19
Introduction to Transformer Architecture and Large Language Models

Since 2020, language models (LMs) have become a prominent area of AI research. This chapter introduces their foundational concepts.

The goal of a Large Language Model (LLM) is to model the probability distribution of human language, $P(\text{text})$. This is a generative task over sequential, conditional distributions. A successful model must predict the next piece of text given a preceding context, a task known as *autoregressive generation.*

19.1 The Probabilistic Framework of Language

Human text is assumed to be sampled from a probability distribution. Given a sequence of tokens $w_1, w_2, \ldots, w_L$, the probability of the sequence is given by the chain rule:

$$P(w_1, \ldots, w_L) = P(w_1) \prod_{t=2}^{L} P(w_t | w_{<t}), \tag{19.1}$$

where $w_{<t}$ denotes the context $(w_1, \ldots, w_{t-1})$. This factorization implies that modeling language requires a way to compute the conditional probability of the next token given its history. This is the core task of an autoregressive language model.[1]

This idea dates to the 1950s, when Claude Shannon used computable approximations of this conditional probability to estimate the entropy of printed English at around 1 bit per character, suggesting that text is highly predictable. Simple n-gram models make a Markov assumption, limiting the context to the previous $n-1$ tokens: $P(w_t|w_{<t}) \approx P(w_t|w_{t-n+1}, \ldots, w_{t-1})$. While effective for decades, n-gram models cannot capture long-range dependencies.

Next-word prediction quality depends critically on context length. Short contexts are often ambiguous; longer ones can resolve ambiguity and sharpen the prediction.

[1] This exposition focuses on autoregressive models. Other architectures like BERT are trained on different objectives, such as filling in masked words, to compute semantic embeddings.

Example 19.1.1 (The Importance of Context)**.** *Language models need long context, since distant context can dramatically alter the probability distribution for the next word. Consider predicting the next word:*

1. ***Short Context:*** *"The witness took the stand and swore to tell the..". Here the most probable next word is truth.*
2. ***Longer Context:*** *"The defense attorney's star witness was a known compulsive liar. The witness took the stand and swore to tell the" The added context reduces the probability of truth and might increase the probability of completions like whole truth, or even ironic ones.*

To process arbitrarily long histories, early neural approaches used *Recurrent Neural Networks* (RNNs). An RNN maintains a hidden state vector, $h_t = f(h_{t-1}, w_t)$, which acts as a compressed summary of the sequence history.

19.1.1 The Autoregressive Objective: Next-Word Prediction

Modern LMs use a deep neural network to learn a parameterized function, P_θ, that approximates the true conditional probability $P(w_t|w_{<t})$. The network is trained via Maximum Likelihood Estimation (MLE). Given a training corpus $\mathcal{D}$, the goal is to find parameters θ that maximize the probability of the data. The log-likelihood is:

$$\mathcal{L}_{\text{MLE}}(\theta) = \log P_\theta(\mathcal{D}) = \sum_{\text{sequence} \in \mathcal{D}} \sum_t \log P_\theta(w_t|w_{<t}) \tag{19.2}$$

Maximizing this is equivalent to minimizing the negative log-likelihood, which is the cross-entropy loss. For a single prediction, the loss is the negative log-probability of the correct next token:

$$H(p, q_\theta) = \mathbb{E}_{w\sim p}\left[-\log q_\theta(w|\text{context})\right] \tag{19.3}$$

An Information-Theoretic Viewpoint Let $p(w|\text{context})$ be the true distribution of the next word and $q_\theta(w|\text{context})$ be our model's approximation. The loss we minimize, $-\log q_\theta(w_t^{\text{actual}})$, is a single-sample estimate of the cross-entropy $H(p, q_\theta)$:

$$D_{KL}(p||q_\theta) = \mathbb{E}_{w\sim p}\left[\log \frac{p(w|\text{context})}{q_\theta(w|\text{context})}\right] = H(p, q_\theta) - H(p) \tag{19.4}$$

Cross-entropy relates to the Kullback-Leibler (KL) divergence, which measures the inefficiency of q_θ in representing p:

$$D_{KL}(p||q_\theta) = \mathbb{E}w \sim p\left[\log \frac{p(w|\text{context})}{q\theta(w|\text{context})}\right] = H(p, q_\theta) - H(p), \tag{19.5}$$

where $H(p)$ is the entropy of the true distribution—the irreducible unpredictability of language. This gives a fundamental relationship

for the expected loss:

$$\text{Expected Cross-Entropy Loss} = \text{Entropy} + \text{KL Divergence}. \quad (19.6)$$

Since entropy is a fixed property of language, minimizing the cross-entropy loss is equivalent to minimizing the KL divergence between the model and the true data-generating distribution.

