

CMPUT 654: Mathematical Foundations of Modern AI Systems

Fall 2026

Lecture 1: Decoder transformers and autoregressive inference

September 1, 2026

*Lecturer: Csaba Szepesvári**Note prepared by: Csaba Szepesvári*

About these lecture notes. These notes give a self-contained account of inference through an archetypical decoder-transformer architecture. They began as an example of a scribe note and have grown to include further derivations, figures, and background. This extended treatment is not expected of student scribe submissions; published notes may incorporate additions by the lecturer.

Scope relative to class. Administrative material occupied a substantial part of the meeting. The class covered the harness, tokenization, decoder inference, mixture-of-experts layers, attention cost, KV caching, and context-length limitations. Training was discussed verbally, without loss or gradient equations. These notes supply explicit matrix conventions, multi-head attention and normalization formulas, and further derivations and figures. The architecture comparison in [Section 1.13](#) and tokenizer construction in [Appendix A](#) provide optional background. The motivation for learning and the first learning-theoretic results begin in Lecture 2.

1.1 The model inside a harness

The goal of this lecture is to understand the computation performed by a decoder transformer and its role in an AI system. We follow the path from a token prefix to the next-token distribution, then examine generation, computational cost, and context-length limitations.

As of 2026, today's cutting edge AI systems are composed of a big neural network and some code that makes the neural network usable. A *harness* is the surrounding program that constructs the model's context, calls the model, interprets the result, optionally invokes tools, records the new observation, and decides whether to call the model again. It may provide instructions, conversation history, retrieved documents, external memory, or tool outputs. [Figure 1](#) shows the harness loop and the token-generation loop within a model call. The *prompt* is the initial input supplied to a model call. After tokenization, the prompt tokens form the initial prefix; generated tokens then extend that prefix. Output tokens may be detokenized into text or parsed as a tool call. The resulting observation updates the task state, and the loop may begin again. A harness can use several model calls before returning an answer. The behavior of the complete system can therefore depend on the tokenizer, model parameters, decoding rule, maintained state, tools, and control loop. The rest of this note expands the tokenizer and model boxes.

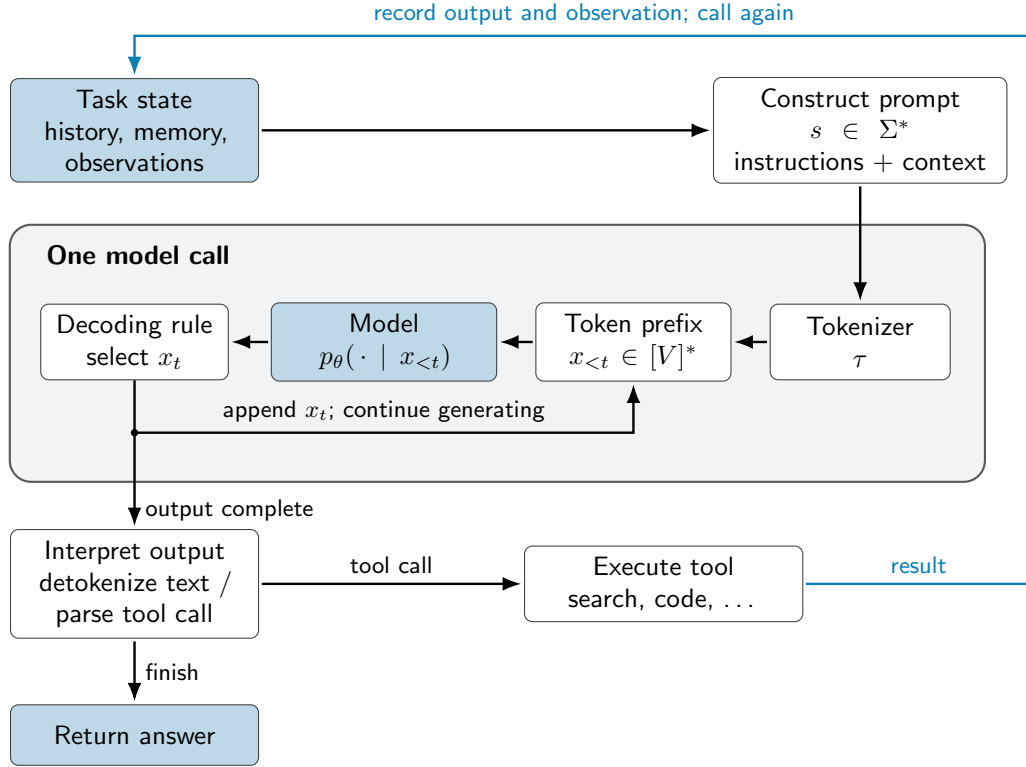


Figure 1: The model inside a harness. Within a call, decoded tokens extend the prefix until generation stops. The harness interprets the output and may execute a tool, record its result, and construct another prompt before returning an answer. The blue feedback path carries the tool observation back into the maintained task state. The call boundary is schematic: some interfaces include tool use and further generation within a single model call. These variations can be convenient; they preserve the basic loop, so we use this schematic throughout the note.

1.2 Tokenization and the modeled sequence

The word *tokenization* comes from the compiler pipeline. A compiler’s lexer maps a character stream to tokens such as identifiers, numerals, and operators using a language specification. An LLM tokenizer has the same broad string-to-sequence interface. Corpus data normally determines its vocabulary and segmentation rules; a programming-language specification determines a compiler’s lexer.

Let Σ be a finite *base alphabet* and let the *vocabulary* $\mathcal{V}$ consist of V nonempty strings:

$$\mathcal{V} = \{v_1, \dots, v_V\} \subset \Sigma^+. \quad (1.1)$$

To give the model a fixed set of token choices, we segment the text into vocabulary items and represent each by its index. A *lossless tokenizer* τ preserves the original text under concatenation:

$$\tau : \Sigma^* \longrightarrow [V]^*, \quad \tau(s) = (x_1, \dots, x_T), \quad s = v_{x_1} v_{x_2} \cdots v_{x_T}. \quad (1.2)$$

Detokenization concatenates the vocabulary strings. Several token sequences may concatenate to the same string, so retokenizing a detokenized sequence need not recover that sequence. Because

every vocabulary item is nonempty,

$$|s| = \sum_{t=1}^T |v_{x_t}| \geq T. \quad (1.3)$$

Thus tokenization, understood as segmentation over a fixed base alphabet, cannot increase sequence length and decreases it whenever at least one chosen token contains more than one base symbol. Normalization, conversion from Unicode characters to bytes, and the addition of special tokens may change the sequence before or after this segmentation; those are separate operations.

Practical tokenizers may also normalize text, first split it into coarse pieces, or begin from bytes to guarantee coverage. Tokens can represent short words, word stems, syllable-like fragments, or other character sequences; their boundaries need not have linguistic significance. Tokenization depends on surrounding text: prepending characters, words, or even a space can change the segmentation of existing text, rather than simply insert extra tokens. The extent of this change depends on the tokenizer’s boundary rules. Appendix A describes how tokenizers may be constructed from a corpus.

For sequence modeling, we augment the vocabulary with special control tokens and continue to denote its index set by $[V]$. Once the tokenizer is fixed, a *probabilistic sequence model* p_θ , with parameters θ , over the token vocabulary $[V]$ is a map from finite token sequences to probability distributions over the next token:

$$\begin{aligned} p_\theta : [V]^* &\longrightarrow \Delta_V, & x_{<t} &\longmapsto p_\theta(\cdot \mid x_{<t}), \\ \Delta_V &:= \left\{ p \in [0, 1]^V : \sum_{v=1}^V p_v = 1 \right\}. \end{aligned} \quad (1.4)$$

We introduce a special *beginning-of-sequence token* $x_0 = \text{BOS}$ to mark the beginning of a sequence. It serves as the initial prefix, allowing the same next-token model to specify the first-token distribution $p_\theta(\cdot \mid x_0)$. We write $x_{<t} = (x_0, \dots, x_{t-1})$; $p_\theta(v \mid x_{<t})$ is the probability assigned to token v after this prefix. Repeatedly drawing from these distributions defines an infinite *stochastic process* $(X_t)_{t \geq 1}$ on $[V]$. The chain rule gives a fixed prefix’s probability by multiplying its successive conditional probabilities:

$$p_\theta(x_{1:T}) = \prod_{t=1}^T p_\theta(x_t \mid x_{<t}). \quad (1.5)$$

Since we are interested in modeling probability distributions over *finite* (as opposed to infinite) token sequences, we introduce another special token, the *end-of-sequence token* EOS. The idea is that once the EOS appears, generation stops. From the infinite process $(X_t)_{t \geq 1}$, we stop at its first EOS occurrence, the *stopping time* τ_{EOS} :

$$\tau_{\text{EOS}} = \inf\{t \geq 1 : X_t = \text{EOS}\}, \quad (1.6)$$

with $\tau_{\text{EOS}} = \infty$ if EOS never occurs. For this construction to define a probability distribution over finite sequences, one needs the additional constraint that $\Pr(\tau_{\text{EOS}} < \infty) = 1$. Deployed systems additionally enforce a maximum generation length M , stopping at $\min\{\tau_{\text{EOS}}, M\}$. This guarantees finite output, potentially truncating it before the model produces EOS.

1.3 Decoder-only transformers as next-token models

The preceding section defined a probabilistic sequence model as a map from token prefixes to next-token distributions, (1.4). We now describe decoder-only transformers, a class of neural networks

that power many of today’s most capable AI systems. Their computations take place in real vector spaces: the token prefix is first embedded as a sequence of real vectors, this sequence is transformed through several layers, and a *readout* converts the last vector into a next-token distribution.

A central design constraint is that the number of trainable parameters is independent of the input sequence length: the same parameters are reused across positions and supported sequence lengths. One trained model can thus handle inputs of different lengths. Since training data are limited, learning shared operations from examples at many positions can reduce the data needed for good generalization. This *inductive bias* assumes that similar computations are useful across positions and lengths. Longer inputs require more computation and activation storage, but no additional trainable parameters.

We use an archetypical architecture built by composing a few reusable functions. Specific architectures vary in their component maps and in how those maps are arranged. *Attention* enriches each position’s representation by collecting relevant information from other positions; a *positionwise feed-forward network* (FFN) combines and transforms the features available at each position. Normalization and residual addition organize these operations into blocks. [Figure 2](#) represents the map from tokens to probabilities as a *computation graph*. Nodes and labels specify operations and intermediate quantities; arrows show which quantities each operation needs. An operation can be evaluated once its inputs are available, and branching reuses a quantity in several computations.

Sequences of activation vectors. Throughout the course, every vector is a column vector. We represent a sequence $(h_1, \dots, h_T)$ of vectors in $\mathbb{R}^d$ by a matrix H whose rows store their transposes:

$$H = \begin{bmatrix} h_1^\top \\ \vdots \\ h_T^\top \end{bmatrix} \in \mathbb{R}^{T \times d}, \quad h_i = (H_{i,:})^\top. \quad (1.7)$$

Thus positions run left to right in a sequence display and top to bottom in a matrix. Columns index feature coordinates.

For a computation local to one position, we reuse the same rule throughout the sequence. A function $f : \mathbb{R}^d \rightarrow \mathbb{R}^r$ therefore has a *positionwise lift*, denoted by the same symbol:

$$f(H) := \begin{bmatrix} f(h_1)^\top \\ \vdots \\ f(h_T)^\top \end{bmatrix} \in \mathbb{R}^{T \times r}. \quad (1.8)$$

The same function, with the same parameters, is applied at every position. It may mix coordinates within a vector, but it does not exchange information between positions.

Attention and positionwise operations. A *normalization map* $N_\nu : \mathbb{R}^d \rightarrow \mathbb{R}^d$ has learned parameters ν and adjusts the scale of a vector’s feature coordinates; some choices also center them. This makes the following learned update less sensitive to the overall activation scale. The FFN computes a map $\text{FF}_\psi : \mathbb{R}^d \rightarrow \mathbb{R}^d$ with learned parameters ψ . We lift both maps positionwise using (1.8), keeping their parameters fixed across positions. LayerNorm and RMSNorm are examples of N_ν ; see [Section 1.4.3](#). Examples of FFN architectures appear in [Section 1.4.2](#).

An *attention head* $\text{Head}_{\eta,M}$ gathers one input-dependent weighted combination of source information at each position. We first specify its interface: for each sequence length T , it computes a map

$$\text{Head}_{\eta,M} : \mathbb{R}^{T \times d} \longrightarrow \mathbb{R}^{T \times d_h}. \quad (1.9)$$

Here η collects its learned parameters, whose number is independent of T ; the same parameters are reused at every position and sequence length. The output width is d_h , and the mask M specifies which source positions each output position may use. A causal mask allows position i to use only positions $1, \dots, i$, ensuring that a next-token prediction uses only information available in its prefix. This interface does not yet specify the computation. [Section 1.4.1](#) defines a head using just three learned matrices, with $3dd_h$ parameters in total.

Several heads allow a position to gather different kinds of information in parallel. An *output projection* W^O combines their retrieved features into an update at the model width. For h heads, collect their parameters and a learned output projection¹ as $\xi = (\eta_1, \dots, \eta_h, W^O)$, where $W^O \in \mathbb{R}^{hd_h \times d}$. The *multi-head attention map* $\text{SelfAttn}_{\xi, M} : \mathbb{R}^{T \times d} \rightarrow \mathbb{R}^{T \times d}$ is defined by

$$\text{SelfAttn}_{\xi, M}(U) = [\text{Head}_{\eta_1, M}(U) \mid \dots \mid \text{Head}_{\eta_h, M}(U)] W^O. \quad (1.10)$$

The brackets concatenate the head outputs into a $T \times hd_h$ matrix; W^O mixes their features and returns to width d . Usually $d_h = d/h$, so $hd_h = d$. The heads receive the same input and mask, with distinct parameters. The mask M is prescribed, separate from the learned parameters ξ .

Blocks and their composition. A block refines the current representation through *residual connections*: each sublayer adds an update to its input. A near-identity map then requires only a small update, a useful bias when composing many layers.

Fix component architectures and a block parameter collection $\vartheta = (\xi, \psi, \nu_1, \nu_2)$. One *pre-normalized decoder block* is the function $\mathcal{B}_{\vartheta, M} : \mathbb{R}^{T \times d} \rightarrow \mathbb{R}^{T \times d}$ defined in two steps. First, normalize the input H to control the scale seen by attention, gather information from the sequence, and add it to the unnormalized input to obtain G :

$$G = H + \text{SelfAttn}_{\xi, M}(N_{\nu_1}(H)). \quad (1.11)$$

Next, the FFN processes these enriched features separately at each position, again using normalization before the update and a residual addition after it:

$$\mathcal{B}_{\vartheta, M}(H) = G + \text{FF}_{\psi}(N_{\nu_2}(G)). \quad (1.12)$$

Every term has shape $T \times d$, so the updates can be added positionwise.

