

Lecture 1: Introduction: language modeling; the transformer

Instructors: *Alex Andoni, Edo Liberty*Scribes: *James Wang*

1 Language modeling

Before embarking on our journey to study Algorithms in LLMs, we must formally define the concept of a *language model*. When you think of a language model today, the first thing that pops into your head might be a large language model like GPT. However, as we will see later, there are many other approaches to language modeling and in studying the algorithmic principles behind language modeling we must work with a more general notion of a language model.

Henceforth, fix an alphabet V consisting of finitely many symbols. These symbols can be characters, syllables, words, or something else.¹

Definition 1 (Language model). *A language model $q : V^* \rightarrow [0, 1]$ is a probability distribution over (finite) strings which end in a special symbol, which we denote as EOS or $\perp$.*

To generate from a language model, we simply return a string $x \sim q$. This is usually implemented by a procedure known as *autoregressive sampling*.

Definition 2 (Autoregressive sampling). *For any string $x = x_{1:n}$, we have by the chain rule of probability*

$$q(x) = \prod_{i=1}^n q(x_i \mid x_{<i}).$$

Thus, we may sample from q as follows: starting with the empty prefix, repeatedly sample $x_i \sim q(\cdot \mid x_{<i})$ and append x_i to the prefix. Once $x_i = \text{EOS}$, return the resulting string $x_{1:i}$.

So far, we have introduced language models as distributions over strings. However, this definition leaves out an important implementation detail, namely: *how can we sample from q algorithmically?* Thus, going forward, we will overload our definition of a language model to also denote an algorithm which may be used to sample $x \sim q$ autoregressively.

Definition 3 (Language model). *A language model is an algorithm which takes as input a prefix $x_{<i}$ and returns the conditional distribution $q(x_i \mid x_{<i})$ of the next token.*

1.1 Building a language model

A central question we will consider is: *how can we build a language model?* Modern large language models involve multiple stages, including (but not limited to) pretraining, midtraining, supervised finetuning (SFT), and reinforcement learning.² For now, we will focus on the *pretraining stage*, which reduces the problem to supervised learning (i.e., next-token prediction) on a large corpus of text.

¹Deciding how to choose V is an interesting question in itself, which we will discuss in the near future.

²For a description of a modern training pipeline, see [DeepSeek-AI et al. \(2024\)](#).

Definition 4 (Pretraining). Given a number of sequences $x^{(1)}, \dots, x^{(m)}$, let $p = \text{Uniform}\{x^{(1)}, \dots, x^{(m)}\}$ denote their empirical distribution. Let $Q \subseteq \Delta(V^*)$ denote a class of models. In the pretraining stage, we minimize the next-token prediction loss:

$$L = \min_{q \in Q} \mathbb{E}_{x \sim p}[-\log q(x)]. \quad (1)$$

We refer to the quantity $2^{\mathbb{E}_{x \sim p}[-\log q(x)]/\mathbb{E}_{x \sim p}[|x|]}$ as the perplexity.

The objective (1) is known as the *negative log-likelihood* or the *cross-entropy*, and the corresponding minimization problem is equivalent to *maximum likelihood estimation*. A minimizer $q^* \in Q$ for (1) is a model for which the likelihood of observing $x^{(1)}, \dots, x^{(m)}$ under q^* is higher over all models $q \in Q$.

Another useful interpretation of (1) is distribution matching. To see this, we can rewrite (1) as

$$\begin{aligned} \mathbb{E}_{x \sim p}[-\log q(x)] &= \mathbb{E}_{x \sim p}[\log 1/q(x)] \\ &= \underbrace{\mathbb{E}_{x \sim p} \left[\log \frac{1}{p(x)} \right]}_{=H(p)} + \underbrace{\mathbb{E}_{x \sim p} \left[\log \frac{p(x)}{q(x)} \right]}_{=D_{\text{KL}}(p \| q)}. \end{aligned}$$

The first term, $H(p)$, is known as the *entropy* and does not depend on q . The second term, $D_{\text{KL}}(p \| q)$, is known as the *KL-divergence* and is one way of measuring the similarity between p and q . Hence any minimizer q^* for (1) is also a minimizer for $D_{\text{KL}}(p \| q)$ and thus (1) can be viewed as distribution matching in terms of the KL-divergence.

As a quick aside, the KL-divergence is an extremely important quantity. Before proceeding, we will list and prove a few of its most fundamental properties.

Fact 5 (Nonnegativity and positivity). Let $\mathcal{X}$ be a sample space and $p, q \in \Delta(\mathcal{X})$. Then, $D_{\text{KL}}(p \| q) \geq 0$ with equality iff $p = q$.

Proof. By Jensen's inequality,

$$D_{\text{KL}}(p \| q) = -\mathbb{E}_{x \sim p} \left[\log \frac{q(x)}{p(x)} \right] \geq -\log \left(\sum_{x:p(x)>0} q(x) \right) \geq 0.$$

By strict concavity of log, equality throughout holds iff $p = q$. $\square$

Fact 6. The KL-divergence is not symmetric, i.e., we generally have $D_{\text{KL}}(p \| q) \neq D_{\text{KL}}(q \| p)$.