Goal of Language Modeling: *Minimize the KL divergence from the true data-generating distribution to the model's distribution.*

19.1.2 *A Conceptual Bridge to the Transformer: Parallel Processing with Queries, Keys, and Values*

The transformer architecture was designed to overcome the limitations of the RNN architecture. In theory, RNN allows the model to capture dependencies of any length. In practice, training must contend with the dreaded vanishing gradient problem (Chapter 13). Since the RNN is used to generate token-wise output, the gradient calculation needs to be done over extremely long paths when training even on short paragraphs.

The Transformer architecture[2] overcomes this by processing all tokens in parallel. Furthermore, to avoid blowing up the total number of parameters (which would in turn blow up all other costs, including amount of training data needed), a single computational unit, called a *Transformer block*, is applied to every token's representation simultaneously. This is yet another example of *weight-sharing*, and should remind you of the convolutional net (Chapter 12).

[2] A. Vaswani et al. *Attention is all you need. 2017.*

In effect this creates a parallel processing highway for each token as it passes through the layers. For this parallel processing to be meaningful, the model needs a dynamic way for tokens to exchange information as they pass through the layers. How can the token 'it' in the sentence "The bottle is empty, so you can throw it away" figure out that it should focus on 'bottle' and not 'empty'?

The Transformer solves this with a mechanism inspired by information retrieval. At each layer, every token's vector representation is used to derive three distinct vectors that play different roles in this information exchange:

- **Query** (**q**): Represents what information a token is looking for. The query from 'it' might effectively ask, "what noun am I referring to?"
- **Key** (**k**): Represents what kind of information a token offers or advertises. The key from 'bottle' might signal, "I am a singular, physical object that can be acted upon."

- **Value** (**v**): Represents the actual content or meaning of the token. If the key from 'bottle' is a good match for the query from 'it', then the value from 'bottle' is the information that gets passed along.

The model can then dynamically compute a relevance score for each token by matching every **Query** with every **Key**. A high score indicates that a key's corresponding **Value** is important for the querying token. This allows the model to build a new, context-aware representation for each token by aggregating the relevant information from all other tokens in the sequence.

This query-key-value mechanism, known as **self-attention**, is the core of the Transformer. But before we can detail its mathematical formulation, we must first understand how raw text is converted into the initial vector representations that serve as its input.

19.2 Representing Language: From Text to Tensors

Neural networks require numerical tensors. Converting text to tensors involves three steps: tokenization, embedding, and positional encoding.

19.2.1 Step 1: Tokenization

Tokenization segments text into discrete units called tokens. Using words as tokens is impractical due to the unbounded size of language (proper nouns, slang, misspellings), which leads to the out-of-vocabulary (OOV) problem.

Modern models use subword tokenization. This method breaks rare words into smaller, meaningful units (e.g., "decentralization" -> de, central, iz, ation) while keeping common words whole. This approach eliminates the OOV problem, keeps the vocabulary size manageable (typically 30k-100k), and improves statistical efficiency by sharing subword representations. Algorithms like Byte-Pair Encoding (BPE) build the vocabulary by starting with individual characters and iteratively merging the most frequent adjacent token pairs. The output is a sequence of integers, each the unique ID of a token.

19.2.2 Step 2: Embeddings

The integer token IDs are mapped to dense vectors. An **embedding layer**, which is a learnable lookup table $E \in \mathbb{R}^{V \times d}$, performs this mapping. Here, V is the vocabulary size and d is the model's hidden dimension. The embedding for the token with ID id_t is the corresponding row vector from this matrix:

$$\mathbf{x}_t = E[id_t]. \tag{19.7}$$

The embedding matrix E is learned during training, organizing the vector space so that semantically similar tokens are close to one another.

19.2.3 Step 3: Positional Encoding

As mentioned, a transformer processes all tokens in parallel and has no inherent sense of sequence order. Without positional information, it would treat "the man bit the dog" and "the dog bit the man" as identical.