Repeating blocks lets later attention computations use information gathered and processed by earlier blocks, allowing a computation to proceed in stages. Let H^0 be the embedded input sequence. A transformer uses L copies of this block architecture, with parameter collections $\theta_\ell = (\xi_\ell, \psi_\ell, \nu_{\ell,1}, \nu_{\ell,2})$ that generally differ between layers:

$$H^{\ell+1} = \mathcal{B}_{\theta_\ell, M}(H^\ell). \quad \ell = 0, \dots, L-1, \quad (1.13)$$

Alternatively, one can write the computation as the composition of the L underlying maps:

$$H^L = (\mathcal{B}_{\theta_{L-1}, M} \circ \dots \circ \mathcal{B}_{\theta_0, M})(H^0). \quad (1.14)$$

Each layer has its own trainable parameters θ_ℓ , shared across all positions.

We keep parameters explicit in definitions and when distinguishing component copies. In a calculation or diagram for a fixed block, we may write

$$\text{SelfAttn}_M = \text{SelfAttn}_{\xi, M}, \quad \text{FF} = \text{FF}_{\psi}, \quad N_j = N_{\nu_j} \quad (j = 1, 2).$$

¹Following the literature we use the word projection to denote a linear map, even when this map does not have the properties of a mathematical projection, hoping that this will not cause confusion. If this difference becomes important, we will point it out.

We may also shorten $\mathcal{B}_{\vartheta,M}$ to $\mathcal{B}_M$, or write $\mathcal{B}_{\ell,M}$ for $\mathcal{B}_{\theta_\ell,M}$ when a layer index identifies the parameters. These abbreviations are used only when the suppressed parameters are fixed and unambiguous.

For a prefix $x_{<t}$ of length $T = t$, the readout converts the last hidden vector $h_t^L = (H_{t,:}^L)^\top$ into vocabulary scores $z_t \in \mathbb{R}^V$, called *logits*, and then into $p_t := p_\theta(\cdot \mid x_{<t})$ by softmax. A separate decoding rule selects the next token from p_t . Each block reuses its parameters at every position and for every input sequence, an extensive form of *weight sharing*.

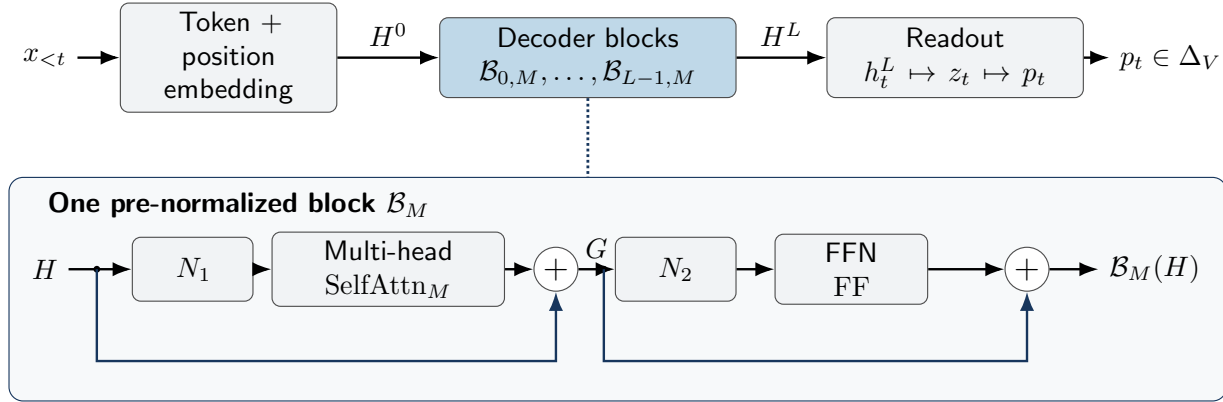


Figure 2: An archetypical decoder transformer as a composition of maps. A prefix $x_{<t}$ is embedded as H^0 , transformed by L decoder blocks, and read out from the last row of H^L to produce logits z_t and the next-token distribution p_t . The inset expands one block: N_1, N_2 are positionwise normalization maps, SelfAttn_M exchanges information across positions allowed by mask M , and the feed-forward network (FFN) applies FF independently at each position. Blue paths carry the inputs to the two residual additions. All block inputs and outputs have shape $T \times d$. The labels abbreviate $\mathcal{B}_{\ell,M} = \mathcal{B}_{\theta_\ell,M}$ and suppress the inset’s fixed component parameters. Different blocks may have different parameters; within a block, parameters are shared across positions and sequence lengths.

The input embedding. The first stage of Figure 2 assigns a trainable vector of features to each token. Learning these vectors lets the model choose a representation suited to prediction. Identify token $x_t \in [V]$ with the standard basis vector $e_{x_t} \in \{0, 1\}^V$. For T prediction positions, define $Z \in \{0, 1\}^{T \times V}$ by

$$Z_{t,:} = e_{x_{t-1}}^\top, \quad t \in [T]. \quad (1.15)$$

Rows of Z index sequence positions and columns index vocabulary items: $Z_{t,v} = 1$ precisely when $x_{t-1} = v$. An *embedding table* $E \in \mathbb{R}^{V \times d}$ assigns the feature vector $E^\top e_v = (E_{v,:})^\top$ to token v . Multiplying by Z performs the lookup at every position:

$$(ZE)_{t,:} = e_{x_{t-1}}^\top E.$$

Word order also changes meaning: “the dog bites the man” and “the man bites the dog” describe different events. We make position information available to subsequent computations by adding a position vector $P_{t,:}^\top$ to each token embedding; attention-based alternatives appear in Section 1.4.5. Stacking these absolute-position vectors as rows of $P \in \mathbb{R}^{T \times d}$ gives the input

$$H^0 = ZE + P, \quad h_t^0 = (H_{t,:}^0)^\top = E^\top e_{x_{t-1}} + P_{t,:}^\top \in \mathbb{R}^d. \quad (1.16)$$

Rows of E index vocabulary items and columns index model features; rows of P and H^0 store the transposes of position and hidden vectors. The one-hot notation makes the map explicit; an implementation performs a row lookup in E .

Position vectors may be computed by a fixed rule. With learned *absolute position embeddings*, P consists of the first T rows of a table $P_{\max} \in \mathbb{R}^{T_{\max} \times d}$, requiring $T \leq T_{\max}$. This table contributes $T_{\max}d$ parameters, depending on the chosen maximum context length $T_{\max}$, not the current input length T .

The matrix H^0 starts the composition in (1.14); Section 1.5 specifies its final readout.

Remark 1.3.1 (Position information when $P = 0$). With $P = 0$, $h_t^0 = E^\top e_{x_{t-1}}$, so a token has the same input vector at every position. The causal mask also carries ordering information through its prefix restriction, so $P = 0$ alone does not make the whole decoder insensitive to order.

1.4 The component maps

We now define the functions used in Figure 2, starting with the operation that transfers information between positions. A component's parameters are held fixed while its input sequence varies. Layer indices are needed only when we assemble different copies of the components and as such we will suppress them in this section.

1.4.1 One attention head

Attention enriches the information available at a sequence position by collecting information from other positions according to its relevance to the current position (Figure 3). For example, in a coding task, a variable use may need information from its earlier declaration. We follow one row of the figure, then collect all rows into a matrix computation.

Let $U \in \mathbb{R}^{T \times d}$ be the input, with activation $u_i = (U_{i,:})^\top$ at position i .

Queries, keys, and values. Each input activation vector u_i supplies three vectors: a *query vector* q_i describing what to look for, a *key vector* k_i used for matching, and a *value vector* v_i containing what can be retrieved. Separate learned projections W^Q, W^K, W^V let matching and retrieval use different features. In a coding task, for example, keys might encode variable names while values carry type information. The head's parameters are

$$\eta = (W^Q, W^K, W^V), \quad W^Q, W^K, W^V \in \mathbb{R}^{d \times d_h}. \quad (1.17)$$

At query position i and source position j , these projections give

$$q_i = (W^Q)^\top u_i, \quad k_j = (W^K)^\top u_j, \quad v_j = (W^V)^\top u_j \in \mathbb{R}^{d_h}.$$

The same activation u_j supplies both k_j and v_j ; the same projection matrices are reused at every position. This is *self-attention*: queries, keys, and values all come from the same input sequence.

Relevance scores. A dot product combines matches across feature coordinates into an *attention score* S_{ij} for source j at query position i . Dividing by the square root of the head width compensates for typical score growth with width at initialization:

$$S_{ij} = \frac{q_i^\top k_j}{\sqrt{d_h}}.$$

Distinct query and key projections allow *directional matching*: the relevance of one position to another need not be symmetric. Remark 1.4.1 explains the scaling heuristic.

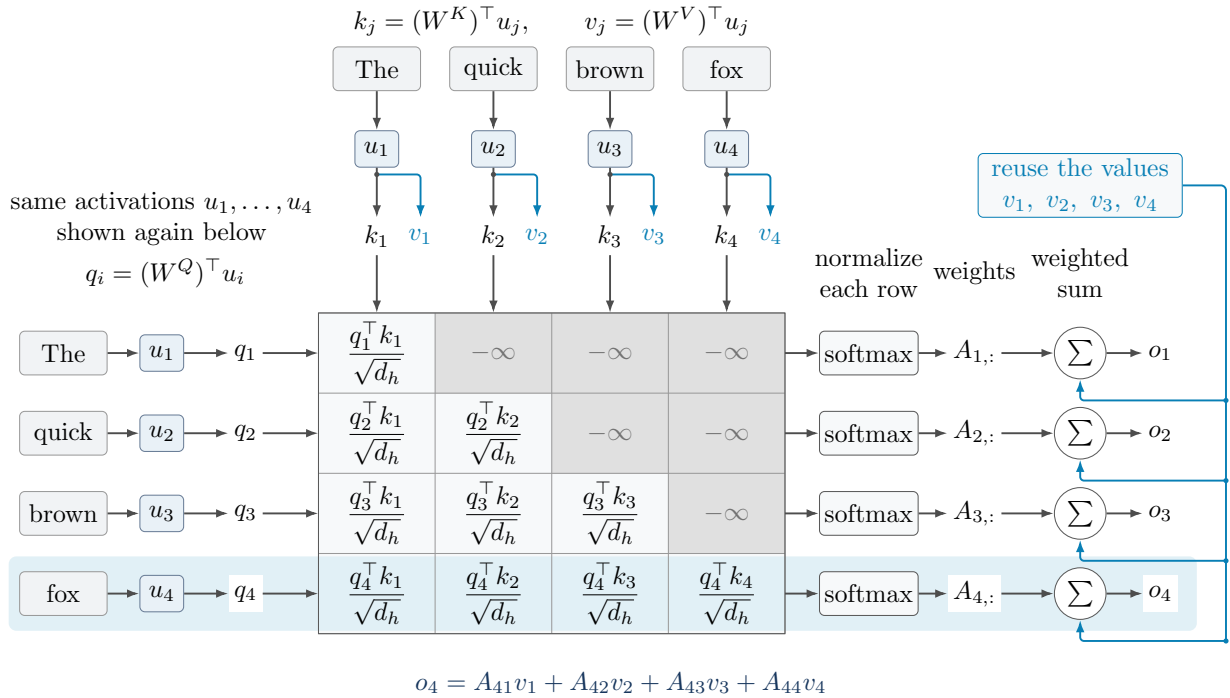


Figure 3: One causal attention head on “The quick brown fox,” assuming one token per word for illustration (spaces and control tokens are omitted). Each word labels its position’s input activation u_j ; in deeper layers this already contains information from earlier positions. Shared projections compute a key k_j , a value v_j , and a query q_j at every position. Matching u_j boxes at the top and left show the same activations; repeated v_j labels likewise denote reused values. The grid displays $S + M$: each cell compares its row’s query with its column’s key; gray cells mask future positions with $-\infty$. Softmax acts on a whole row, producing weights $A_{i,:}$ with zeros at masked positions. The sum node uses these weights and the values supplied by the blue arrows to compute $o_i = \sum_{j \leq i} A_{ij}v_j$. The highlighted “fox” row can retrieve from all four positions. Keys and values are computed once per position and reused across queries; the learned matrices W^Q, W^K, W^V are shared across positions.

Available positions and weights. For autoregressive prediction, the representation predicting x_i must depend only on $x_{<i}$. During parallel training, later input positions contain x_i and subsequent tokens, so reading them would reveal the answer. *Causal attention* enforces the prefix restriction: position i may use only positions $1, \dots, i$, excluding future positions. We enforce this restriction by adding the prescribed *causal mask* M to the scores:

$$M_{ij} = \begin{cases} 0, & j \leq i, \\ -\infty, & j > i. \end{cases} \quad (1.18)$$

Softmax converts each row's available scores into nonnegative *attention weights* A_{ij} summing to one. Larger scores receive larger weights, while $e^{-\infty} = 0$ removes masked positions:

$$A_{ij} = \begin{cases} \frac{e^{S_{ij}}}{\sum_{r=1}^i e^{S_{ir}}}, & j \leq i, \\ 0, & j > i. \end{cases}$$

Its smooth dependence on the finite scores allows gradient-based training. For *bidirectional attention*, every position can use all T sources: set $M = 0$ and normalize over all of them. Directional matching and causal masking are separate choices: even without a mask, query–key scores need not be symmetric.

Collecting the values. The weights specify how much information to take from each available source. Their weighted sum produces the output o_i at position i :

$$o_i = \sum_{j=1}^i A_{ij} v_j \in \mathbb{R}^{d_h}. \quad (1.19)$$

Because the weights are nonnegative and sum to one, this is a weighted average. The highlighted row in [Figure 3](#) shows the case $i = 4$.

All positions at once. Stack the transposes of q_i, k_i, v_i, o_i as rows of Q, K, V, O . Their rows index positions and their columns index head features; in S and A , rows index queries and columns index sources. The following matrix computation summarizes the head and evaluates all query rows together:

$$\begin{aligned} Q &= UW^Q, & K &= UW^K, & V &= UW^V, & Q, K, V &\in \mathbb{R}^{T \times d_h}, \\ S &= \frac{QK^\top}{\sqrt{d_h}}, & & & & & S &\in \mathbb{R}^{T \times T}, \\ A &= \text{softmax}_{\text{row}}(S + M), & & & & & A &\in \mathbb{R}^{T \times T}, \\ O &= AV, & & & & & O &\in \mathbb{R}^{T \times d_h}. \end{aligned} \quad (1.20)$$

This defines $\text{Head}_{\eta, M}(U) = O$, with $o_i = (O_{i,:})^\top$. The $T \times T$ score and weight matrices are computed activations, not additional parameters. Each key and value is computed once and reused across query rows; the formulas reuse W^Q, W^K, W^V at every sequence length.

