled Dot-Product Attention

The fundamental attention operation in transformers is scaled dot-product attention. Given queries Q, keys K, and values V, all of shape (batch_size, seq_len, d_k), attention is computed as: Attention(Q, K, V) = softmax(Q * K^T / sqrt(d_k)) * V.

5.2.1 Breaking Down the Formula

Let's understand each component. Q * K^T computes dot products between all query-key pairs, giving a score matrix of shape (batch_size, seq_len, seq_len). Each entry (i,j) represents how much position i should attend to position j. Division by sqrt(d_k) scales the dot products. Without this scaling, for large d_k, the dot products can become very large, pushing softmax into regions with extremely small gradients. The sqrt(d_k) factor keeps the variance of dot products roughly constant regardless of dimension.

Softmax normalizes each row of the score matrix into a probability distribution. For each query position i, we get probabilities summing to 1 across all key positions. Finally, multiplication by V computes weighted sums of values according to the attention probabilities. Each output position is a weighted average of all value vectors.

5.2.2 Why Dot Product?

The dot product measures similarity between vectors. If a query and key point in similar directions (high dot product), they're considered related and the corresponding value receives high weight. This is efficient to compute using matrix multiplication and naturally captures semantic similarity when queries and keys are learned representations.

Alternative attention mechanisms exist, such as additive attention which uses a feed-forward network to compute compatibility. However, dot-product attention is faster and more space-efficient in practice, making it the standard choice for large models.

5.3 Multi-Head Attention

Single attention might focus on one type of relationship. Multi-head attention runs multiple attention operations in parallel, each with different learned projections. This allows the model to jointly attend to information from different representation subspaces at different positions.

5.3.1 How Multi-Head Attention Works

Given input X of shape (batch_size, seq_len, d_model), we first project it to queries, keys, and values for each head. For h heads and model dimension d_model, each head uses dimension d_k = d_model / h. We have learned weight matrices W_Q^i, W_K^i, W_V^i for each head i, all with shape (d_model, d_k).

For each head i, we compute Q_i = X * W_Q^i, K_i = X * W_K^i, and V_i = X * W_V^i. Then we apply scaled dot-product attention: head_i = Attention(Q_i, K_i, V_i). All heads are concatenated and projected: MultiHead(X) = Concat(head_1, ..., head_h) * W_O, where W_O has shape (d_model, d_model).

5.3.2 Benefits of Multiple Heads

Different heads can specialize in different relationships. For example, one head might focus on syntactic dependencies, another on coreference, and another on semantic similarity. Empirically, models with multiple heads consistently outperform single-head models with the same total parameters.

The number of heads is a hyperparameter. GPT-2 small uses 12 heads with 64 dimensions each (12 * 64 = 768). GPT-3 uses 96 heads with 128 dimensions each (96 * 128 = 12,288). More heads allow more specialized attention patterns but reduce the dimension per head, which can limit capacity.

5.4 Self-Attention vs Cross-Attention

In self-attention, queries, keys, and values all come from the same source (the layer's input). Each position attends to all positions in the same sequence. This is what decoder-only models like GPT use.

Cross-attention is used in encoder-decoder models. Queries come from the decoder, while keys and values come from the encoder. This allows the decoder to attend to the encoder's representations. While not used in GPT-style models, cross-attention is important in models like T5 and in vision-language models.

5.5 Attention Patterns and Interpretability

Attention weights can be visualized to understand what the model is focusing on. In practice, different heads learn different patterns. Some heads show clear linguistic behavior: attending to previous tokens, to delimiter tokens, to tokens of the same entity, or forming specific syntactic structures.

However, attention is not a perfect interpretability tool. High attention weight doesn't necessarily mean high importance for the final prediction. The value vectors also matter, and information can flow through the network in complex ways. Still, attention patterns provide useful insights into model behavior.

5.6 Efficient Attention Variants

Standard attention has quadratic complexity O(n^2) in sequence length n, as every position attends to every other position. For very long sequences (thousands to millions of tokens), this becomes prohibitive. Many efficient attention variants have been proposed to reduce this complexity.

5.6.1 Sparse Attention

Sparse attention restricts which positions can attend to each other, reducing complexity to O(n * sqrt(n)) or O(n * log(n)). For example, each position might attend to nearby positions (local attention) and every k-th position (strided attention). Combining these patterns can capture both local and global dependencies while being more efficient.