To solve this, we inject the position of each token into its vector. The original Transformer adds a fixed **positional encoding** vector, $\mathbf{p}_t \in \mathbb{R}^d$, to each token embedding. The components of $\mathbf{p}_t$ are defined by sinusoids of varying frequencies:

$$(\mathbf{p}_t)_{2i} := \sin(t/10000^{2i/d}) \tag{19.8}$$

$$(\mathbf{p}_t)_{2i+1} := \cos(t/10000^{2i/d}) \tag{19.9}$$

where t is the position, and $i \in \{0, \ldots, d/2 - 1\}$ indexes the pairs of dimensions. This function provides a unique encoding for each position. Because $\mathbf{p}_{t+k}$ can be represented as a linear function of $\mathbf{p}_t$, the model can easily learn relative positioning. The final input vector for each token is the sum of its embedding and positional encoding:

$$\mathbf{x}'_t = \mathbf{x}_t + \mathbf{p}_t. \tag{19.10}$$

This produces a sequence of vectors where each vector encodes both a token's meaning and its position.

Positional encodings are further explained in Section 19.4.

19.3 The Transformer Architecture

19.3.1 The Core Innovation: Self-Attention

RNNs process tokens sequentially, creating a computational bottleneck and suffering from vanishing gradients over long distances. Self-attention solves this by allowing every token to directly interact with every other token in its context in parallel.

Mathematical Formulation: Scaled Dot-Product Attention The mechanism operates on three vectors derived from each input token representation $\mathbf{x}_i \in \mathbb{R}^d$: **Query** ($\mathbf{q}_i$), **Key** ($\mathbf{k}_i$), and **Value** ($\mathbf{v}_i$).[3] These are produced by learned linear projections with weight matrices

[3] The query-key-value terminology is from databases, which is the analogy that the designers were working with.

$W_Q, W_K \in \mathbb{R}^{d \times d_k}$ and $W_V \in \mathbb{R}^{d \times d_v}$.

$$\mathbf{q}_i = \mathbf{x}_i W_Q$$
$$\mathbf{k}_i = \mathbf{x}_i W_K$$
$$\mathbf{v}_i = \mathbf{x}_i W_V$$

Intuition: The query $\mathbf{q}_i$ represents the current token's request for information. It is matched against all available keys $\mathbf{k}_j$ (where $j \leq i$) to compute attention scores. These scores determine how much each value vector $\mathbf{v}_j$ contributes to the output for token i.

The output for token i, $\mathbf{z}_i$, is a weighted sum of the value vectors:

$$\mathbf{z}_i = \sum_{j=1}^{i} \alpha_{ij} \mathbf{v}_j \tag{19.11}$$

The weights α_{ij} are computed via a scaled dot-product followed by a softmax function. For autoregressive decoding, we only attend to previous tokens ($j \leq i$), a constraint known as causal masking.

$$\alpha_{ij} = \text{softmax}\left(\frac{\mathbf{q}_i \cdot \mathbf{k}j}{\sqrt{d_k}}\right) \quad \text{for } j \leq i \tag{19.12}$$

Matrix Form: For a sequence of length L, the inputs $(\mathbf{x}_1, \ldots, \mathbf{x}_L)$ are stacked into a matrix $X \in \mathbb{R}^{L \times d}$. The operation can be expressed concisely:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V \tag{19.13}$$
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V \tag{19.14}$$

Here, M is a mask matrix where $M_{ij} = 0$ for $j \leq i$ (allowing attention) and $M_{ij} = -\infty$ for $j > i$ (prohibiting attention), thereby enforcing causality. The scaling factor $\sqrt{d_k}$ prevents the dot products from growing too large, which would push the softmax into regions with vanishing gradients.

19.3.2 The Transformer Block

A Transformer is a stack of identical blocks. A single decoder block has three main components:

1. **Masked Multi-Head Self-Attention:** Instead of a single attention function, the mechanism is run multiple times in parallel with different, learned projection matrices. These "heads" allow the model to attend to different types of information and relationships simultaneously. If there are h heads, the dimensions are typically set to $d_k = d_v = d/h$. The outputs of the heads are concatenated and projected back to d with a final weight matrix W_O.

2. **Feed-Forward Network (FFN):** This is a two-layer multilayer perceptron (MLP), applied independently to each token's representation. It consists of a linear transformation to a higher dimension (often $4 \times d$), a ReLU or GeLU activation, and a linear transformation back to d. This component processes the information gathered by the attention layer.