Remark 1.4.1 (The score-scaling heuristic). Consider a simple random-initialization model. Treat u_i, u_j as fixed inputs with $\|u_i\|_2^2, \|u_j\|_2^2 \approx d$: their mean squared coordinates are about one, the scale targeted by the normalization examples in [Section 1.4.3](#) before learned rescaling. Initialize all

entries of W^Q, W^K independently with mean zero and variance $1/d$. This choice keeps projected coordinates at the same scale:

$$\text{Var}(q_{ir}) = \sum_{a=1}^d u_{ia}^2 \text{Var}(W_{ar}^Q) = \frac{\|u_i\|_2^2}{d} \approx 1, \quad \text{Var}(k_{jr}) = \frac{\|u_j\|_2^2}{d} \approx 1.$$

All variances here concern the random initialization. The two projections are independent, so $q_{ir}k_{jr}$ has mean zero and variance $\text{Var}(q_{ir}) \text{Var}(k_{jr}) \approx 1$. Different coordinates use independent projection columns, making these products independent across r . Summing their variances gives

$$\text{Var}(q_i^\top k_j) \approx d_h, \quad \text{Var}\left(\frac{q_i^\top k_j}{\sqrt{d_h}}\right) \approx 1.$$

The scaling compensates for growth in the initial score variance, helping avoid overly concentrated softmax weights and small softmax gradients merely because the head is wider. Training can change both independence and scale; this argument motivates the initialization scale without bounding learned scores.

Remark 1.4.2 (Masks and available source positions). A mask can equivalently be specified by the positions available to each query. For every $i \in [T]$, let $\emptyset \neq S_i \subseteq [T]$ be those source positions. The corresponding matrix mask has $M_{ij} = 0$ for $j \in S_i$ and $M_{ij} = -\infty$ otherwise. Given real scores s_{ij} for $i, j \in [T]$, applying rowwise softmax after adding this mask gives weights α_{ij} :

$$\alpha_{ij} = \begin{cases} \frac{e^{s_{ij}}}{\sum_{r \in S_i} e^{s_{ir}}}, & j \in S_i, \\ 0, & j \notin S_i. \end{cases} \quad (1.21)$$

Thus we can compute attention by summing over available positions directly. The causal mask uses $S_i = [i]$. For a positive integer width w , a *causal window mask* uses $S_i = \{j \in [i] : i - j < w\}$, retaining only the most recent w positions, including the query position. Near the start of a sequence, fewer than w positions are available.

1.4.2 The positionwise feed-forward network

The FFN combines and transforms the features available at one position, including information collected by attention. This lets later attention heads match using newly computed features, and ultimately helps form the features used for prediction. The FFN applies the same learned function $\text{FF}_\psi : \mathbb{R}^d \rightarrow \mathbb{R}^d$ independently at each position. Its architecture can be an MLP, a gated network, or a mixture of experts. We use ψ for its learned parameters.

A *multilayer perceptron* (MLP) is a feed-forward network built from learned affine maps and coordinatewise activation functions. For a two-layer example, fix the parameters

$$\psi = (W^1, W^2, b^1, b^2), \quad (1.22)$$

$$W^1 \in \mathbb{R}^{d \times d_{\text{ff}}}, \quad W^2 \in \mathbb{R}^{d_{\text{ff}} \times d}, \quad b^1 \in \mathbb{R}^{d_{\text{ff}}}, \quad b^2 \in \mathbb{R}^d.$$

For an input $u \in \mathbb{R}^d$, the first affine map forms a vector a of d_{ff} feature combinations:

$$a = (W^1)^\top u + b^1 \in \mathbb{R}^{d_{\text{ff}}}.$$

A nonlinear *activation function* $\phi : \mathbb{R} \rightarrow \mathbb{R}$ transforms these combinations into hidden features z . Nonlinearity matters because composing affine maps alone would yield another affine map. Apply ϕ coordinatewise:

$$z = \phi(a), \quad z_r = \phi(a_r).$$

Common choices include *ReLU*, $\text{ReLU}(s) = \max\{0, s\}$, and *GELU*, $\text{GELU}(s) = s\Phi(s)$, where Φ is the standard normal cumulative distribution function. Finally, a second affine map combines the new features into an update at the model width:

$$\text{FF}_\psi(u) = (W^2)^\top z + b^2 = (W^2)^\top \phi((W^1)^\top u + b^1) + b^2. \quad (1.23)$$

To apply this computation to all positions at once, lift it using (1.8). Write $\mathbf{1}_n \in \mathbb{R}^n$ for the all-ones column vector; the outer products below copy each bias into every row:

$$\text{FF}_\psi(R) = \phi(RW^1 + \mathbf{1}_T(b^1)^\top)W^2 + \mathbf{1}_T(b^2)^\top, \quad R \in \mathbb{R}^{T \times d}. \quad (1.24)$$

For example, row i of $RW^1 + \mathbf{1}_T(b^1)^\top$ is $((W^1)^\top r_i + b^1)^\top$, where $r_i = (R_{i,:})^\top$, so the matrix formula applies (1.23) independently at each position.

Remark 1.4.3 (Mixture of experts). A *mixture-of-experts* (MoE) FFN combines expert maps $f_1, \dots, f_m : \mathbb{R}^d \rightarrow \mathbb{R}^d$ using input-dependent routing. A sparse router selects only a few experts to evaluate at each position, allowing a large collection of learned parameters without using every expert on every input. For an input $r \in \mathbb{R}^d$, the router selects a small subset $S(r)$ and nonnegative weights $\rho_e(r)$ summing to one over that subset. Combine only the selected outputs:

$$f_{\text{MoE}}(r) = \sum_{e \in S(r)} \rho_e(r) f_e(r). \quad (1.25)$$

The positionwise lift applies this same router and expert collection at every position. Here $\text{FF}_\psi = f_{\text{MoE}}$, and ψ collects all expert and router parameters. This choice of FFN leaves the attention map unchanged.

Remark 1.4.4 (SwiGLU). For an input $r \in \mathbb{R}^d$, a *gate* $g(r)$ lets one learned set of features modulate another, $v(r)$, according to the input. For *SwiGLU* with hidden width m , fix $W_g, W_v \in \mathbb{R}^{m \times d}$ and $W_o \in \mathbb{R}^{d \times m}$. The projection W_v supplies the features; W_g , followed by the nonlinear *swish* function, supplies the gate:

$$v(r) = W_v r, \quad g(r) = \text{swish}(W_g r), \quad \text{swish}(s) = \frac{s}{1 + e^{-s}},$$

where *swish* acts coordinatewise. Multiply each feature by its gate value using the coordinatewise product $\odot$, defined by $(a \odot b)_i = a_i b_i$, then combine the products into an output of width d :

$$f_{\text{SwiGLU}}(r) = W_o(g(r) \odot v(r)) = W_o(\text{swish}(W_g r) \odot (W_v r)). \quad (1.26)$$

Here $\text{FF}_\psi = f_{\text{SwiGLU}}$ and $\psi = (W_g, W_v, W_o)$.

1.4.3 Normalization maps

Successive updates can change the magnitude of a position's activation. Normalization controls the scale supplied to attention and the FFN, reducing their sensitivity to such changes. For a fixed normalization rule, $N_\nu : \mathbb{R}^d \rightarrow \mathbb{R}^d$ denotes its map with learned parameters ν . These parameters are separate from those of attention and the FFN.

Remark 1.4.5 (Example: LayerNorm-type normalization). For an input $u \in \mathbb{R}^d$, first decide whether to remove a common offset across features. Let $\mu(u)$ be its coordinate mean. A centering choice $c \in \{0, 1\}$ gives the vector $\tilde{u}_c$, subtracting the mean when $c = 1$:

$$\mu(u) = \frac{1}{d} \sum_{r=1}^d u_r, \quad \tilde{u}_c = u - c\mu(u)\mathbf{1}_d.$$

Next, form the normalized vector $\hat{u}_c$ by dividing by the root mean square, with a fixed $\varepsilon > 0$ to prevent division by zero:

$$\hat{u}_c = \frac{\tilde{u}_c}{\sqrt{d^{-1}\|\tilde{u}_c\|_2^2 + \varepsilon}}.$$

Finally, learned scales $\gamma \in \mathbb{R}^d$ and shifts $\beta \in \mathbb{R}^d$ let the model adjust individual features. The resulting map is

$$N_{c,\gamma,\beta}(u) = \gamma \odot \hat{u}_c + \beta = \gamma \odot \frac{\tilde{u}_c}{\sqrt{d^{-1}\|\tilde{u}_c\|_2^2 + \varepsilon}} + \beta. \quad (1.27)$$

Standard *LayerNorm* centers the features and learns both scale and shift; *RMSNorm* omits centering and the shift:

$$\text{LayerNorm}_{\gamma,\beta} = N_{1,\gamma,\beta}, \quad \text{RMSNorm}_{\gamma} = N_{0,\gamma,0}. \quad (1.28)$$

When $c = 1$, the denominator uses the coordinate variance; when $c = 0$, it uses the mean square. For LayerNorm, $\nu = (\gamma, \beta)$ and $N_\nu = N_{1,\gamma,\beta}$. For RMSNorm, which is used in Llama 3, $\nu = \gamma$ and $N_\nu = N_{0,\gamma,0}$. The centering choice and ε are fixed.

The lift in (1.8) normalizes each position separately across its features: $(N_\nu(H)_{i,:})^\top = N_\nu(h_i)$. Within a fixed block, $N_1 = N_{\nu_1}$ and $N_2 = N_{\nu_2}$ abbreviate its two normalization maps, with separate learned parameter collections ν_1, ν_2 .

1.4.4 Layers and shared parameters

We now assemble copies of these maps in the stack (1.14). At layer ℓ , head a has parameters $\eta_{\ell,a}$, the attention output projection is W_ℓ^O , the FFN has parameters ψ_ℓ , and the normalizations have parameters $\nu_{\ell,1}, \nu_{\ell,2}$. Thus the block's learned parameters are

$$\xi_\ell = (\eta_{\ell,1}, \dots, \eta_{\ell,h}, W_\ell^O), \quad \theta_\ell = (\xi_\ell, \psi_\ell, \nu_{\ell,1}, \nu_{\ell,2}). \quad (1.29)$$

Thus layer ℓ uses $\text{SelfAttn}_{\xi_\ell, M}$, FF_{ψ_ℓ} , and $N_{\nu_{\ell,1}}, N_{\nu_{\ell,2}}$. The same component formulas apply in every layer, with each layer's own parameter collection. When the layer index suffices, we may write $\text{SelfAttn}_M^\ell = \text{SelfAttn}_{\xi_\ell, M}$ and $\text{FF}^\ell = \text{FF}_{\psi_\ell}$.

Figure 4 expands the block graph in Figure 2 across four positions and two layers. Let g_i^ℓ be the vector at position i after the attention residual update: it is the transpose of row i of G in (1.11), applied at layer ℓ . The operation nodes compute

$$g_i^\ell = \mathcal{A}_{\ell,i}(h_1^\ell, \dots, h_i^\ell), \quad h_i^{\ell+1} = \mathcal{F}_{\ell,i}(g_i^\ell).$$

Here $\mathcal{A}_{\ell,i}$ includes multi-head attention, the first normalization, and the first residual update; $\mathcal{F}_{\ell,i}$ includes the FFN, second normalization, and second residual update from (1.12). Their parameter collections are

$$\vartheta_{\ell,1} = (\xi_\ell, \nu_{\ell,1}), \quad \vartheta_{\ell,2} = (\psi_\ell, \nu_{\ell,2}).$$

Solid arrows show the causal dependencies: $\mathcal{A}_{\ell,i}$ reads positions $1, \dots, i$, while $\mathcal{F}_{\ell,i}$ reads only g_i^ℓ . Dashed blue branches show reuse of one parameter collection across positions. Adding positions adds computation nodes and activations, with no new parameter collections.

Remark 1.4.6 (Shared parameters and changing activations). Within one head, $q_i = (W^Q)^\top u_i$ uses the same matrix at every position. The FFN and normalization likewise reuse their parameters across positions. Activations depend on the input and position; longer sequences require more computation and activation storage while reusing the same block parameters.

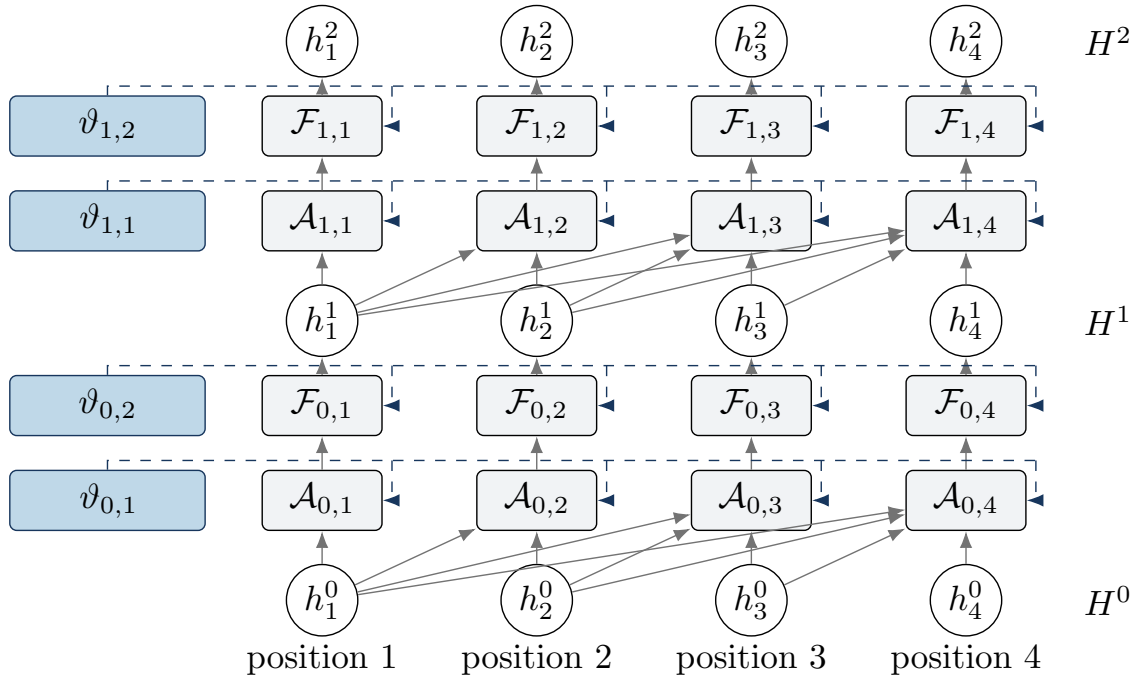


Figure 4: The block graph of Figure 2 expanded across four positions and two layers, read bottom to top. Circles are vectors h_i^ℓ ; their transposes form the rows of H^ℓ . $\mathcal{A}_{\ell,i}$ computes causal multi-head attention and its residual update at position i , including the first normalization; its output is g_i^ℓ . Each head inside this operation follows Figure 3. $\mathcal{F}_{\ell,i}$ normalizes g_i^ℓ , applies the FFN, and adds the result to g_i^ℓ , producing $h_i^{\ell+1}$. Solid arrows carry activations. Dashed blue branches supply one shared parameter collection $\vartheta_{\ell,1}$ or $\vartheta_{\ell,2}$ to all positions in its sublayer. Crossings are not junctions. Attention heads and intermediate vectors are suppressed.