5.6.2 Linear Attention

Linear attention methods reformulate attention to avoid explicitly computing the full attention matrix. By approximating softmax or using kernel methods, they achieve O(n) complexity. However, these methods often sacrifice some modeling capacity compared to full attention.

5.6.3 Flash Attention

Flash Attention and its successors optimize attention computation for modern GPUs by being more memory-efficient and reducing memory reads/writes. While still O(n^2), Flash Attention can train with much longer sequences within the same memory budget and runs significantly faster. It has become the standard implementation for many large models.

5.7 Positional Information in Attention

Attention is permutation-invariant - swapping the order of inputs produces the same output (aside from the corresponding swap in output positions). This is why positional information must be added separately through positional embeddings or encodings.

Recent work explores incorporating position directly into attention through mechanisms like relative positional encodings. These modify the attention computation to be aware of the relative distance between positions, potentially improving the model's ability to handle position-sensitive tasks.

Chapter 6: Building GPT from Scratch

6.1 GPT Architecture Overview

GPT (Generative Pre-trained Transformer) is a decoder-only transformer architecture designed for autoregressive language modeling. Unlike BERT which uses bidirectional attention, GPT uses causal attention where each position can only attend to previous positions. This makes it naturally suited for text generation.

The GPT architecture consists of token and positional embeddings, N transformer layers with causal self-attention, and a language modeling head that predicts the next token. The model is trained to predict the next token given all previous tokens, learning to model the probability distribution of language.

6.2 Implementation Components

6.2.1 Embedding Layer Implementation

The embedding layer converts token IDs to dense vectors. We need two embedding matrices: one for tokens (vocab_size, d_model) and one for positions (max_seq_len, d_model). In PyTorch, this can be implemented using nn.Embedding modules which are essentially trainable lookup tables.

The token embeddings are initialized randomly, typically from a normal distribution with mean 0 and standard deviation 0.02. Position embeddings can be either learned (also randomly initialized) or fixed sinusoidal encodings. The final embedding is the sum of token and position embeddings.

6.2.2 Attention Implementation

Implementing multi-head attention requires careful handling of dimensions. Given input x of shape (batch_size, seq_len, d_model), we project to queries, keys, and values using linear layers. These are then reshaped to (batch_size, num_heads, seq_len, head_dim) where head_dim = d_model / num_heads.

The attention scores are computed as Q * K^T / sqrt(head_dim). For causal attention, we apply a mask to prevent attending to future positions. This is typically done by creating a lower triangular matrix of ones and adding negative infinity to positions above the diagonal. After softmax, these positions have zero attention weight.

The attention weights are multiplied by values, and the result is reshaped back to (batch_size, seq_len, d_model) by concatenating heads. Finally, an output projection is applied to mix information across heads.

6.2.3 Feed-Forward Network Implementation

The feed-forward network is straightforward: a linear layer expanding from d_model to d_ff (typically 4 * d_model), a GELU activation, and a linear layer projecting back to d_model. Dropout can be applied after the activation and after the final projection for regularization.

6.2.4 Transformer Block Implementation

A complete transformer block combines attention and feed-forward with layer normalization and residual connections. In code, this looks like: x = x + attention(layer_norm(x)) followed by x = x + ffn(layer_norm(x)). This simple structure is repeated N times to create the full model.

6.3 Language Modeling Head

The final component is the language modeling head that converts transformer outputs to vocabulary predictions. This is typically a linear layer from d_model to vocab_size, often with tied weights - the same weight matrix used for both input token embeddings and output predictions. Weight tying reduces parameters and can improve performance.

During training, we compute cross-entropy loss between predicted logits and true next tokens. The loss is averaged over all positions and all examples in the batch. During inference, we can either greedily select the highest-probability token or sample from the probability distribution for more diverse outputs.

6.4 Complete Model Architecture

Putting it all together, a GPT model has the following flow: input token IDs are embedded and added to positional embeddings. Dropout is applied to the sum. The embeddings pass through N transformer blocks, each applying causal self-attention and feed-forward with residual connections and layer normalization. A final layer normalization is applied. The language modeling head projects to vocabulary size to produce logits.