3. **Residual Connections & Layer Normalization:** To enable stable training of deep networks, each of the two sub-layers (attention and FFN) is wrapped with a residual connection and followed by Layer Normalization. The output of each sub-layer is $\text{LayerNorm}(\mathbf{x} + \text{Sublayer}(\mathbf{x}))$. This ensures an unimpeded gradient flow and stabilizes layer inputs as discussed in Chapter 13.

The Data Pathway Through the Block The flow of information through a single Transformer block proceeds in two main steps. First, the input vectors for each token pass through the **Masked Multi-Head Self-Attention** layer. The output of this layer—a new, context-aware vector for each token—is then added to the original input via a **residual connection**. This sum is then normalized using **Layer Normalization**. This entire result then serves as the input to the second step: the **Feed-Forward Network (FFN)**. The FFN processes each token's representation independently, providing further non-linear transformation and allowing the model to "digest" the information gathered by the attention mechanism. The output of the FFN is, in turn, passed through its own residual connection and a final Layer Normalization. This final output vector for each token is the result of the entire Transformer block, which is then passed as input to the next identical block in the stack.

19.4 Deeper Dive: Why Positional Encodings

Students are often confused about positional encodings and the particular choices above. Now we explain this.

(a) The Problem: Self-Attention is a Set Operation

The self-attention mechanism, at its core, is a set of parallel matrix multiplications that is fundamentally insensitive to order. If we only feed it a sequence of token embeddings $(x_1, x_2, \ldots, x_L)$, the model treats them not as a sequence, but as an unordered set.

Consider the sentences:

1. "The man bit the dog."

2. "The dog bit the man."

Without positional information, the initial input is the set of embeddings $\{x_{\text{the}}, x_{\text{man}}, x_{\text{bit}}, x_{\text{dog}}\}$. The Query, Key, and Value vectors for each token (e.g., $q_{\text{man}}, k_{\text{man}}, v_{\text{man}}$) depend only on the token's embedding, not its location.

Let the input matrices for Sentence 1 be Q_1, K_1, V_1. For Sentence 2, the words are just permuted. This means the input matrices are simply row-permutations of the first set: $Q_2 = PQ_1$, $K_2 = PK_1$, and $V_2 = PV_1$, where P is the permutation matrix that swaps the rows for "man" and "dog."

The attention output is $Z = \text{softmax}(QK^T/\sqrt{d_k})V$. Let's see what happens in Sentence 2:

$$\begin{aligned} Z_2 &= \text{softmax}\left(\frac{(PQ_1)(PK_1)^T}{\sqrt{d_k}}\right)(PV_1) \\ &= \text{softmax}\left(\frac{PQ_1K_1^TP^T}{\sqrt{d_k}}\right)PV_1 \end{aligned}$$

Because the softmax is applied element-wise, applying a permutation before and after is the same as permuting the final result. So, the attention score matrix for Sentence 2 is just $A_2 = PA_1P^T$. The final output is:

$$Z_2 = (PA_1P^T)(PV_1) = PA_1(P^TP)V_1 = PA_1V_1 = PZ_1$$

This proves that the output representations Z_2 are just a permutation of Z_1. The final vector for "man" in Sentence 2 is identical to the final vector for "man" in Sentence 1. The model is completely blind to the critical change in meaning, which is unacceptable.

(b) The Naive Solution: Using an Integer Index

The most straightforward idea is to add the integer position t in $1, 2, 3, ...$ as an extra feature to each token's vector. This fails for two key reasons, one theoretical and one practical:

1. **Poor Generalization:** A neural network learns patterns from its training data. If we feed it raw position integers, a weight that learns a useful pattern for position '5' has no reason to work for position '50' or '500'. The model cannot generalize to sequence lengths it hasn't seen. It would have to learn the "meaning" of each position number independently.

2. **Practical Infeasibility:** This generalization issue has a critical practical consequence. In real-world applications, models must handle a flexible and variable context length. It is computationally prohibitive and practically impossible to run a new, expensive

training run for every possible context length. We need a system that can gracefully handle sequence lengths longer than those seen during training. An integer index provides no mechanism for this kind of extrapolation.