Causal attention uses only preceding and current positions, and all other block operations act positionwise. Consequently, after every layer the vector h_t^ℓ depends only on the token prefix $x_{<t}$. [Section 1.13](#) uses the block notation to compare encoder-only, decoder-only, and encoder–decoder architectures.

1.4.5 Position mechanisms inside attention

A query may need the value three positions back, or a content match among nearby positions. Positional mechanisms let attention express such preferences. For one head, a *relative-position bias* matrix B assigns an additive preference to each displacement $i - j$. A function ρ clips or buckets the displacement, and b assigns a learned scalar to each bucket:

$$A = \text{softmax}_{\text{row}} \left(\frac{QK^\top}{\sqrt{d_h}} + M + B \right), \quad (B)_{ij} = b(\rho(i - j)). \quad (1.30)$$

Rows index query positions and columns index key positions, so $i - j$ compares output position i with source position j . T5 uses this form with the bias parameters shared across layers and different across heads. ALiBi uses a fixed, head-dependent linear penalty such as $-m(i - j)$ for a causal pair $j \leq i$. *Rotary position embeddings* (RoPE) modify the attention calculation by applying position-dependent linear transformations—specifically, rotations—to the query and key vectors before taking their inner product. The query–key dot product then depends explicitly on the displacement between their positions (see [Exercise 1](#) for details).

Choosing the frequencies. For head width $d_h = 2m$, coordinate pair $r \in [m]$ rotates through $i\omega_r$ radians at position i . Its *angular frequency* ω_r is the angle added per position; we call it simply a *frequency*. Standard RoPE fixes a geometric schedule:

$$\omega_r = b_{\text{RoPE}}^{-(r-1)/m}, \quad r \in [m], \quad b_{\text{RoPE}} > 1. \quad (1.31)$$

The original base was $b_{\text{RoPE}} = 10,000$. Fast rotations help distinguish nearby offsets; slow ones support content comparisons stable over longer distances. Increasing the base slows rotations for $r > 1$, trading local resolution for stability over longer distances.

What does RoPE learn, and what repeats? Training learns the query and key projections. The projected vectors’ magnitudes and relative directions control each frequency’s contribution to the score. Contributions can reinforce one another at a desired displacement and disagree elsewhere. Because queries and keys depend on the input, these preferences can depend on content; separate heads learn separate comparisons. Applying the rotations costs linear work in the head dimension.

One frequency gives periodic dependence on displacement, with a possibly noninteger period. Combining frequencies can avoid short repetitions, but distant displacements can still have similar rotations, so long-context retrieval is not guaranteed. RoPE also does not enforce locality: prescribed causal and *window masks* exclude future and distant positions, respectively, as in Mistral 7B. [Exercise 1](#) examines retrieval and periodicity; the bibliographical notes discuss trained models.

1.4.6 Architecture examples

Figures 5 and 6 instantiate our modular calculation for two dense decoder-only models with different normalization, FFNs, attention heads, and position mechanisms. They show the full residual paths and operation placement, including dropout. Numerical labels give architectural dimensions, not input-dependent activation shapes.

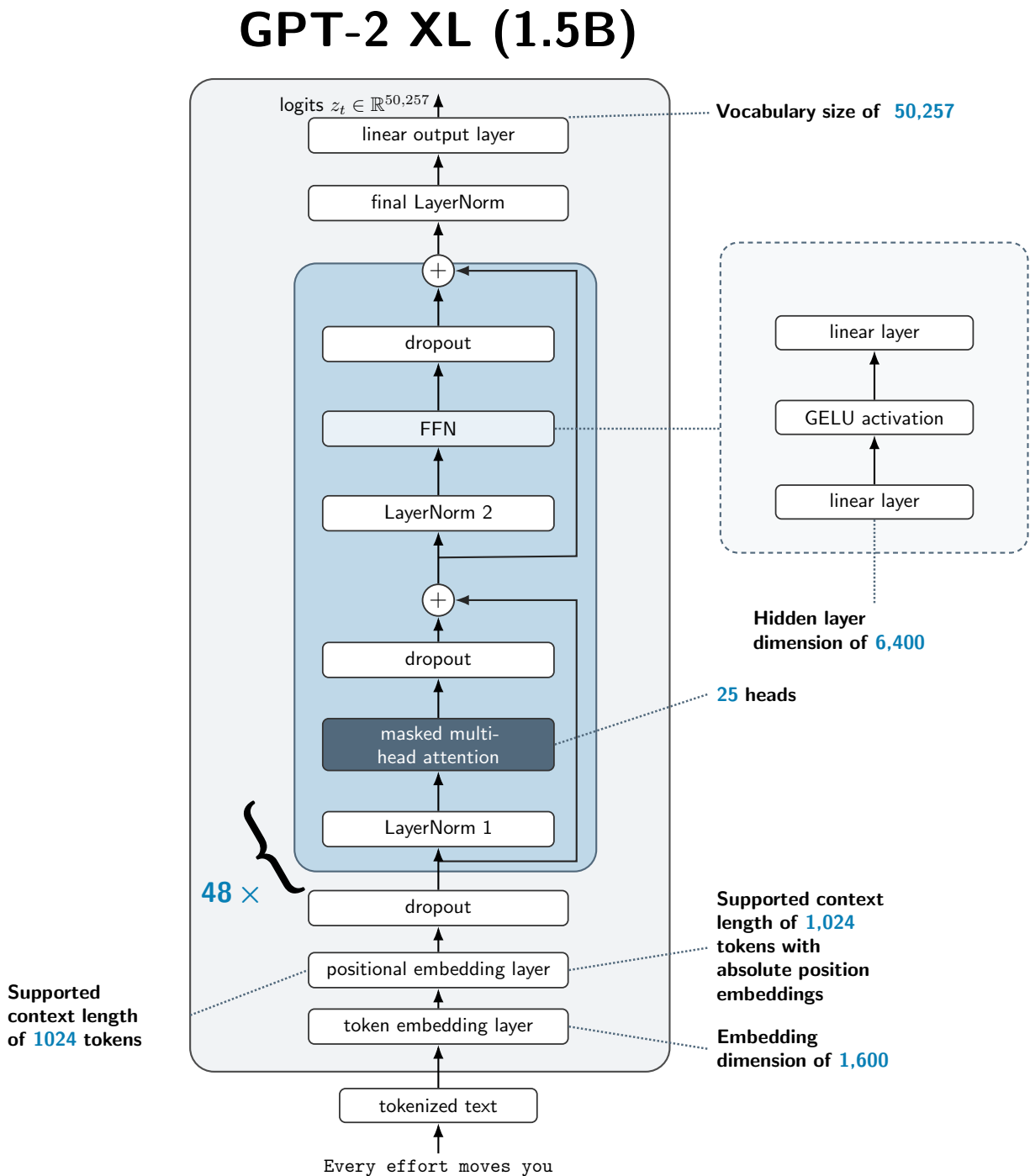


Figure 5: The GPT-2 XL decoder stack. The central blue region is one pre-normalized transformer block, repeated 48 times; the two side paths are its residual connections. The inset expands the feed-forward network (FFN), a two-layer MLP, into its two affine maps and GELU activation.

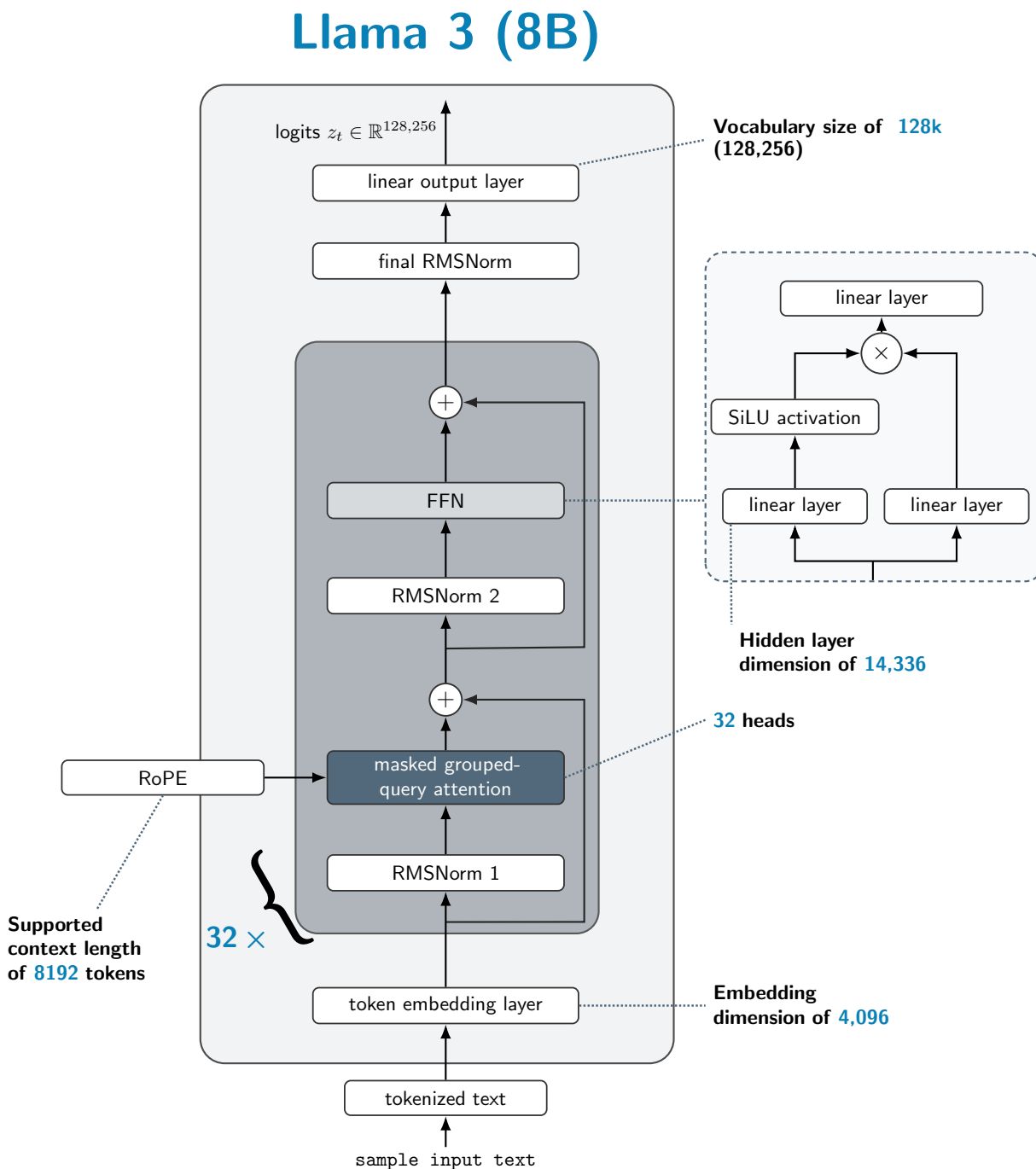


Figure 6: The Llama 3 8B decoder stack. The central gray region is one pre-normalized transformer block, repeated 32 times. RoPE modifies the query and key vectors inside grouped-query attention. The inset expands the feed-forward network (FFN), showing both SwiGLU branches, their coordinatewise product, and the output projection. The 32 attention heads are query heads, grouped over 8 key/value heads.

1.4.7 Composing attention computations

An addressing rule may involve several conditions. For example, in a coding task, we might want to find an earlier occurrence of an identifier in the same scope and retrieve its associated value. This requires representing the identifier and scope, comparing candidate sources, and retrieving information from a suitable match. How could the components of a transformer divide this work?

Within one head, the query–key dot product adds contributions from vector components. If different groups of components represent identifier agreement and scope agreement, their contributions can jointly favor sources satisfying both conditions. The representations and score margins must make a joint match beat a partial match. The positionwise FFNs can transform and combine features to prepare them for such comparisons in later layers.

Different heads have separate attention distributions and can retrieve different information in parallel. Their outputs are then combined. Retrieving an identifier match in one head and a scope match in another, however, does not establish that the same source satisfies both conditions. A computation requiring that conjunction must test the conditions together or retain enough information to check that the retrieved sources coincide.

Layers allow a later query to depend on information retrieved earlier. In the coding example, one layer could retrieve a relevant declaration, and a later layer could use information from that declaration to locate an associated value. The residual connections preserve information that subsequent computations may also need. These are possible ways to organize the computation; how training distributes the work among components, heads, and layers is a further question.

RoPE can add positional conditions to these content-based comparisons. [Exercise 1](#) isolates retrieval at a fixed relative offset and examines the required width and score scale. More complex addressing also depends on the available heads and layers. The bibliographical notes introduce languages for studying how such computations can be organized.

1.5 From a hidden vector to the next-token distribution

We now return to the full computation in [Figure 2](#) and complete its final step: converting the last hidden vector $h_t^L = (H_{t,:}^L)^\top \in \mathbb{R}^d$ into a next-token distribution. The readout first assigns one score, or logit, $z_{t,v}$ to each vocabulary item v . A shared output matrix $W_U \in \mathbb{R}^{V \times d}$ and bias $b_U \in \mathbb{R}^V$ define this affine map:

$$z_t = W_U h_t^L + b_U \in \mathbb{R}^V. \quad (1.32)$$

Each row of W_U specifies how the model features contribute to one token’s score. Softmax then converts the scores into next-token probabilities $p_{t,v} = p_\theta(x_t = v \mid x_{<t})$, favoring higher scores while ensuring positive probabilities summing to one:

$$p_{t,v} = \frac{\exp(z_{t,v})}{\sum_{u=1}^V \exp(z_{t,u})}. \quad (1.33)$$

The score differences have a direct probabilistic meaning: $\log(p_{t,v}/p_{t,u}) = z_{t,v} - z_{t,u}$. Adding the same constant to every logit leaves the probabilities unchanged. For two outcomes, with $p = p_{t,v}$, this difference is the binary *logit*, or *log-odds*, $\log(p/(1-p))$. The full map $h \mapsto \text{softmax}(W_U h + b_U)$ is the *softmax readout*: it converts a hidden representation into a distribution over output tokens. The architecture examples also apply a final normalization N_{out} ; their readout is $h \mapsto \text{softmax}(W_U N_{\text{out}}(h) + b_U)$. [Lecture 3](#) develops logistic and softmax regression and their log-loss objectives.