6.5 Model Configurations

Different GPT model sizes use different configurations. GPT-2 small has 12 layers, 768 hidden dimensions, 12 attention heads, and 117M parameters. GPT-2 medium has 24 layers, 1024 dimensions, 16 heads, and 345M parameters. GPT-2 large has 36 layers, 1280 dimensions, 20 heads, and 762M parameters. GPT-2 XL has 48 layers, 1600 dimensions, 25 heads, and 1.5B parameters.

GPT-3 dramatically scaled up with 96 layers, 12,288 dimensions, 96 heads, and 175B parameters. The architectural changes were minimal - mainly just scaling existing dimensions. This demonstrates that the transformer architecture is highly scalable.

6.6 Implementation Best Practices

When implementing GPT, several practices improve training stability and performance. Use pre-layer normalization rather than post-layer normalization for deeper models. Initialize weights carefully, with smaller initialization for deeper networks. Apply gradient clipping to prevent exploding gradients. Use mixed precision training (FP16 or BF16) to reduce memory and increase speed. Implement efficient attention using Flash Attention or similar optimizations.

Also consider implementation details like: using learned positional embeddings up to max sequence length, applying dropout to embeddings, attention, and feed-forward layers, tying token embedding and language modeling head weights, using GELU activation in feed-forward networks, and implementing causal masking efficiently to avoid recomputation.

Chapter 7: Training Large Language Models

7.1 The Training Objective

LLMs are trained using next-token prediction, also called autoregressive language modeling. Given a sequence of tokens x_1, x_2, ..., x_n, the model learns to predict each token x_i given all previous tokens x_1, ..., x_(i-1). The training objective is to maximize the log-likelihood of the training data.

Mathematically, for a sequence, the loss is the sum of cross-entropy losses at each position: L = -Σ log P(x_i | x_1, ..., x_(i-1)). This is equivalent to minimizing the perplexity of the model on the training data. Lower perplexity indicates the model is more confident and accurate in its predictions.

7.2 Dataset Preparation

Training data quality and quantity are crucial for LLM performance. Modern LLMs are trained on diverse text from the internet, books, code repositories, scientific papers, and more. The total training data can be trillions of tokens. For example, GPT-3 was trained on approximately 300 billion tokens, while even larger datasets are used for newer models.

7.2.1 Data Collection and Filtering

Raw internet text contains noise, duplicates, toxic content, and low-quality data. Careful filtering is essential. Common filtering steps include: removing duplicate documents using hashing or near-duplicate detection, filtering by language using language detection models, removing low-quality content based on heuristics like word count, sentence structure, and readability scores, filtering toxic or harmful content using classifier models, and removing personally identifiable information for privacy.

7.2.2 Dataset Mixing

Different data sources have different characteristics. Web text is diverse but noisy, books are well-written but may be outdated, code teaches programming but has different statistics than natural language, and scientific papers are high-quality but dense and specialized. Mixing these sources in appropriate proportions is important for balanced performance.

The mixing proportions are hyperparameters. Too much web text makes the model susceptible to learning internet quirks, while too little reduces coverage of current events and diverse topics. Dataset mixing can significantly impact downstream performance on different tasks.

7.3 Optimization and Learning Rate Scheduling

LLMs are almost universally trained using AdamW optimizer, a variant of Adam with decoupled weight decay regularization. AdamW maintains exponential moving averages of gradients and squared gradients, providing adaptive learning rates for each parameter. This is essential for stable training of large models.

7.3.1 Learning Rate Schedules

The learning rate is one of the most important hyperparameters. Too high causes instability and divergence, while too low results in slow training and poor convergence. Modern LLMs typically use a

learning rate schedule with warmup and decay.

A common schedule starts with linear warmup from 0 to a maximum learning rate over the first few thousand steps. This allows the model to settle into a good region of the loss landscape before taking large steps. After warmup, the learning rate decays, typically using cosine decay to some minimum value. This decay helps the model converge to a better optimum.

7.3.2 Hyperparameter Selection

Key hyperparameters include maximum learning rate (typically 1e-4 to 6e-4 for large models), warmup steps (typically 2000-4000 steps), weight decay (typically 0.1), beta values for Adam (typically beta1=0.9, beta2=0.95 or 0.999), and gradient clipping (typically 1.0). These values have been refined through extensive experimentation across many model scales.