3. **No Inherent Relational Information:** The integers '5' and '6' are numerically close, but a linear layer doesn't automatically understand this as "adjacent positions." To the network, they are just different values. There is no smooth, geometric relationship between the numbers t and $t + k$ that a matrix multiplication can easily exploit.

(c) The Elegant Solution: Encoding Position in Geometry

We need an encoding scheme where the relationship between the encodings of position t and $t + k$ is not arbitrary, but is instead captured by a simple, consistent geometric transformation—specifically, a linear transformation that a neural network can perform.

The sinusoidal encoding achieves exactly this. For any fixed offset k, the positional encoding of $t + k$ is a **linear transformation** of the positional encoding of t.

Let's illustrate with a simplified 2D example. Imagine our positional encoding for position t was just a single 2D vector determined by one frequency ω:

$$\mathbf{p}_t = \begin{bmatrix} \sin(\omega t) \\ \cos(\omega t) \end{bmatrix}$$

This vector lies on the unit circle. As t increases, the vector simply rotates. Now, let's look at the encoding for position $t + k$:

$$\mathbf{p}_{t+k} = \begin{bmatrix} \sin(\omega(t+k)) \\ \cos(\omega(t+k)) \end{bmatrix}$$

Using the trigonometric angle addition identities, we can expand this:

$$\sin(\omega t + \omega k) = \sin(\omega t)\cos(\omega k) + \cos(\omega t)\sin(\omega k)$$
$$\cos(\omega t + \omega k) = \cos(\omega t)\cos(\omega k) - \sin(\omega t)\sin(\omega k)$$

This looks complicated, but it is just a matrix-vector multiplication:

$$\mathbf{p}_{t+k} = \begin{bmatrix} \cos(\omega k) & \sin(\omega k) \\ -\sin(\omega k) & \cos(\omega k) \end{bmatrix} \begin{bmatrix} \sin(\omega t) \\ \cos(\omega t) \end{bmatrix} = \mathbf{M}_k \cdot \mathbf{p}_t$$

This is the crucial insight. The matrix $\mathbf{M}_k$ is a simple 2D rotation matrix. Critically, **it depends only on the offset k, not the original position t.**

This means that the task of finding a token k positions away is now a consistent, learnable, linear operation. The network doesn't

have to memorize every position. It just has to learn the rotation matrices ($\mathbf{M}_k$) that correspond to relative positions. This "learning a matrix" is exactly a task that a linear layer is perfectly equipped for.

The full d-dimensional positional encoding in the Transformer simply does this in parallel across many different frequencies ω, creating a high-dimensional "fingerprint" that encodes relative positions in a way that is perfectly suited for the linear algebra of the attention layer.

19.5 *The Science of Scale: Predictable Improvement and the Chinchilla Law*

The remarkable capabilities of modern Large Language Models did not arise from a single architectural breakthrough, but rather from the discovery that their performance improves in a predictable way with scale. This understanding, known as **scaling laws**, transformed LLM development from an art into a science of resource allocation. The core question these laws address is: given a fixed budget of computational resources, how should we best allocate it to achieve the lowest possible prediction error?

The scaling paradigm rests on three pillars:

- **Model Size (N):** The number of trainable parameters in the neural network.
- **Dataset Size (D):** The number of tokens the model is trained on.
- **Compute (C):** The total number of floating-point operations (FLOPs) used for training. For a given model and dataset, this is the primary constraint.

For Transformer models, the training compute is dominated by the forward and backward passes. A widely used approximation relates these three variables:

$$C \approx 6 \times N \times D \tag{19.15}$$

This formula highlights the fundamental trade-off: for a fixed compute budget C, making the model larger (increasing N) requires training on fewer tokens (decreasing D), and vice versa. The goal is to find the optimal balance.

19.5.1 *Early Insights and the Chinchilla Correction*

The foundational work by Kaplan et al. (2020) first demonstrated that an LLM's test loss decreases as a smooth and predictable power law as N, D, and C are increased. This was a landmark result, suggesting that simply scaling up existing architectures on more data would

reliably yield better models. However, their work suggested that model size (N) was the most important factor, leading to a race to build ever-larger models, such as the 175-billion parameter GPT-3.

In 2022, researchers at DeepMind revisited this question in the paper "Training Compute-Optimal Large Language Models," colloquially known as the **Chinchilla** paper. They hypothesized that previous models might be "undertrained"—that is, they were too large for the amount of data they were trained on. To test this, they meticulously trained over 400 models, systematically varying N and D while keeping the compute budget C fixed for different model families.