Remark 1.5.1 (Weight tying). The same token features can be used both to represent an input token and to score it as a possible output, reducing the number of learned parameters. This *weight tying* sets $W_U = E$: the input embedding matrix also serves as the output readout matrix. If $u_v = E^\top e_v \in \mathbb{R}^d$ is the embedding of token v , then its output logit is

$$z_{t,v} = u_v^\top h_t^L + b_{U,v}. \quad (1.34)$$

The same vector represents token v when it is read and scores it as a possible next token. Sharing the matrices replaces two independently learned $V \times d$ matrices by one, saving Vd parameters. This is an architectural choice; the input and output matrices can also be learned separately.

The single-token path is therefore

$$e_{x_{t-1}} \longrightarrow E^\top e_{x_{t-1}} \longrightarrow h_t^L \longrightarrow z_t \longrightarrow p_t. \quad (1.35)$$

The contextual vector h_t^L also depends on the earlier tokens through the masked attention blocks.

1.6 Autoregressive generation

In *autoregressive generation*, each selected token extends the prefix used to predict the next. To sample from the sequence model, draw the next token X_t from its conditional distribution:

$$X_t \sim p_\theta(\cdot \mid X_{<t}), \quad (1.36)$$

Alternatively, *greedy decoding* chooses a most probable next token:

$$X_t \in \arg \max_v p_\theta(v \mid X_{<t}). \quad (1.37)$$

In either case the selected token is appended and the procedure repeats until a stopping rule fires.

Greedy decoding is neither sampling from the model nor, in general, global optimization of the probability of the complete sequence. A locally most probable token can lead to poor later continuations.

There are narrow cases in which greedy decoding is justified. If the system must make exactly one token-valued decision and incurs zero–one loss, an argmax token is Bayes optimal. Greedy decoding is also harmless when every conditional distribution is effectively deterministic, or when a special problem has a greedy-choice property. None of these conditions holds in general for open-ended generation.

Remark 1.6.1 (Greedy decoding). For open-ended multi-token generation, forget greedy decoding: it is a bad idea. It throws away nearly all of the predictive distribution at every step, because a locally best token can irreversibly eliminate much better continuations.

[Exercise 2](#) examines how greedy decoding can produce endless repetition even when sampling usually yields a short, complete sentence.

1.7 Attention cost and the key–value cache

Generation extends the prefix one token at a time. Which computations need to be repeated when a new position is added to [Figure 3](#)? The first full forward pass over a supplied prompt is called *prefill*. It computes the prompt’s activations and the distribution for its next token. After selecting and appending that token, a *decode step* computes the next distribution. Caching lets this step advance only the new position.

What can be reused? Every earlier *layer-position activation*—the vector already computed at a particular layer and position—remains unchanged: its input embedding is unchanged, causal attention cannot read the new position, and normalization and the FFN act positionwise. Applying this argument layer by layer shows that all earlier rows of each head’s input U , and hence of Q, K, V , are unchanged.

Fix one layer and head, suppressing their indices. Appending position t adds a bottom query row and a rightmost key column to the grid in [Figure 3](#). The new column is masked above its diagonal, so only the new bottom row of scores and weights needs evaluation. Store the earlier keys and values as rows of a *key-value cache*, $K_{<t}, V_{<t} \in \mathbb{R}^{(t-1) \times d_h}$, for positions $1, \dots, t-1$. The new query row uses these keys for its scores and these values for its weighted sum; earlier queries and attention weights are not needed.

At position t , project the new input activation u_t to obtain q_t, k_t, v_t as before. Append the new key and value to form the matrices for the enlarged prefix:

$$K = \begin{bmatrix} K_{<t} \\ k_t^\top \end{bmatrix}, \quad V = \begin{bmatrix} V_{<t} \\ v_t^\top \end{bmatrix} \in \mathbb{R}^{t \times d_h}. \quad (1.38)$$

The new query row $Q_{t,:} = q_t^\top$ compares with all these keys. Normalize its scores to obtain just the last row of A :

$$A_{t,:} = \text{softmax}_{\text{row}} \left(\frac{Q_{t,:} K^\top}{\sqrt{d_h}} \right) \in \mathbb{R}^{1 \times t}.$$

Use these weights to combine the cached values into the new output row:

$$O_{t,:} = A_{t,:} V = \sum_{j=1}^t A_{tj} v_j^\top \in \mathbb{R}^{1 \times d_h}. \quad (1.39)$$

This evaluates only the last row of (1.20), the new query row in [Figure 3](#). All cached positions are available to it, so no additional causal mask is needed in this row formula.

Combine the new outputs of all heads, complete the residual, normalization, and FFN operations at this position, and pass the new vector to the next layer. Each layer and head maintains its own cache. Caching changes which intermediate results are recomputed; it preserves the model’s next-token distribution. With RoPE, cache the position-transformed keys actually used in the query-key comparisons; their positions remain fixed as the prefix grows.

Attention work and storage. A full forward pass on a length- t prefix computes every row. The number of permitted query-key pairs per causal head is

$$\sum_{i=1}^t i = \frac{t(t+1)}{2}, \quad (1.40)$$

so computing the scores and weighted sums costs $\Theta(t^2 d_h)$ arithmetic. A cached decode step computes just t scores and one weighted sum, costing $\Theta(t d_h)$. Its cache holds $2t d_h$ real numbers after t positions; the projections and positionwise maps are evaluated only for the new position. Recomputing each growing prefix would repeat the full-pass cost. For successive prefix lengths

$t = 1, \dots, T$, the attention work per head is

$$\begin{aligned} \text{without caching: } & \sum_{t=1}^T \Theta(t^2 d_h) = \Theta(T^3 d_h), \\ \text{with caching: } & \sum_{t=1}^T \Theta(t d_h) = \Theta(T^2 d_h). \end{aligned} \tag{1.41}$$

A longer initial prompt contributes its prefill cost; subsequent decode steps start at that prefix length.

A straightforward full-pass implementation materializes a $t \times t$ attention matrix. Skipping the masked upper triangle can save nearly half the score and attention–value arithmetic compared with the t^2 pairs of bidirectional attention. This applies when processing all positions together, as in training, prefill, and full-sequence scoring. Computing the full square and masking afterward gives no arithmetic saving. The whole-block saving is smaller because projections, normalization, and the FFN remain.

Remark 1.7.1 (Why are queries not cached?). The query q_i was used to compute the output at position i . A future position $t > i$ forms a new query q_t and compares it with the old key k_i ; the resulting weight multiplies the old value v_i . Thus future computation reuses k_i and v_i . The old query q_i has completed its role, so caching it would consume memory without avoiding future computation.

1.8 Context length and a bounded-score limitation

A model call receives a *context window*: the sequence consisting of the prompt followed by any tokens generated so far. Practical systems come with a *maximum context length*, an upper bound on the number of tokens in the context window. Position handling, training range, cache memory, attention cost, and implementation choices can all constrain the maximum context length. The ability to accept T tokens and the ability to use all T tokens effectively are distinct capabilities.

Making full use of a long context window can require sharply selecting a particular position. For one query q attending to T keys $k_1, \dots, k_T$, denote its scores by s_j and its weights by α_j :

$$s_j = \frac{q^\top k_j}{\sqrt{d_h}}, \quad \alpha_j = \frac{e^{s_j}}{\sum_{r=1}^T e^{s_r}}. \tag{1.42}$$

If the scores remain in a fixed bounded range as T grows, no individual softmax weight can remain sharply concentrated. [Exercise 3](#) asks you to quantify this limitation and derive its entropy consequence.

The conclusion is conditional on bounded scores; score gaps that grow with context length evade it.

Entropy language. Entropy $H(\alpha)$ measures how spread out a probability vector $\alpha \in [0, 1]^T$ is. For $\sum_{j=1}^T \alpha_j = 1$, define (with $0 \log 0 = 0$)

$$H(\alpha) = - \sum_{j=1}^T \alpha_j \log \alpha_j. \tag{1.43}$$

It is 0 for a deterministic distribution and $\log T$ for the uniform distribution. Thus small entropy describes concentrated attention and large entropy describes diffuse attention.

Remark 1.8.1 (Observed long-context failures). The terms *lost in the middle* and *context rot* refer to observed failures to use long contexts uniformly or reliably, sometimes well below a model’s advertised maximum length.

1.9 A few notes about training

A known sequence supplies many next-token prediction tasks at once:

Input token	x_0 (BOS)	x_1	x_2	x_3
Target token		x_1	x_2	x_3
			x_3	x_4

Providing the known preceding tokens is called *teacher forcing*. All inputs in this table are available before the forward pass. Within each layer, the positions can therefore be processed in parallel. The causal mask ensures that the prediction of x_t depends only on $x_{<t}$, even though later tokens are present elsewhere in the computation. Layers are still processed in order. During generation, the next input token must first be selected from the model’s prediction, so ordinary autoregressive generation proceeds one token at a time. Skipping masked attention pairs is an additional arithmetic saving, as discussed in [Section 1.7](#).

To train, we combine the losses of the next-token predictions and compute their gradient with respect to the parameters. *Backpropagation* applies the chain rule backward through the computation graph. When a parameter is used in several operations, its gradient sums the contributions from all those uses. Thus every dashed branch in [Figure 4](#) contributes to the gradient of the one shared parameter collection it references. An optimizer uses this gradient to update the parameters. The loss will be developed in the lectures on log loss and autoregression; Lecture 27 will examine the backward computation and its implementation on accelerators.

Initial large-scale training, usually conducted on a broad corpus, is called *pretraining*. *Fine-tuning* names the operation of continuing parameter training from an existing checkpoint. *Post-training* names the stage after pretraining and may contain several fine-tuning procedures, including supervised instruction tuning and preference- or reward-based optimization. A supervised fine-tuning stage after pretraining is both fine-tuning and post-training. During *inference*, the resulting parameters are held fixed while the model calls described in this note compute predictions. The mathematics of pretraining will be developed later.

1.10 What exactly is contained in the parameter vector?

The symbol θ collects the trainable numerical parameters of the conditional model. For our archetypical architecture with learned additive position embeddings, the modular accounting is

$$\theta = (E, P_{\max}, W_U, b_U, \theta_0, \dots, \theta_{L-1}), \quad (1.44)$$

where $P_{\max}$ is the full position-embedding table and (1.29) explicitly lists each block’s head, output-projection, FFN, and normalization parameters. With weight tying, E determines $W_U = E$. Fixed position maps and the causal mask M lie outside the learned parameters. For an MoE FFN, ψ_ℓ collects the expert and router parameters. The token vocabulary and segmentation rules are normally constructed before neural-network training and held fixed. They determine V , the input sequences, and the shapes of E and W_U while remaining outside θ in this account.

The token prefix is an input; hidden states, queries, keys, values, attention weights, logits, and the KV cache are computed quantities. All lie outside θ . The decoding rule and the surrounding harness are also outside θ . Thus training the model means choosing the shared numerical parameters in this display; specifying the deployed AI system requires additional choices.

1.11 Hardware, software, and the search for better systems

My first neural-network experiment, in the 1990s, used a Kohonen network with about ten neurons on an IBM PC. I remember an experiment taking perhaps hours. The models whose parameters we have just counted operate at a very different scale: billions of learned parameters. Another experience made the change in computing speed particularly tangible. About 10 years later, we had developed a face-recognition program. Five years later, I found the executable on my computer and ran it out of curiosity. After it took my picture, a dialog box flashed on the screen. I had to look up the source code to find out what it said: “Patience. This can take some time.”

Computing resources change which experiments we can afford and which methods are practical. Hardware and software also coevolve: researchers develop methods that use available hardware well, while hardware and software infrastructure are improved to support successful methods. These investments make further improvements to established designs especially attractive.

This resembles local search: build on a working system, change some components, and retain useful improvements. The architecture we studied reflects this history of choices and refinements. Its success gives us reasons to understand it; it does not establish that the search for better designs is finished. The more successful a technology becomes, the more support it attracts and the harder it may become to replace. An alternative competes with a whole ecosystem of optimized hardware, software, and accumulated expertise. Early experiments may therefore understate what it could achieve with comparable development. Research requires patience and judgment about which limitations are fundamental and which might disappear with scale or better implementation. This feedback between adoption and improvement can lead to technological lock-in: the ecosystem supporting a successful technology makes alternatives harder to adopt.

1.12 Take-home messages

- A deployed AI system places the language model inside a “harness” that constructs context, can maintain state and invoke tools, and may make repeated model calls.
- A lossless tokenizer segments a sequence over a base alphabet into nonempty vocabulary strings. Its token sequence cannot be longer than the base-symbol sequence and is shorter whenever a selected token contains several base symbols. Tokens may lack semantic meaning.
- A probabilistic sequence model maps each finite token prefix to a next-token distribution. Repeated sampling defines a stochastic process; finite-sequence probabilities are products of the next-token conditionals.
- Our archetypical decoder transformer composes embedding, decoder blocks, and a readout. Each block combines causal attention, a positionwise FFN, and normalization maps through residual additions. LayerNorm and RMSNorm are examples of the normalization component.
- Absolute position embeddings are added once at the input. RoPE and relative-position biases instead modify the attention calculation in every layer that uses them.
- The learned conditional distribution and the decoding algorithm are different objects. An argmax is optimal for a single zero-one-loss decision. Repeated tokenwise argmax generally fails to optimize a sequence-level objective and is a poor default for open-ended generation.
- A full dense-attention pass on a length- t prefix has quadratic arithmetic cost. An implementation of causal attention can skip nearly half the attention pairs used by bidirectional attention at the

same length. During sequential generation, earlier keys and values remain unchanged and are cached. Each new query uses those rows plus the new key and value. For a fixed-length initial prompt, this changes total attention work from cubic to quadratic in the generated length. Old queries have no future use.

- Maximum context length is an input-capacity limit. Performance on some retrieval and reasoning tasks may deteriorate before the input reaches this limit. With bounded attention scores, a single softmax head cannot remain sharply concentrated as the context grows.
- The parameter vector θ contains the learned weights of the conditional model. The current activations, KV cache, tokenizer, decoding rule, and harness lie outside θ . Lecture 2 takes up the question of why learning can produce useful values of θ .

1.13 Optional reading: three transformer architecture types

An encoder–decoder model supplies next-token distributions for a target sequence conditioned on a separate source sequence. A decoder-only model supplies next-token distributions for the continuation of one sequence. These define distributions over completed strings when generation terminates with probability one. This section compares these models, together with encoder-only models, through their information flow: which tokens are available simultaneously, and which tokens must be generated from left to right?