7.4 Batch Size and Gradient Accumulation

Larger batch sizes generally lead to more stable training and better final performance, up to a point. However, larger batches require more memory. Modern LLM training uses batch sizes of millions of tokens. For example, training might use batches of 2048 sequences of length 2048, totaling 4M tokens per batch.

When hardware memory is limited, gradient accumulation allows simulating larger batches. Instead of updating parameters after each forward-backward pass, gradients are accumulated over multiple passes and then averaged. This provides the training dynamics of a larger batch while fitting in available memory.

7.5 Distributed Training

Training large LLMs requires distributed training across multiple GPUs and often multiple machines. Several parallelism strategies are used in combination: data parallelism, model parallelism, and pipeline parallelism.

7.5.1 Data Parallelism

In data parallelism, each GPU has a copy of the full model. Different GPUs process different batches of data in parallel. After computing gradients, they are averaged across all GPUs and used to update the model parameters on each GPU. This is the simplest form of parallelism and scales well up to the point where model parameters can fit on a single GPU.

7.5.2 Model Parallelism

When models are too large to fit on one GPU, model parallelism splits the model across GPUs. Tensor parallelism partitions individual layers across GPUs - for example, splitting attention heads or feed-forward dimensions. This requires communication between GPUs during forward and backward passes but allows training models larger than a single GPU's memory.

7.5.3 Pipeline Parallelism

Pipeline parallelism splits the model depth-wise, with different transformer layers on different GPUs. Each GPU processes a stage of the pipeline. To keep all GPUs busy, micro-batches are used - a batch is split into smaller chunks that flow through the pipeline. This reduces idle time and improves throughput.

7.6 Mixed Precision Training

Modern training uses mixed precision, typically FP16 (16-bit floating point) or BF16 (bfloat16) instead of FP32. This reduces memory usage by half and speeds up computation on modern GPUs that have specialized hardware for lower precision operations.

To maintain numerical stability, a careful strategy is needed. Master weights are kept in FP32, gradients are computed in FP16/BF16, and loss scaling is used to prevent gradient underflow. The specific implementation depends on the framework - PyTorch's automatic mixed precision or NVIDIA Apex are commonly used.

7.7 Checkpointing and Recovery

Training large models can take weeks or months. Failures are inevitable, so robust checkpointing is essential. Model checkpoints are saved periodically (e.g., every 1000-5000 steps), including model weights, optimizer state, random number generator state, and training step count.

Activation checkpointing (gradient checkpointing) is a different technique to save memory. Instead of storing all intermediate activations during the forward pass, only some checkpoints are stored. During the backward pass, missing activations are recomputed from checkpoints. This trades computation for memory, allowing larger models or batch sizes.

7.8 Monitoring and Evaluation

During training, several metrics are monitored. Training loss should decrease steadily, though it may have noise. Validation loss on a held-out set checks for overfitting. Gradient norms help detect instability. Learning rate is logged to verify the schedule. Memory usage and throughput track computational efficiency.

Beyond these metrics, periodic evaluation on downstream tasks provides insight into capabilities. Perplexity on validation data, accuracy on question-answering tasks, and qualitative examination of generated text all help assess model quality during training.

Chapter 8: Fine-tuning and Instruction Following

8.1 From Pretrained Models to Instruction Followers

Base pretrained models are trained to predict the next token, making them good at continuing text but not inherently good at following instructions or having conversations. Fine-tuning adapts pretrained models to specific behaviors and tasks. This is how base GPT models become ChatGPT or how base Claude models become conversational assistants.

8.2 Supervised Fine-Tuning (SFT)

The first stage of post-training is supervised fine-tuning on demonstration data. This data consists of prompts paired with desired responses. For example: Prompt: 'What is the capital of France?' Response: 'The capital of France is Paris.' The model learns to generate responses like the demonstrations.

8.2.1 Dataset Construction

Creating high-quality demonstration data is crucial. Data is typically collected by having humans write responses to prompts covering diverse tasks: question answering, summarization, translation, creative writing, code generation, mathematical reasoning, and more. The diversity ensures the model learns to handle various instruction types.

Quality matters more than quantity for SFT. Tens of thousands of high-quality demonstrations can be sufficient to adapt a large pretrained model. The demonstrations should be well-written, accurate, helpful, and aligned with desired behavior.