This methodology allowed them to map out the relationship between loss and the N/D ratio. They found that for any fixed compute budget, the loss follows a U-shaped curve. Models that are too small for the data budget cannot learn effectively, while models that are too large for the data budget begin to overfit and do not have enough data to realize their full potential. The minimum of this curve represents the **compute-optimal** model for that budget.

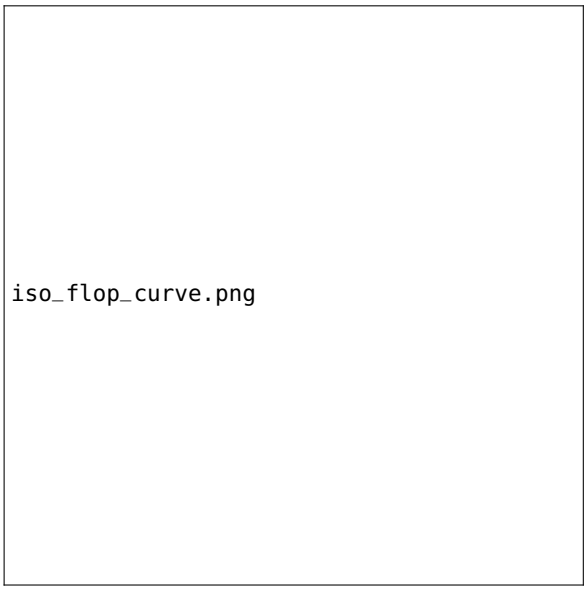

Figure 19.1: A conceptual illustration of an IsoFLOP curve. For a fixed compute budget, both very small and very large models yield a higher loss. The minimum of the curve identifies the compute-optimal model size for that budget.

19.5.2 The Chinchilla Scaling Law: Key Findings

The most impactful finding from the Chinchilla paper was a simple and powerful rule of thumb: for compute-optimal training, the model size and dataset size should be scaled in roughly equal proportion. Specifically, they found that the optimal number of training tokens is

approximately **20 times the number of model parameters**.

$$D_{\text{optimal}} \approx 20 \times N \tag{19.16}$$

This was a radical departure from the previous paradigm. It implied that models like GPT-3 (175B parameters, 300B tokens) and Gopher (280B parameters, 300B tokens) were massively undertrained. According to the Chinchilla law, a 175B parameter model should have been trained on nearly 3.5 trillion tokens.

The paper's central demonstration was the Chinchilla model itself: a 70-billion parameter model trained on 1.4 trillion tokens. Despite being less than half the size of Gopher, Chinchilla outperformed it on a wide range of downstream tasks, using the same total compute budget. This proved that a smaller model trained on more data was a more efficient allocation of resources.

The Chinchilla authors proposed a refined functional form for the scaling law:

$$L(N, D) = E_{\text{min}} + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}} \tag{19.17}$$

Here, E_{min} represents the irreducible entropy of language, while the other two terms represent the error contributions from finite model size and finite data, respectively. By fitting this equation to their experimental data, they found new exponents α and β that gave much greater weight to the dataset size D than previous estimates had.

Figure 19.2: The Chinchilla paper's main finding. It shows that their new scaling law (blue) predicts that for a given compute budget, the optimal model is smaller and trained on more data than predicted by the earlier Kaplan et al. law (dashed black). Existing large models like Gopher and GPT-3 were found to be far from the optimal frontier.

19.5.3 The New Frontier: Beyond Monolithic Pre-training

The Chinchilla law defined the paradigm for several years, but the frontier of scaling is once again evolving, driven by new practical constraints and goals.

The Inference vs. Training Trade-off The Chinchilla law is "compute-optimal" for *training*. However, in practice, models are trained once but used for inference millions of times. A smaller model is faster and cheaper to run. This has led to the strategy of deliberately **"over-training"** a smaller model. For instance, the Llama 2 7B model was trained on 2 trillion tokens, far exceeding the Chinchilla-optimal 140 billion tokens. While this is inefficient from a training-compute perspective, the resulting model is highly capable for its size, making it far more economical to deploy.