Let $x_{1:S} = (x_1, \dots, x_S)$ denote a source sequence. In an encoder–decoder model, let $y_{1:T} = (y_1, \dots, y_T)$ denote the target sequence. Before the model generates target token y_t , it has access to the target prefix

$$y_{<t} := (y_0, y_1, \dots, y_{t-1}), \quad (1.45)$$

where y_0 is a beginning-of-target token. During teacher-forced training, the shifted inputs $y_0, \dots, y_{T-1}$ produce predictions for $y_1, \dots, y_T$ in parallel under a causal mask. The notation $\text{Embed}_{\text{src}}$ and $\text{Embed}_{\text{tgt}}$ refers to the token-and-position embedding construction in (1.16); the two subscripts indicate separately learned embedding parameters.

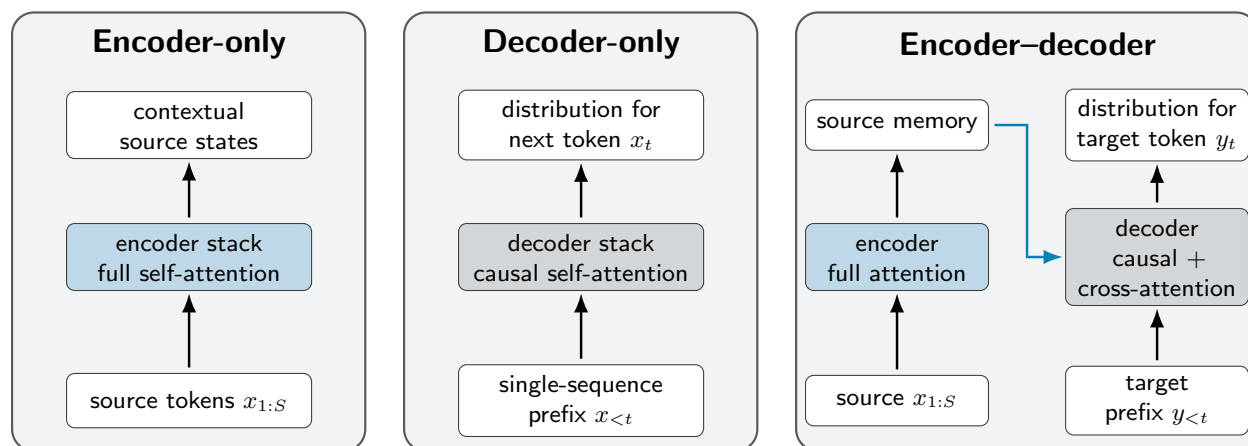


Figure 7: Information flow in three transformer architecture types. Encoder-only models contextualize a fixed input. Decoder-only models predict the continuation of one causal sequence. Encoder–decoder models first encode the complete source and then generate a target that can attend to that source memory.

Encoder-only. An encoder-only model constructs a contextual representation of a fixed input. Write $\mathcal{B}_{\ell,M}^{\text{enc}} = \mathcal{B}_{\theta_{\ell,M}^{\text{enc}}}$ for its block at depth ℓ , with parameters $\theta_{\ell}^{\text{enc}}$. Let $M_{\text{full}} = 0$ allow every position to attend to every other position. The encoder stack is

$$E^0 = \text{Embed}_{\text{src}}(x_{1:S}), \quad E^{\ell+1} = \mathcal{B}_{\ell,M_{\text{full}}}^{\text{enc}}(E^{\ell}), \quad \ell = 0, \dots, L_E - 1. \quad (1.46)$$

Thus every row of $E^{L_E} \in \mathbb{R}^{S \times d}$ can contain information from the entire source sequence. A task-specific readout converts these contextual states into predictions.

Decoder-only. A decoder-only model represents a single sequence and predicts its continuation. Similarly, $\mathcal{B}_{\ell,M}^{\text{dec}} = \mathcal{B}_{\theta_{\ell,M}^{\text{dec}}}$ denotes its independently parameterized block at depth ℓ . With the triangular mask M_{causal} in (1.18),

$$H^0 = \text{Embed}(x_{1:T}), \quad H^{\ell+1} = \mathcal{B}_{\ell,M_{\text{causal}}}^{\text{dec}}(H^{\ell}), \quad \ell = 0, \dots, L - 1. \quad (1.47)$$

At every depth, the state at position i depends only on tokens at positions $[i]$. A prompt and its continuation therefore occupy one causal token stream.

Encoder–decoder. An encoder–decoder model is designed for generating a target sequence conditioned on a separately supplied source sequence. The encoder first computes E^{L_E} using (1.46). This matrix is the *source memory*: it contains one contextual state for each source position and remains fixed while the target is generated. The target decoder starts from

$$D^0 = \text{Embed}_{\text{tgt}}(y_{<t}). \quad (1.48)$$

For the target decoder, fix each component's parameters at depth r and abbreviate $\text{SelfAttn}_M^r = \text{SelfAttn}_{\xi_r^{\text{self}},M}$ and $\text{FF}^r = \text{FF}_{\psi_r}$. The map SelfAttn_M^r is the multi-head self-attention map defined in (1.10); it forms queries, keys, and values from the same matrix U and applies mask M . The map FF^r is computed by the positionwise FFN at depth r ; (1.24) gives the two-layer MLP example.

Cross-attention lets each target position retrieve relevant information from the source memory. For target states $U \in \mathbb{R}^{m \times d}$ and source memory $E \in \mathbb{R}^{S \times d}$, form queries Q from U , and keys K and values V from E . With learned projections $W^Q, W^K, W^V \in \mathbb{R}^{d \times d_h}$, the same score, softmax, and retrieval operations as in self-attention give weights A and outputs O :

$$\begin{aligned} Q &= UW^Q, & K &= EW^K, \quad V = EW^V, \\ A &= \text{softmax}_{\text{row}}\left(\frac{QK^{\top}}{\sqrt{d_h}}\right), & O &= AV. \end{aligned} \quad (1.49)$$

Here $A \in \mathbb{R}^{m \times S}$ and $O \in \mathbb{R}^{m \times d_h}$. The map $\text{CrossAttn}^r(U, E)$ combines several such heads, with separate parameters, by concatenation and an output projection as in (1.10). Its superscript identifies the fixed collection of cross-attention parameters at depth r , separate from ξ_r^{self} and ψ_r . Write $N_{r,j} = N_{\nu_{r,j}}$ for the three positionwise normalization maps, $j \in \{1, 2, 3\}$, each with its own learned parameters. These parameter collections may differ between depths. The decoder block at depth r first mixes information among available target-prefix positions, producing states G^r :

$$G^r = D^r + \text{SelfAttn}_{M_{\text{causal}}}^r(N_{r,1}(D^r)).$$

Cross-attention then enriches these states with information from the source memory, producing J^r :

$$J^r = G^r + \text{CrossAttn}^r(N_{r,2}(G^r), E^{L_E}).$$

Finally, the FFN transforms the features at each target position independently, giving the next layer's states D^{r+1} :

$$D^{r+1} = J^r + \text{FF}^r(N_{r,3}(J^r)). \quad (1.50)$$

Each step normalizes the input to its sublayer and adds the result back through a residual connection.

What changes computationally? For a length- n sequence, the number of permitted query–key pairs is $N_{\text{full}}(n)$ for one full-attention head and $N_{\text{causal}}(n)$ for one causal head. Count all pairs for the former and only lower-triangular pairs for the latter:

$$N_{\text{full}}(n) = n^2, \quad N_{\text{causal}}(n) = \frac{n(n+1)}{2}. \quad (1.51)$$

A triangular attention kernel can avoid computing the masked upper triangle. These expressions count attention pairs per head; they do not include linear projections or FFN computation.

For source length S and target length T , add the pairs used by source self-attention, causal target self-attention, and target-to-source cross-attention. With L_E encoder blocks and L_D encoder–decoder blocks, the total per head is

$$N_{\text{enc-dec}} = L_E S^2 + L_D \frac{T(T+1)}{2} + L_D ST. \quad (1.52)$$

A fully causal decoder-only stack of L blocks on a concatenated source and target permits

$$N_{\text{dec-only}} = L \frac{(S+T)(S+T+1)}{2}. \quad (1.53)$$

Complete runtime comparisons also depend on model depths, widths, head counts, parameter counts, FFN architectures, and the available hardware kernels.

Remark 1.13.1 (Why are decoder-only models common for language modeling?). Next-token pretraining on raw text supplies a prediction target at every position and requires no separate source–target split. Training and generation also use the same single-sequence interface. These properties simplify large-scale training and serving. Encoder–decoder models provide a different advantage for conditional generation: every target token can use a bidirectionally encoded source.

Generation and caching. During encoder–decoder generation, the encoder computes the source memory once. Each decoder layer caches the key and value projections of that fixed memory for cross-attention. It separately grows a causal self-attention cache for the generated target. At step t , the model forms new queries for the current target position and reuses both caches.

1.14 Exercises

General instructions. Justify every answer, including yes/no answers, even when the question does not explicitly ask for a justification.

The following tool may be used in [Exercise 1](#).

Theorem 1.14.1 (Hoeffding's inequality). *Let $n \geq 1$ be an integer and let $Z_1, \dots, Z_n$ be independent real random variables. Suppose $Z_r \in [a_r, b_r]$ almost surely for each $r \in [n]$, where $a_r < b_r$ are real numbers. For every real $t \geq 0$,*

$$\mathbb{P}\left(\sum_{r=1}^n (Z_r - \mathbb{E}[Z_r]) \geq t\right) \leq \exp\left(-\frac{2t^2}{\sum_{r=1}^n (b_r - a_r)^2}\right).$$

1. **Relative addressing with RoPE.** Can a query request a value at a specified relative offset while leaving the source keys unchanged? We will construct such queries, identify collisions between addresses, and examine how additional coordinate pairs help distinguish them.

Let $T \geq 2$ and $d_v \geq 1$ be integers. Position $j \in [T]$ stores an arbitrary fixed value $v_j \in \mathbb{R}^{d_v}$. Every source uses the same unrotated unit key $k \in \mathbb{R}^2$. For each integer offset $1 \leq d < T$, we seek a unit-length query vector $q(d) \in \mathbb{R}^2$ that retrieves v_{i-d} at every query position $i \in [T]$ with $i > d$. Reusing this query at different positions should implement the same addressing rule, independently of the stored values. As described earlier, RoPE modifies the attention calculation by applying position-dependent linear transformations—specifically, rotations—to the query and key vectors before taking their inner product.

- (a) **Encoding the request.** For a real angle ϕ , define

$$R_\phi = \begin{pmatrix} \cos \phi & -\sin \phi \\ \sin \phi & \cos \phi \end{pmatrix}.$$

Fix an *angular frequency* $\omega > 0$ (called simply a *frequency* below): moving forward one position increases the rotation angle by ω radians. RoPE compares the query at position i with the key at position j using the score

$$s_{ij} = (R_{i\omega} q(d))^\top (R_{j\omega} k), \quad i, j \in [T].$$

- (i) Show that $R_\alpha^\top R_\beta = R_{\beta-\alpha}$ for real angles α, β .
 - (ii) Find the unique unit vector $q(d)$ so that $s_{i,i-d} = 1$ whenever $i > d$, and derive a formula for s_{ij} . Show that no source has a larger score.
 - (iii) Verify that shifting i and j by the same integer amount leaves the score unchanged whenever both positions remain in $[T]$.
- (b) **From scores to retrieval.** For the scores constructed in (a), let $\emptyset \neq S_i \subseteq [T]$ be the source positions available to query i . At positive real scale λ , define the attention weights and output by

$$a_{ij}(\lambda) = \frac{e^{\lambda s_{ij}}}{\sum_{r \in S_i} e^{\lambda s_{ir}}}, \quad j \in S_i, \quad o_i(\lambda) = \sum_{j \in S_i} a_{ij}(\lambda) v_j \in \mathbb{R}^{d_v}.$$

Fix $\varepsilon \in (0, 1/2)$.

- (i) For any $j \in S_i$, show that

$$\|o_i(\lambda) - v_j\|_2 \leq (1 - a_{ij}(\lambda)) \max_{r \in S_i} \|v_r - v_j\|_2.$$

Thus $a_{ij}(\lambda) > 1 - \varepsilon$ gives an $O(\varepsilon)$ retrieval error for fixed values. More explicitly, for a positive real bound B , show that $\|v_r\|_2 \leq B$ for all $r \in S_i$ gives error less than $2B\varepsilon$.

- (ii) Show that guaranteeing error at most $2B\varepsilon$ for every such collection of values requires $a_{ij}(\lambda) \geq 1 - \varepsilon$.
- (iii) Use $S_i = [i]$ and suppose $0 < \omega < 2\pi/T$. Show that $i - d$ is the unique maximizing source at every query position $i > d$. For each such i , show that $\lim_{\lambda \rightarrow \infty} o_i(\lambda) = v_{i-d}$, accounting for the weight at every source position.
- (iv) At any such position, prove that $a_{i,i-d}(\lambda) \geq 1 - \varepsilon$ requires

$$\lambda \geq \frac{T^2}{2\pi^2} \log \frac{1 - \varepsilon}{\varepsilon}.$$

Hence, for fixed ε , uniform retrieval over values of norm at most B requires $\lambda = \Omega(T^2)$ in this construction. Reducing the frequency makes neighboring scores increasingly close; increasing λ compensates for this shrinking gap. The next part examines collisions when the frequency is held fixed as the sequence grows, and how a window removes them.

- (c) **Collisions and a window repair.** For this part, hold d fixed and allow the sequence length T to grow. Choose an integer period $P > d$, set $\omega = 2\pi/P$, and use the query construction from (a) with $S_i = [i]$. A *collision* occurs when another source has the same score as the requested source $i - d$.
- (i) Characterize all colliding source positions.
 - (ii) Find the first query position $i > d$ at which a collision occurs.
 - (iii) Evaluate $\lim_{\lambda \rightarrow \infty} o_i(\lambda)$ at the first collision position.
 - (iv) Now use the window $S_i = \{j \in [i] : i - j < w\}$, where w is a positive integer. Find the smallest width that retains $i - d$. For this width, prove that $i - d$ is the unique maximizing source and that $\lim_{\lambda \rightarrow \infty} o_i(\lambda) = v_{i-d}$ for every $i > d$, however long the sequence becomes.
- (d) **Reliable addressing at a fixed score scale.** Close scores can require high numerical precision to distinguish, even when λ is large. Can additional coordinate pairs create better-separated scores and permit a smaller scale? Increasing λ is equivalent to increasing query and key norms. To compare this with increasing width, we fix $\lambda = 1$ and unit query and key norms within each pair.

Fix an integer context length $T \geq 2$. Throughout this part, use $S_i = [i]$. Determine, up to constant factors, how many coordinates are necessary and sufficient to give the requested source attention weight at least $3/4$, simultaneously for every integer offset $1 \leq d < T$ and every query position $i \in [T]$ with $i > d$.