8.2.2 Training Process

SFT uses the same next-token prediction objective as pretraining but only on the response portion of each example. The prompt is provided as context without computing loss, then loss is computed on predicting the response tokens. This teaches the model to generate appropriate responses given prompts.

Training uses a lower learning rate than pretraining (typically 1e-5 to 5e-5) and fewer steps (a few thousand to tens of thousands). The model is already quite capable from pretraining, so SFT mainly steers its behavior rather than teaching new knowledge.

8.3 Reinforcement Learning from Human Feedback (RLHF)

RLHF further refines model behavior using human preferences. Instead of demonstrating correct responses, humans compare multiple model outputs and indicate which is better. This preference data is used to train a reward model, which then guides the language model via reinforcement learning.

8.3.1 Reward Modeling

The reward model learns to predict human preferences. Given a prompt and response, it outputs a scalar score estimating how much humans would like that response. The reward model is typically initialized from the SFT model and trained on comparison data where humans ranked multiple

responses to the same prompt.

For each prompt, the model generates multiple responses. Humans rank these responses or select the best one. The reward model is trained so that higher-ranked responses get higher scores. This is typically done using a ranking loss that encourages the score difference between responses to match their preference difference.

8.3.2 Proximal Policy Optimization (PPO)

With a reward model trained, the language model itself is fine-tuned using reinforcement learning. Proximal Policy Optimization (PPO) is the most common algorithm. The model generates responses to prompts, the reward model scores these responses, and the language model's parameters are updated to increase the expected reward.

However, maximizing reward alone could cause the model to exploit the reward model or produce degenerate outputs. To prevent this, a KL divergence penalty is added, keeping the fine-tuned model close to the SFT model. The objective becomes: maximize reward - β * KL(policy || reference), where β controls the tradeoff between reward and staying close to the original model.

8.3.3 Training Loop

PPO training proceeds in episodes. For each episode: sample prompts from the training set, generate responses using the current policy model, compute rewards using the reward model, compute KL penalty against the reference model, and update the policy model using PPO to maximize the combined objective. This is repeated for thousands of episodes.

8.4 Direct Preference Optimization (DPO)

DPO is a simpler alternative to RLHF that skips reward modeling and RL entirely. Instead, it directly optimizes the language model on preference data. Given pairs of responses where one is preferred, DPO increases the probability of the preferred response while decreasing the probability of the rejected response, with an implicit reward model derived from the policy.

The DPO loss encourages the model to assign higher probability to preferred responses while staying close to a reference model. It's simpler to implement than PPO, more stable, and often achieves comparable or better results. DPO has become increasingly popular for model alignment.

8.5 Constitutional AI and Self-Improvement

Constitutional AI is an approach where the model helps generate its own training data according to principles or a constitution. The model generates responses, critiques them according to principles (e.g., 'be helpful and harmless'), revises its responses, and learns from the revised responses. This can reduce dependence on human labor for fine-tuning data.

8.6 Evaluation and Benchmarking

Evaluating fine-tuned models requires multiple approaches. Automatic metrics include perplexity, task-specific accuracy (e.g., on question-answering benchmarks), and preference model scores. Human evaluation assesses helpfulness, harmlessness, and honesty of responses. Both quantitative benchmarks and qualitative human judgment are important for a complete picture.

Chapter 9: Advanced Topics and Optimization

9.1 Model Compression and Efficiency

Large models are expensive to deploy. Various techniques reduce model size and computational cost while maintaining performance. These include quantization, pruning, distillation, and efficient architectures.

9.1.1 Quantization

Quantization reduces the precision of model weights and activations. Instead of 32-bit floats, we can use 16-bit, 8-bit, or even lower precision. Post-training quantization converts a trained model to lower precision with minimal accuracy loss. Quantization-aware training includes quantization in the training process, learning weights that work well when quantized.

8-bit quantization can reduce model size by 4x with minimal performance degradation. 4-bit quantization is more aggressive but still viable for many applications. These reductions make large models deployable on consumer hardware.

9.1.2 Knowledge Distillation

Knowledge distillation trains a smaller student model to mimic a larger teacher model. The student learns from the teacher's outputs (soft labels), which contain more information than just hard labels. This can produce compact models that retain much of the teacher's capability.

For LLMs, distillation can involve training on the teacher's predicted probability distributions over tokens, or using the teacher to generate training data for the student. Models like DistilBERT and DistilGPT demonstrate that distillation can achieve 60-90% of teacher performance with 40-50% fewer parameters.