The Data Constraint and Quality Imperative The Chinchilla recipe requires trillions of tokens for state-of-the-art models, pushing the limits of available high-quality text data on the internet. This has led to two major shifts:

1. **Data Repetition:** Research has shown that training on repeated data is a viable strategy, but with diminishing returns. Training on the same data for up to 4 epochs yields negligible performance loss compared to unique data, but beyond that, the value of additional compute decays rapidly. This creates a "data-constrained" scaling regime.

2. **Shift from Quantity to Quality:** The most significant recent trend is the move away from simply scraping more data to meticulously curating and filtering it. The composition of the pre-training dataset is now seen as a critical factor. Instead of a monolithic dataset scraped from the web, leading models are trained on a carefully crafted mix of text, code, and dialogue. Furthermore, to enhance specific skills like complex reasoning, developers are increasingly using high-quality **synthetic data**—data generated by other powerful AI models—as part of the pre-training mix. This focus on data quality, not just scale, marks the new frontier in building powerful foundation models.

19.6 Mixture-of-Experts (MoE) Architecture

As the size of dense Transformer models grew, they faced a fundamental bottleneck: every part of the model was activated for every single token processed. This meant that doubling the model size

roughly doubled the computational cost of inference. The **Mixture-of-Experts (MoE)** architecture was introduced as a way to break this paradigm, allowing for a dramatic increase in model capacity (parameter count) without a proportional increase in computational cost.

The core idea is to replace the dense, monolithic components of a neural network with a collection of smaller "expert" networks and a "router" that dynamically chooses which experts to use for each input token.

19.6.1 Architecture of an MoE Layer

In a Transformer, the MoE layer typically replaces the feed-forward network (FFN) sub-block. It consists of two main components:

1. **A Set of Experts ($E_1, ..., E_n$):** Each expert is itself a standard FFN. There might be 8, 16, or even more experts within a single MoE layer.

2. **A Gating Network (or Router):** This is a small, trainable neural network that takes a token's representation as input and outputs a probability distribution over all the available experts. Its job is to learn which experts are best suited to process that specific token.

The process for a single token $\mathbf{x}$ is as follows:

1. The token representation $\mathbf{x}$ is fed into the gating network G.

2. The gating network outputs a vector of logits, which are converted to probabilities (e.g., via softmax) representing the "vote" for each expert: $G(\mathbf{x}) = \text{softmax}(\mathbf{W}_g\mathbf{x})$.

3. Instead of using all experts, the system employs **sparse routing**. Only the top-k experts (typically $k = 2$) are chosen to process the token.

4. The final output $\mathbf{y}$ for the token is the weighted sum of the outputs from the chosen top-k experts. The weights are the probabilities assigned by the gating network.

$$\mathbf{y} = \sum_{i \in \text{TopK}(G(\mathbf{x}))} G(\mathbf{x})_i \cdot E_i(\mathbf{x}) \tag{19.18}$$

This sparsity is the key. Even if a model has 8 experts, by only using 2 for each token, the computational cost (FLOPs) is similar to that of a much smaller dense model, while still benefiting from the capacity and specialized knowledge of all 8 experts.

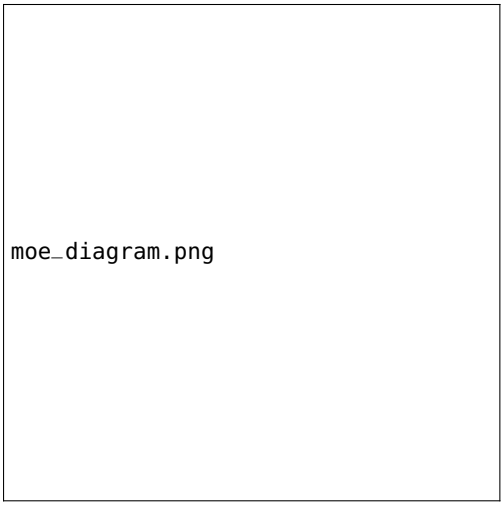

Figure 19.3: Diagram of a Mixture-of-Experts layer. The gating network (router) directs each input token to a sparse subset of experts (e.g., the top 2). The final output is a weighted combination of the outputs from the active experts.

19.6.2 Benefits and Challenges

MoE models offer a compelling path to scaling beyond what is practical with dense architectures, but they come with their own set of trade-offs.