Let m be a positive integer, which determines the number of coordinate pairs used. For each coordinate pair $r \in [m]$, use a fixed unit key $k_r \in \mathbb{R}^2$, a frequency $\omega_r \in [0, 2\pi)$, and the unit query $q_r(d) \in \mathbb{R}^2$ constructed in (a). Define the combined score as

$$s_{ij} = \sum_{r=1}^m (R_{i\omega_r} q_r(d))^\top (R_{j\omega_r} k_r), \quad i, j \in [T].$$

- (i) **A necessary dimension.** Show that all scores lie in $[-m, m]$. Use [Exercise 3](#) to prove that, at query position $i = T$, attention weight at least $3/4$ on the requested source requires

$$m \geq \frac{1}{2} \log(3(T-1)).$$

- (ii) **Finding frequencies that separate addresses.** Prove that, for every integer $m \geq 8 \log(2T)$, there exist frequencies $\omega_1, \dots, \omega_m \in [0, 2\pi)$ such that

$$\sum_{r=1}^m \cos(\omega_r h) \leq m/2 \quad \text{for every integer } h \text{ with } 1 \leq |h| < T. \quad (1.54)$$

- (iii) **Retrieval at every position and offset.** Fix frequencies satisfying [Equation \(1.54\)](#). Show that the requested source has score m and that every other available source has score at most $m/2$, simultaneously for all offsets and query positions in the question. Prove that the total attention weight outside the requested source is at most $(T-1)e^{-m/2}$. Deduce that $m = \lceil 8 \log(2T) \rceil$ suffices for target weight at least

$3/4$ with $\lambda = 1$. State the resulting error bound when every value vector has norm at most a positive real bound B . Together with (i), conclude that $\Theta(\log T)$ coordinate pairs are necessary and sufficient.

Conclusion. With suitably chosen frequencies, RoPE supports relative addressing with logarithmic width and bounded coordinates. Extra coordinate pairs both distinguish addresses and accumulate the score advantage needed to concentrate attention ([Exercise 3](#)). Keeping each coordinate bounded still allows the total score to grow: width supplies an alternative to increasing λ .

Standard RoPE instead uses geometrically spaced frequencies to provide both rapidly and slowly varying positional features. The existence guarantee in (d) does not automatically apply to that schedule. Indeed, its frequencies satisfy $0 < \omega_r \leq 1$, so at displacement $h = 1$,

$$\sum_{r=1}^m \cos \omega_r \geq m \cos 1 > m/2.$$

Thus it fails the particular sufficient separation condition in (d)(ii); this does not rule out retrieval with that schedule.

HINTS For (d)(i), apply the bound from [Exercise 3](#) with T sources and score range $2m$. For (d)(ii), choose the frequencies independently and uniformly on $[0, 2\pi)$. Compute $\mathbb{E}[\cos(\omega_r h)]$ for a nonzero integer h , then apply [Theorem 1.14.1](#) and a union bound over $h \in [T - 1]$. Use the evenness of cosine to handle negative h .

2. **Greedy decoding and repetition.** A toy language model completes the prefix “The movie was”. Let “**very**” and “**good.**” (including the full stop, not including the apostrophes, which we will drop) be two tokens in the model’s output token vocabulary. Given this prefix, the toy model outputs token **very** with probability 0.6 or token **good.** with probability 0.4. When the prefix ends with token **very**, it repeats the same choice; when it ends with **good.**, it outputs the stop token with probability one and generation stops. Count the stop token as a generated token, including for the budget below.

- What token sequence does greedy decoding produce with a budget of 10 additional tokens? What token sequence does it produce without a budget?
- Under sampling, calculate the probability that generation terminates within 10 additional tokens. Without a token budget, show that generation terminates with probability one and compute the expected number of generated tokens.
- Which complete sentence has the highest probability under the model? What is its probability?

Connection to language models. [Welleck et al. \(2019, Table 4, p. 11\)](#) show a GPT-2 continuation that repeats “Portuguese” under greedy decoding. The [Qwen3-8B model card](#) also warns that greedy decoding can cause endless repetition in thinking mode.

3. **Bounded scores and attention concentration.** Let $T \geq 2$ be an integer and let $a, b \in \mathbb{R}$ satisfy $a \leq b$. Write $R = b - a$ for the score range. For a score vector $s \in [a, b]^T$, define the probability vector $\alpha(s)$ by

$$\alpha_j(s) = \frac{e^{s_j}}{\sum_{r=1}^T e^{s_r}}, \quad j \in [T].$$

- (a) Compute $\sup_{s \in [a,b]^T} \max_{j \in [T]} \alpha_j(s)$ and give a score vector attaining this value.
- (b) *Optional extension.* Writing $H(\alpha) = -\sum_{j=1}^T \alpha_j \log \alpha_j$ for the *entropy*, show that for every $s \in [a,b]^T$,

$$\log T - R \leq H(\alpha(s)) \leq \log T.$$

Deduce that, for fixed a, b , $H(\alpha(s))/\log T \rightarrow 1$ uniformly over $s \in [a,b]^T$ as $T \rightarrow \infty$. This concerns entropy relative to its maximum; it does not imply $\sum_{j=1}^T |\alpha_j(s) - 1/T| \rightarrow 0$.

- (c) For a fixed $c \in (0, 1)$, determine the smallest $R \geq 0$ for which some $s \in [a, a + R]^T$ satisfies $\max_{j \in [T]} \alpha_j(s) \geq c$. Give such a score vector at this minimum range. Describe how this minimum range grows with T for fixed c , and deduce whether a score range independent of T can maintain $\max_j \alpha_j(s) \geq c$ as $T \rightarrow \infty$.

Glossary

Harness

The surrounding program that constructs model inputs, calls the model, interprets outputs, invokes tools, and controls further model calls.

Task state

Information maintained by the harness between calls, such as conversation history, tool results, and progress on the task. Only the portion supplied to the model is visible to that call.

Prompt

The initial input supplied to a model call. In the text-only setting, it is a string that tokenization converts into the initial token sequence.

Prefix

The token sequence $x_{<t}$ conditioning the prediction at position t . During generation, it consists of the tokenized prompt followed by the tokens generated so far.

Context

Information available to the model for its current prediction, represented here by the token prefix. External task state influences the prediction only through what the harness supplies.

Context window

The span of tokens available to the model at a prediction step. In the full-prefix setting of this note, it contains the prompt and the tokens generated so far, subject to the maximum context length.

Maximum context length

The supported upper bound on the number of tokens in the context window. Prompt and generated tokens share this capacity. Accepting a context of this length does not guarantee effective use of every token.

Base alphabet; vocabulary; token

The base alphabet Σ contains raw symbols, such as bytes. The vocabulary $\mathcal{V} = \{v_1, \dots, v_V\}$ contains token strings over Σ . A token is one vocabulary item, represented inside the model by its index in $[V]$; it need not be a whole word.

Tokenizer; detokenizer

A tokenizer maps a string to a sequence of vocabulary indices. The detokenizer maps $(x_1, \dots, x_T)$ to $v_{x_1} \cdots v_{x_T}$ in the text-only setup. It undoes lossless tokenization; the reverse composition need not recover the original token sequence.

Sequence orientation; matrix layout

In a sequence display, positions run left to right. In the matrices here, positions are stacked as rows and the columns index coordinates such as vocabulary items or learned features. Every vector is a column vector; a matrix row that stores a vector contains its transpose.

Stochastic process

A jointly distributed collection of random variables (X_t) indexed by time or sequence position.

Probabilistic sequence model; stochastic sequence model

A map $p_\theta : [V]^* \rightarrow \Delta_V$ assigning a next-token distribution to each finite token prefix; see [Equation \(1.4\)](#).

Autoregressive model

A model specified through next-token conditional distributions $p_\theta(x_t \mid x_{<t})$. The probability of a fixed finite sequence is their product, as in [Equation \(1.5\)](#).

Pretraining; post-training; fine-tuning

Initial large-scale fitting on broad data; the stage of training that follows pretraining; and continued parameter training from an existing checkpoint. A post-training pipeline may contain several fine-tuning procedures.

Inference

Computing predictions with the model parameters held fixed. Inference can score a supplied sequence as well as support generation.

Generation

Repeatedly computing a next-token distribution, selecting a token by a decoding rule, and appending it to the prefix until a stopping rule applies.

Encoder-only; decoder-only; encoder–decoder transformer

A bidirectional stack that produces contextual representations of an input; a causal stack that places prompt and continuation in one sequence; and a two-stack architecture in which a causal decoder separately reads the representation produced by a bidirectional encoder.

Self-attention

Attention whose queries, keys, and values are computed from the same sequence of activations, with a mask specifying which source positions are available.

Cross-attention

Attention whose queries come from one sequence and whose keys and values come from another. Here decoder states query the encoder’s source states; see [Equation \(1.49\)](#).

Transformer block

One repeated layer combining attention and a positionwise feed-forward network with residual connections and normalization. An encoder–decoder’s decoder block also contains cross-attention.

Embedding

The learned vector assigned to a token: for token v , it is $E^\top e_v \in \mathbb{R}^d$. The same token uses the same input embedding at each occurrence; subsequent hidden vectors also depend on context.

Logit

For a binary probability $p \in (0, 1)$, the logit is the log-odds $\text{logit}(p) = \log(p/(1-p))$. In a multiclass model $p = \text{softmax}(z)$, the real-valued scores z_v are called logits: their differences satisfy $z_v - z_u = \log(p_v/p_u)$. A common additive shift of all logits leaves p unchanged. See [Section 1.5](#) and Lecture 3 on logistic and softmax regression.

Softmax

The map from real scores $z \in \mathbb{R}^V$ to probabilities $p_v = \text{softmax}(z)_v = e^{z_v} / \sum_{u=1}^V e^{z_u}$. In attention, the same map normalizes the scores over the available source positions for each query.

Categorical distribution

A distribution on finitely many outcomes, represented here by $p \in [0, 1]^V$ with $\sum_{v=1}^V p_v = 1$, where p_v is the probability of token v .

Readout; output head

A map from a hidden representation to output predictions. The *softmax readout* here is $h \mapsto \text{softmax}(W_U h + b_U)$: a learned affine map produces logits, and softmax converts them to token probabilities.

Parameter

A trainable numerical quantity in θ , such as an embedding, projection weight, or bias. Its value is held fixed during the inference computations described here.

Activation; layer-position activation

An intermediate value computed for the current input, such as a hidden vector, query, key, or value. The layer-position activation $h_i^\ell = (H_{i,\cdot}^\ell)^\top$ is the hidden vector at position i after layer ℓ .

Normalization map

A vector-to-vector map $N_\nu : \mathbb{R}^d \rightarrow \mathbb{R}^d$ with learned parameters ν that adjusts feature scale and may also center features. Its positionwise lift acts separately at each sequence position. LayerNorm-type maps are examples; see [Equation \(1.27\)](#).

Layer normalization (LayerNorm)

Subtracting a vector's coordinate mean and dividing by $\sqrt{\text{coordinate variance} + \varepsilon}$, then applying learned coordinatewise scaling and shifting. It acts separately at each sequence position, across features. This is $N_{1,\gamma,\beta}$ in [Equation \(1.27\)](#).

Pre-normalized block

A block that normalizes the input to each attention or FFN sublayer before applying it. The sublayer output is then added to the unnormalized residual stream.

Root-mean-square normalization (RMSNorm)

Dividing u by $\sqrt{d^{-1} \sum_{r=1}^d u_r^2 + \varepsilon}$ and applying learned coordinatewise scaling. It is the choice $N_{0,\gamma,0}$ in [Equation \(1.27\)](#): centering and the shift are omitted.

Parameter sharing

Using the same learned parameters across positions and input sequence lengths, or in different model calls; see [Remark 1.4.6](#).

Teacher forcing

Training next-token predictions using the observed preceding tokens as inputs, rather than feeding back the model’s own generated tokens.

Backpropagation

Applying the chain rule backward through the computation graph to compute derivatives of a loss with respect to parameters and intermediate values. An optimizer uses the parameter gradients to update the model.

Weight tying

Sharing the input embedding matrix with the output readout matrix, $W_U = E$; see [Remark 1.5.1](#).

Activation function

A nonlinear scalar function applied coordinatewise within a neural-network layer, such as ReLU or GELU. This is a function, whereas an activation is a computed value.

ReLU

The rectified linear unit, $\text{ReLU}(z) = \max\{0, z\}$.

GELU

The Gaussian error linear unit, $\text{GELU}(z) = z\Phi(z)$, where Φ is the standard normal cumulative distribution function.

SiLU; swish

The sigmoid linear unit, $\text{SiLU}(z) = z/(1 + e^{-z})$. This is the swish function used in [Remark 1.4.4](#).

SwiGLU

A gated FFN computing the map $W_o(\text{SiLU}(W_g r) \odot (W_v r))$. The two input projections interact by coordinatewise multiplication before the output projection; see [Remark 1.4.4](#).

Attention head

A sequence-to-sequence map $\text{Head}_{\eta, M} : \mathbb{R}^{T \times d} \rightarrow \mathbb{R}^{T \times d_h}$. Its parameters η , whose number is independent of T , specify the query, key, and value projections; the mask M specifies available sources. It computes query–key weights and uses them to combine value vectors; see [Figure 3](#).

Query; key; value

The query q_i is computed from an output position’s activation. The key k_j is computed from a source position’s activation and is compared with q_i to score that source. The value v_j is the source vector used in the attention-weighted sum; its projection is generally distinct from the key projection.

Attention score

A real-valued comparison between query i and source j , here the scaled inner product $q_i^\top k_j / \sqrt{d_h}$, possibly modified by positional rotations or an additive relative-position bias.

Attention weight

The coefficient $\alpha_{ij} = e^{s_{ij}} / \sum_{r \in S_i} e^{s_{ir}}$ for an available source $j \in S_i$, and zero for a masked source. Each query’s weights sum to one; see [Equation \(1.21\)](#).

Causal mask

The restriction $j \leq i$ on sources available to query position i . It is implemented by adding $-\infty$ to future-position scores before softmax, giving those sources zero attention weight.

Causal dependence

The property that an activation at position i depends only on input positions $j \leq i$. In this transformer it follows from causal attention and positionwise operations, and permits reuse of earlier keys and values.

Multi-head attention (MHA)

Parallel attention heads whose outputs are concatenated and linearly projected back to the model dimension. In $\text{SelfAttn}_{\xi, M}$, the learned parameters $\xi = (\eta_1, \dots, \eta_h, W^O)$ include each head's projections and the output projection; M specifies the mask. Standard MHA gives each query head its own key and value projections; see [Equation \(1.10\)](#).

Grouped-query attention (GQA); multi-query attention (MQA)

GQA partitions query heads into groups that share key and value projections within each group. MQA uses one shared key/value head for all query heads. Both reduce the key/value data stored and read during cached decoding.

Residual connection; residual stream

A residual connection adds a sublayer's output to its input, $h \mapsto h + F(h)$. The residual stream is the sequence of hidden representations updated by these additions as they pass through the blocks.