9.2 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) combines language models with external knowledge retrieval. When answering a query, the system first retrieves relevant documents from a knowledge base, then conditions the language model on both the query and retrieved documents. This allows accessing information beyond what's stored in model parameters.

RAG systems typically use dense retrieval with learned embeddings. Documents and queries are embedded using a separate model (like a fine-tuned BERT), and the most similar documents are retrieved based on embedding similarity. The language model then generates a response incorporating the retrieved information.

9.3 Chain-of-Thought and Reasoning

Chain-of-thought prompting improves LLM reasoning by encouraging step-by-step thinking. Instead of directly answering a question, the model first generates intermediate reasoning steps, then the final answer. This significantly improves performance on complex reasoning tasks.

Few-shot prompting with chain-of-thought examples is highly effective. By showing the model a few examples of problems solved step-by-step, it learns to apply the same reasoning pattern to new problems. This emergent capability appears more strongly in larger models.

9.4 Multimodal Language Models

Modern LLMs are being extended to handle multiple modalities like images, audio, and video in addition to text. This typically involves encoding non-text modalities into embeddings compatible with the language model, then processing them together with text tokens.

For vision-language models, images are encoded using a vision transformer or CNN, producing a sequence of visual tokens. These are concatenated with text tokens and processed by the language model. Models like GPT-4V and Gemini demonstrate impressive multimodal understanding and reasoning capabilities.

9.5 Long Context and Memory

Extending context length is an active research area. Techniques include positional interpolation (adjusting positional encodings to handle longer sequences), sparse attention patterns, and memory mechanisms that compress old context. Models like GPT-4 support 128K tokens, and some models handle millions of tokens.

External memory systems augment LLMs with explicit memory storage. The model can read from and write to memory, allowing it to track information across very long interactions. This is particularly useful for maintaining conversation history or working on long documents.

Chapter 10: Deployment and Inference

10.1 Inference Optimization

Efficient inference is critical for practical LLM deployment. Autoregressive generation requires running the model once for each token generated, making it computationally intensive. Various optimizations reduce latency and increase throughput.

10.1.1 KV Caching

During autoregressive generation, each token attends to all previous tokens. Without optimization, we would recompute attention for all previous tokens at each step. KV caching stores the key and value projections from previous tokens, so only the new token needs attention computation. This dramatically speeds up generation.

The cache grows with sequence length, consuming memory. For long sequences, memory can become a bottleneck. Techniques like multi-query attention (sharing key and value heads) and grouped-query attention reduce cache size while maintaining performance.

10.1.2 Batching and Continuous Batching

Processing multiple requests in parallel improves throughput. Traditional batching processes requests that arrive together, but requests often arrive at different times and have different lengths. Continuous batching dynamically adds and removes requests from batches as they arrive and complete, maximizing GPU utilization.

10.2 Sampling Strategies

Given the model's probability distribution over next tokens, we need a strategy to select tokens. Greedy sampling always picks the highest-probability token, producing deterministic but often repetitive outputs. Random sampling introduces diversity but can produce incoherent text.

10.2.1 Temperature Scaling

Temperature controls randomness. The logits are divided by temperature before softmax. Temperature < 1 makes the distribution sharper (more deterministic), while temperature > 1 makes it flatter (more random). Temperature = 1 uses the model's original distribution. Lower temperatures work well for factual tasks, while higher temperatures suit creative tasks.

10.2.2 Top-k and Top-p Sampling

Top-k sampling restricts sampling to the k most likely tokens, preventing very unlikely tokens from being selected. Top-p (nucleus) sampling includes the smallest set of tokens whose cumulative probability exceeds p. This adapts the number of candidates to the distribution - more when the model is uncertain, fewer when it's confident.

Combining temperature with top-p is common. For example: temperature 0.7 with top-p 0.9 produces diverse but coherent outputs. These hyperparameters can be tuned based on the application and desired output style.

10.3 Serving Infrastructure

Production LLM serving requires robust infrastructure. Model servers handle requests, manage batching, monitor performance, and scale based on demand. Popular frameworks include vLLM, TGI (Text Generation Inference), and TensorRT-LLM.