Benefits

- **Massive Model Capacity:** MoE allows for models with staggering parameter counts (hundreds of billions or even trillions) that would be infeasible to train or run as dense models.
- **Compute-Efficient Inference:** The number of active parameters for any given forward pass is much smaller than the total. For example, a 100B parameter MoE model with a top-2 routing might only use around 25B active parameters per token, offering inference speeds comparable to a 25B dense model.
- **Expert Specialization:** In theory, the experts can learn to specialize in different domains of knowledge, such as programming syntax, historical facts, or conversational language, making the model more effective.

Challenges

- **High Memory Requirements:** While inference FLOPs are low, the entire model—all experts included—must be loaded into VRAM. This makes deploying large MoE models very demanding on hardware.
- **Training Instability and Load Balancing:** A major challenge is ensuring that all experts are used effectively. The gating network

might develop a preference for a few "popular" experts, leaving others under-trained. To counteract this, an auxiliary **load balancing loss** is added during training. This loss penalizes the model if the distribution of tokens to experts is too uneven, encouraging the router to spread the load.

- **Communication Overhead:** In a distributed training setup, tokens and their activations need to be shuffled between different GPUs/TPUs to reach their assigned experts, which can introduce communication bottlenecks.

In summary, MoE represents a crucial evolution in model architecture, trading the simplicity of dense models for a more complex but far more scalable and compute-efficient alternative.

19.7 How to generate text from an LM

The title of this section may feel ridiculous, since the very definition of LM's in (19.1) involves ability to predict the next word given the preceding words. This suggests an obvious way to generate good text: use the model distribution for the first word to sample a particular w_1, then sample from the distribution $\Pr[w_2|w_1]$ to generate the second word, and carry on like that. The procedure just described is called **random generation** or simply **sampling** and it actually does not produce good text[4].

[4] The quality of **sampling** gets better as models scale up.

Example 19.7.1. *Reason from first principles why random generation might be problematic. (Hint: it works better and better as you scale up the model.)*

The next natural idea is **greedy:** Having generated $w_1 w_2 \ldots w_i$, generate w_{i+1} using the word that has the highest probability in the next position. At first sight this seems appealing since the training objective for LMs implicitly trains them to maximise the probability they assign to the training text given to them. So when generating text, why not try to generate pieces of text that are given the highest possible probability by the model? [5] The reason is that, as explained above, the true goal of language modeling is to minimize KL distance to the human distribution.

[5] Actually greedy doesn't quite maximise the probability but is an attempt in that direction.

Greedy text looks flat and unexciting. For instance the greedy continuations of *Thanks for the dessert, it was* ... is probably *great*, but there could be a variety of more interesting ones with lower probability, such as *exquisite*, *life-changing* etc. Human communication often veers into low-probability words. Indeed, the perplexity of text produced by the greedy method is far from that of human-generated text! A good discussion of this issue appears in [6], from where Fig-

[6] A Holtzman, J Buys, L Du, M Forbes, and Y Choi. The curious case of neural text degeneration. *ICLR*, 2020

ure 19.4 was taken.

Method	Perplexity
Human	12.38
Greedy	1.50
Beam, b=16	1.48
Stochastic Beam, b=16	19.20
Pure Sampling	22.73
Sampling, t=0.9	10.25
Top-k=40	6.88
Top-k=640	13.82
Top-k=40, t=0.7	3.48
Nucleus p=0.95	**13.13**

Figure 19.4: Perplexity of text from various generation methods. Random and Greedy are pretty bad. Nucleus Sampling with $p = 0.95$ gets closest to human. (We did not describe beam search, so please ignore those rows.)

The best methods actually *reshape* the distribution via some greedy-ish selections.

- **Top-k:** Identify the top k choices for first word, and restrict yourself to pick the first word among them (i.e., disallow picking any other word in this position). If their combined probability is p_k you do this by rescaling the probabilities of these k words by $1/p_k$ and zero-ing probabilities of all other words, and then pick from this distribution. Having picked the first word, continue similarly to pick the rest. You set k by trial and error to best match perplexity to human text.
- **Nucleus sampling (aka "top p"):** This is a softer variant of the above. Instead of making a hard decision about the number of possible choices for the first word (i.e., k), decide that you will allow k to vary but then impose hard constraint that the total probability of all the choices you will allow in the first position is p. Continue similarly for rest of the word positions, with the same "probability budget." You set p by trial and error to best match perplexity to human text.