Positionwise map; positionwise lift

The extension of $f : \mathbb{R}^d \rightarrow \mathbb{R}^r$ to sequences by applying the same function independently at every position, with shared parameters: $(f(H)_{i,:})^\top = f(h_i)$. See [Equation \(1.8\)](#).

Multilayer perceptron (MLP)

A feed-forward neural network composing learned affine maps and coordinatewise activation functions, such as ReLU or GELU. The example in [Equation \(1.24\)](#) has two affine maps with an activation function between them.

Positionwise feed-forward network (FFN)

A neural network applied independently to each position's activation vector, using shared parameters. Feed-forward means that its computation graph has no directed cycles. Its input–output function is the positionwise map FF_ψ , with learned parameters ψ . An MLP, a gated network such as SwiGLU, or a mixture of experts can implement this component; see [Section 1.4.2](#).

Mixture of experts (MoE)

A layer combining expert maps using input-dependent router weights. In the sparse version here, each position evaluates only a small selected subset of experts and takes a weighted sum of their outputs.

Absolute position embedding

A vector assigned to an absolute sequence position and added to its token embedding. The position vectors may be learned or prescribed, for example by sinusoidal functions.

Relative-position bias

A scalar depending on the displacement $i - j$, or a bucket containing it, added to the query–key score. See [Equation \(1.30\)](#).

Rotary position embeddings (RoPE)

Position-dependent rotations applied to query and key coordinate pairs before their inner product.

The paired rotations introduce dependence on relative displacement; see [Exercise 1](#). RoPE does not itself mask sources.

Decoding rule; sampling; greedy decoding

A decoding rule selects a next token from the predicted distribution. Sampling draws a random token with those probabilities. Greedy decoding selects a token with maximum probability, with a rule for breaking ties; this need not maximize the probability of the complete sequence.

Prefill

The initial forward pass over all prompt tokens, which computes their keys and values and the distribution for the first generated token.

Decode step

An incremental forward pass for a newly appended token, using earlier cached keys and values to compute the distribution for the next token.

KV cache

Stored keys and values from earlier positions, separately for each layer and key/value head. Causal dependence lets later decode steps reuse them without recomputing the earlier positions; see [Section 1.7](#).

Attention entropy

The entropy $H(\alpha) = -\sum_{j=1}^T \alpha_j \log \alpha_j$ of one query’s attention weights, with $0 \log 0 = 0$. It ranges from 0 for all mass on one source to $\log T$ for uniform weights on T sources.

Lost in the middle

An observed pattern of worse performance when relevant information is in the middle of a long input rather than near its beginning or end.

Context rot

An empirical term for deteriorating task performance as input length grows, even within the supported context length.

A How tokenizers are constructed

Two common corpus-based constructions illustrate the main ideas.

1. *Byte-pair encoding (BPE)*. Merging a frequent adjacent pair shortens the corpus’s tokenization wherever that pair is replaced. Start with base symbols, count adjacent token pairs in the training corpus, merge a most frequent pair into a new vocabulary item, and repeat until the desired vocabulary size is reached. The learned merge order determines the segmentation of new text.
2. *Unigram tokenization*. To compare alternative segmentations, assign each candidate token string v a probability $q(v)$. Start with a large collection of candidates and score a segmentation $x_1, \dots, x_T$ by multiplying its token probabilities, or equivalently adding their logarithms:

$$\prod_{t=1}^T q(v_{x_t}), \quad \text{or equivalently} \quad \sum_{t=1}^T \log q(v_{x_t}). \quad (1.55)$$

Fit the probabilities to the corpus, remove candidates whose deletion hurts the likelihood least, and repeat. At encoding time one may choose a highest-scoring segmentation or sample among segmentations.

Bibliographical notes

The compiler analogy is literal at the level of the interface: a lexer maps a character stream into tokens. A programming-language specification determines compiler tokens; corpus frequencies help determine LLM tokens. See [Aho et al. \[2006\]](#). The supplementary appendix describes the BPE construction of [Sennrich et al. \[2016\]](#), the unigram formulation of [Kudo \[2018\]](#), and the SentencePiece system of [Kudo and Richardson \[2018\]](#).

The transformer architecture was introduced by [Vaswani et al. \[2017\]](#). Their model combined encoder and decoder stacks; the equations in this lecture are a causal decoder specialization and use a pre-normalized block. Their Section 3.2.1 motivates the $1/\sqrt{d_h}$ attention-score scaling by the variance of a query–key dot product. [Remark 1.4.1](#) supplies an explicit random-initialization model for that calculation.

The SwiGLU FFN architecture was introduced by [Shazeer \[2020\]](#). It uses two hidden projections and coordinatewise gating; the paper also explains how to adjust the hidden width when comparing its parameter and operation counts with a two-layer MLP. SwiGLU is used, for example, in the Llama 3 architecture [[Grattafiori et al., 2024](#)].

Three early pretraining recipes made the architectural alternatives concrete. The original GPT work paired a causal decoder language model with task-specific supervised fine-tuning [[Radford et al., 2018](#)]. BERT instead pretrained a bidirectional encoder with a masked-token objective and then fine-tuned its representations [[Devlin et al., 2019](#)]. T5 retained the encoder–decoder architecture and expressed many tasks through one text-to-text interface [[Raffel et al., 2020](#)]. GPT-3 later demonstrated that a sufficiently large causal language model could use task descriptions and examples supplied in its context, without a gradient update at test time [[Brown et al., 2020](#)]. These are historically important successful recipes.

The modular definitions in [Section 1.13](#) follow the encoder–decoder construction of [Vaswani et al. \[2017\]](#) and the architectural comparisons of [Raffel et al. \[2020\]](#). The latter explain why parameter count and computation cannot both be matched by changing depth alone and report the strongest performance from an encoder–decoder model at approximately matched computation in their setting.

Recent publicly released model families usually preserve the causal decoder-only interface while changing the surrounding recipe and internal details. Llama 3 is a dense example [[Grattafiori et al., 2024](#)], Mistral 7B uses grouped-query and sliding-window attention [[Jiang et al., 2023](#)], and DeepSeek-V3 uses a mixture-of-experts architecture [[DeepSeek-AI, 2024](#)]. The term *open-weight* asserts release of the numerical weights; training data, code, and licensing freedoms require separate verification. For a maintained visual catalogue of model-level architecture diagrams, configuration links, and compact fact sheets, see the LLM Architecture Gallery of [Raschka \[2026\]](#). Figures 5 and 6 are independent LaTeX renderings of the architectures and dimensions catalogued there; the Llama 3 specifications were also checked against [Grattafiori et al. \[2024\]](#).

The original paper used additive sinusoidal position vectors. Rotary position embeddings are due to [Su et al. \[2021\]](#); their position-dependent rotations of queries and keys expose relative displacement in the attention score. [Equation \(1.31\)](#) gives their fixed geometric frequency schedule. The bucketed learned relative-position bias in (1.30) follows [Raffel et al. \[2020\]](#). [Press et al. \[2022\]](#) propose ALiBi, which instead adds a fixed head-dependent linear penalty based on relative distance.

[Shazeer \[2019\]](#) identified KV-cache bandwidth as a decoding bottleneck and introduced multi-query attention: multiple query heads share the same key and value heads. This reduces cached key–value data and the traffic required to read it. Grouped-query attention, introduced by [Ainslie et al. \[2023\]](#), provides an intermediate degree of sharing.

[Dao et al. \[2022\]](#) introduced FlashAttention, which computes exact dense attention by tiling

and an online softmax. This reduces memory traffic and avoids materializing the full attention matrix while retaining the quadratic arithmetic of exact dense attention. The course returns to this distinction in Lecture 27.

The bounded-score calculation in [Exercise 3](#) is elementary. [Chiang and Cholak \[2022\]](#) study a related length-dependent loss of confidence and show that the obstruction depends on architectural details; one of their remedies scales attention logits with $\log T$. This is useful context for the exercise result.

On the empirical capabilities of trained models across context-window lengths and locations of relevant information, [Liu et al. \[2024\]](#) vary the position of relevant information in multi-document question answering and key–value retrieval, demonstrating the distinction between accepting a long context and using it reliably. [Hong et al. \[2025\]](#) use the term “context rot” for performance degradation as input length grows across a broader collection of tasks and models.

One way to study the division of work described in [Section 1.4.7](#) is to write sequence computations as programs with explicit selection, aggregation, and positionwise operations. [Weiss et al. \[2021\]](#) introduced the *Restricted Access Sequence Processing* (RASP) language for this purpose, relating its programs to transformer heads and layers while abstracting away their implementation with weights and activations. [Zhou et al. \[2024\]](#) introduced RASP-L for causal decoder-only transformers and proposed short-program length as an empirical predictor of algorithmic learnability and length generalization. The [codebases below](#) support experiments with these languages.

[Arthur \[1989\]](#) studies competing technologies whose benefits increase with adoption. His model shows how early advantages can lead to technological lock-in. It provides a framework for the closing discussion of coevolution; that discussion’s application to AI research is a qualitative analogy.

For an analysis of how trained models use RoPE, see [Barbero et al. \[2025\]](#). Their study of Gemma 7B identifies heads with positional attention patterns, including attention to the preceding token, and examines how slowly rotating coordinates support content-based comparisons. They also discuss limitations of these behaviors when extending context length. This connects the rotation calculation in [Exercise 1](#) to mechanisms observed in a trained model. For the exercise’s uniform-addressing task, independently uniform frequencies give a separation guarantee. For language modeling, a geometric schedule seems a more promising starting point because it deliberately supplies slow rotations for stable content comparisons. This is a task-dependent judgment, not a conclusion from a head-to-head empirical comparison of the schedules.

Hoeffding’s inequality ([Theorem 1.14.1](#)) bounds deviations of sums of independent bounded random variables [[Hoeffding, 1963](#)]. Applying it to independent random rotation frequencies gives a probabilistic construction of relative addresses with uniformly separated scores over a prescribed finite context.

Codebases for experiments

The following codebases support experiments with the constructions in these notes.

NumPy RASP-L

A small reference implementation for writing and testing causal RASP-L sequence programs: <https://github.com/apple/ml-np-rasp>.

RASP and Tracr

The original RASP interpreter displays selectors and intermediate sequence operations. Tracr compiles a subset of RASP into concrete transformer weights: <https://github.com/tech-srl/RASP> and <https://github.com/google-deepmind/tracr>.

nanochat

A compact end-to-end codebase containing tokenization, pretraining, fine-tuning, evaluation, and inference. It includes small CPU and Apple-Silicon configurations, although training a capable language model still requires substantial computation: <https://github.com/karpathy/nanochat>.

Hugging Face Transformers

A broad model-definition and checkpoint ecosystem, useful for loading models and comparing tokenizers, architectures, and generation rules: <https://github.com/huggingface/transformers>.

TransformerLens

An inspection library for GPT-style models, useful for recording activations and attention patterns or intervening inside a forward pass: <https://github.com/TransformerLensOrg/TransformerLens>.

llama.cpp

A local inference implementation that makes tokenization, quantization, decoding, and key-value-cache behavior accessible to experiments: <https://github.com/ggml-org/llama.cpp>.

References

- W. Brian Arthur. Competing Technologies, Increasing Returns, and Lock-In by Historical Events. *The Economic Journal*, 99(394):116–131, 1989. <https://doi.org/10.2307/2234208>.
- A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, second edition, 2006.
- R. Sennrich, B. Haddow, and A. Birch. Neural Machine Translation of Rare Words with Subword Units. *ACL*, 2016. <https://arxiv.org/abs/1508.07909>.
- T. Kudo. Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates. *ACL*, 2018. <https://arxiv.org/abs/1804.10959>.
- T. Kudo and J. Richardson. SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing. *EMNLP System Demonstrations*, 2018. <https://aclanthology.org/D18-2012/>.
- A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need. *NeurIPS*, 2017. <https://arxiv.org/abs/1706.03762>.
- A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving Language Understanding by Generative Pre-Training. OpenAI technical report, 2018. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.
- J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *NAACL*, 2019. <https://arxiv.org/abs/1810.04805>.
- T. B. Brown et al. Language Models are Few-Shot Learners. *NeurIPS*, 2020. <https://arxiv.org/abs/2005.14165>.
- A. Grattafiori et al. The Llama 3 Herd of Models. 2024. <https://arxiv.org/abs/2407.21783>.

- S. Raschka. LLM Architecture Gallery. Maintained online reference, accessed September 3, 2026. <https://sebastianraschka.com/llm-architecture-gallery/>.
- A. Q. Jiang et al. Mistral 7B. 2023. <https://arxiv.org/abs/2310.06825>.
- DeepSeek-AI. DeepSeek-V3 Technical Report. 2024. <https://arxiv.org/abs/2412.19437>.
- W. Hoeffding. Probability Inequalities for Sums of Bounded Random Variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963. <https://doi.org/10.1080/01621459.1963.10500830>.
- F. Barbero, A. Vitvitskyi, C. Perivolaropoulos, R. Pascanu, and P. Veličković. Round and Round We Go! What Makes Rotary Positional Encodings Useful? *ICLR*, 2025. <https://arxiv.org/abs/2410.06205>.
- J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu. RoFormer: Enhanced Transformer with Rotary Position Embedding. 2021. <https://arxiv.org/abs/2104.09864>.
- C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. <https://www.jmlr.org/papers/v21/20-074.html>.
- O. Press, N. A. Smith, and M. Lewis. Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. *ICLR*, 2022. <https://arxiv.org/abs/2108.12409>.
- N. Shazeer. Fast Transformer Decoding: One Write-Head is All You Need. 2019. <https://arxiv.org/abs/1911.02150>.
- N. Shazeer. GLU Variants Improve Transformer. 2020. <https://arxiv.org/abs/2002.05202>.
- J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. *EMNLP*, 2023. <https://arxiv.org/abs/2305.13245>.
- T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *NeurIPS*, 2022. <https://arxiv.org/abs/2205.14135>.
- D. Chiang and P. Cholak. Overcoming a Theoretical Limitation of Self-Attention. *ACL*, 2022. <https://aclanthology.org/2022.acl-long.527/>.
- N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 2024. <https://arxiv.org/abs/2307.03172>.
- K. Hong, A. Troynikov, and J. Huber. Context Rot: How Increasing Input Tokens Impacts LLM Performance. Chroma technical report, 2025. <https://www.trychroma.com/research/context-rot>.
- G. Weiss, Y. Goldberg, and E. Yahav. Thinking Like Transformers. *ICML*, 2021. <https://proceedings.mlr.press/v139/weiss21a.html>.
- H. Zhou, A. Bradley, E. Littwin, N. Razin, O. Saremi, J. Susskind, S. Bengio, and P. Nakkiran. What Algorithms Can Transformers Learn? A Study in Length Generalization. *ICLR*, 2024. <https://arxiv.org/abs/2310.16028>.