Key considerations include: latency requirements (how fast must responses be?), throughput requirements (how many requests per second?), cost constraints (GPU/CPU budget), model size (determines memory needs), and scaling strategy (horizontal scaling with multiple instances vs. vertical scaling with larger instances).

10.4 Safety and Content Filtering

Deployed LLMs need safety measures to prevent harmful outputs. This includes content filtering to block toxic, dangerous, or policy-violating outputs, input validation to reject inappropriate prompts, rate limiting to prevent abuse, and monitoring to detect and respond to issues.

Multiple layers of defense are recommended: input classifiers to detect problematic prompts before generation, output classifiers to filter responses before returning them, and post-hoc monitoring to identify patterns of misuse. These should complement model training (RLHF, etc.) rather than replace it.

Appendix: Mathematical Foundations

A.1 Linear Algebra Essentials

Understanding LLMs requires solid linear algebra fundamentals. Neural networks are built on vectors, matrices, and their operations. This section reviews key concepts.

A.1.1 Vectors and Matrices

Vectors are ordered arrays of numbers. In LLMs, token embeddings are vectors in high-dimensional space (typically 768 to 12,288 dimensions). Matrices are 2D arrays of numbers used for linear transformations. Weight matrices in neural networks are learned parameters that transform inputs.

A.1.2 Matrix Multiplication

Matrix multiplication is the fundamental operation in neural networks. For matrix A of shape (m, n) and matrix B of shape (n, p), their product C = AB has shape (m, p). Element (i, j) of C is the dot product of row i of A and column j of B. This operation implements linear transformations.

A.2 Probability and Information Theory

Language models are probabilistic models of text. Understanding probability and information theory is essential for understanding how they work and are trained.

A.2.1 Probability Distributions

A probability distribution assigns probabilities to outcomes. For discrete variables like tokens, we use probability mass functions where probabilities sum to 1. Conditional probability P(A|B) is the probability of A given B. Language models learn P(token | previous tokens).

A.2.2 Cross-Entropy and KL Divergence

Cross-entropy measures how well a predicted distribution matches a true distribution. It's the expected negative log-likelihood under the true distribution. Minimizing cross-entropy is equivalent to maximizing likelihood, making it the standard loss for language modeling.

KL divergence measures how one distribution differs from another. It's always non-negative and equals zero only when distributions are identical. KL divergence is used in RLHF to keep the fine-tuned model close to the reference model.

A.3 Calculus and Optimization

Training neural networks requires optimization through gradient descent. Understanding calculus, particularly derivatives and the chain rule, is necessary for understanding how networks learn.

A.3.1 Derivatives and Gradients

The derivative measures how a function changes with its input. For functions of multiple variables, the gradient is a vector of partial derivatives. The gradient points in the direction of steepest increase. Gradient descent moves in the negative gradient direction to minimize a function.

A.3.2 Backpropagation

Backpropagation computes gradients efficiently using the chain rule. It works backward through the network, computing how the loss changes with respect to each parameter. Modern frameworks like PyTorch and TensorFlow implement automatic differentiation, computing gradients automatically.

A.4 Computational Complexity

Understanding computational complexity helps reason about model efficiency and scalability. Big-O notation describes how computation or memory scales with input size.

For transformers, attention has O(n^2) complexity in sequence length n, because every position attends to every other position. Feed-forward networks have O(n) complexity as they process each position independently. Understanding these complexities guides architectural decisions and optimization efforts.

Conclusion

This guide has covered the complete journey of building large language models from scratch. We explored tokenization and text preprocessing, fundamental neural network components, the transformer architecture and attention mechanisms, implementing GPT-style models, training strategies and optimization, fine-tuning for instruction following, advanced topics like efficiency and multimodality, and deployment considerations.

Building LLMs requires bringing together concepts from machine learning, natural language processing, systems engineering, and more. While the fundamental architecture is relatively simple - stacks of attention and feed-forward layers - making these models work at scale requires careful attention to implementation details, training procedures, and infrastructure.

The field continues to evolve rapidly. Models are growing larger, training is becoming more efficient, and new capabilities are emerging. Understanding the foundations covered in this guide provides the basis for working with current models and adapting to future developments.

Whether you're implementing models from scratch, fine-tuning existing models, or deploying LLMs in production, the principles and techniques discussed here form the essential knowledge base. The best way to truly understand these concepts is through hands-on implementation and experimentation.

Thank you for reading this guide. Happy building!