Now, let us return to objective (1). We present another motivation for studying the logarithmic loss $\mathbb{E}_{x \sim p}[-\log q(x)]$ that comes from the perspective of prediction. The following result due to [Bernardo \(1979\)](#) states, informally, that if *any* prediction loss $L(q) = \mathbb{E}_{x \sim p}[S(q, x)]$ (where S is some scoring function; $S(q, x) = -\log q(x)$ for the log loss) is uniquely minimized by taking $q = p$ and it only depends on x through $q(x)$, then minimizing L is equivalent to minimizing the log loss. In this sense, the log loss is a canonical choice of loss function. The formal statement imposes some minor technical conditions.

Theorem 7 (Bernardo's characterization of logarithmic loss). Let $\mathcal{X}$ be finite with $|\mathcal{X}| \geq 3$, and suppose $S(q, x) = s_x(q(x))$, where each s_x is differentiable. If, for every strictly positive distribution p , $\mathbb{E}_{x \sim p}[S(q, x)]$ is uniquely minimized at $q = p$, then

$$S(q, x) = -a \log q(x) + b(x)$$

for some $a > 0$ and function $b : \mathcal{X} \rightarrow \mathbb{R}$.

Proof. For distinct x, y , perturb $q = p$ by increasing $q(x)$ by ε and decreasing $q(y)$ by ε . This preserves $\sum_z q(z) = 1$ and is feasible for sufficiently small $|\varepsilon|$. Since $\varepsilon = 0$ minimizes the expected loss, differentiating gives

$$0 = \frac{d}{d\varepsilon} [p(x)s_x(p(x) + \varepsilon) + p(y)s_y(p(y) - \varepsilon)] \Big|_{\varepsilon=0} = p(x)s'_x(p(x)) - p(y)s'_y(p(y)).$$

Since $|\mathcal{X}| \geq 3$, we can vary $p(x)$ while fixing $p(y)$. Hence $ts'_x(t)$ is a common constant c , so $s_x(t) = c \log t + b(x)$. Therefore,

$$\mathbb{E}_{x \sim p}[S(q, x) - S(p, x)] = -c D_{\text{KL}}(p \parallel q).$$

Unique minimization of the KL-divergence at $q = p$ forces $c < 0$. Taking $a = -c$ proves the claim. $\square$

1.2 Language modeling is compression

In this section we present yet another motivation for the loss function (1), this time from the perspective of compression and coding.

Definition 8 (Code). A code $C : V^* \rightarrow \{0, 1\}^*$ maps strings over V to bit strings. We say that C is decodable if it is injective, and prefix-free if, for all distinct $x, y \in V^*$, $C(x)$ is not a prefix of $C(y)$.³

Given a distribution $p \in \Delta(V^*)$, we consider the goal of minimizing expected code length,

$$\min_C \mathbb{E}_{x \sim p}[l_x], \quad \text{where } l_x = |C(x)|.$$

A celebrated result due to Claude Shannon upper bounds the optimal expected prefix-free code length in terms of the entropy $H(p)$.

Theorem 9 (Shannon). Let $p \in \Delta(V^*)$ have finite entropy $H(p)$. Every prefix-free code C satisfies

$$\mathbb{E}_{x \sim p}[|C(x)|] \geq H(p),$$

and there exists a prefix-free code C^* satisfying

$$\mathbb{E}_{x \sim p}[|C^*(x)|] \leq H(p) + 1.$$

Proof. View bit strings as nodes of a binary tree, with a string of length l at depth l . A code is prefix-free precisely when no codeword is an ancestor of another.

We first prove the lower bound. Fix a depth m . Each codeword $C(x)$ of length at most m has $2^{m-|C(x)|}$ descendants at depth m . These sets of descendants are disjoint because the code is prefix-free. Since there are 2^m nodes at depth m ,

$$\sum_{x: |C(x)| \leq m} 2^{m-|C(x)|} \leq 2^m.$$

Dividing by 2^m and letting $m \rightarrow \infty$ gives $\sum_x 2^{-|C(x)|} \leq 1$. Jensen's inequality now yields

$$H(p) - \mathbb{E}_{x \sim p}[|C(x)|] = \mathbb{E}_{x \sim p} \left[\lg \frac{2^{-|C(x)|}}{p(x)} \right] \leq \lg \left(\sum_{x: p(x) > 0} 2^{-|C(x)|} \right) \leq 0.$$

³Note that every prefix-free code is also decodable.

For the upper bound, assign each positive-probability string the desired length $l_x = \lceil -\lg p(x) \rceil$. These satisfy

$$\sum_{x:p(x)>0} 2^{-l_x} \leq \sum_x p(x) = 1.$$

We construct codewords in order of nondecreasing desired length. To assign a codeword of length l_x , choose a node at depth l_x that does not descend from any previously chosen codeword. Such a node always exists: previously chosen codewords exclude

$$\sum_{\text{previous } y} 2^{l_x - l_y} \leq 2^{l_x} - 1$$

of the 2^{l_x} available nodes, where the inequality follows by including 2^{-l_x} in the preceding bound on the sum. Since the lengths are nondecreasing, the new codeword also cannot be an ancestor of an earlier one. The resulting code is therefore prefix-free, and

$$\mathbb{E}_{x \sim p}[|C^*(x)|] = \mathbb{E}_{x \sim p}[\lceil -\lg p(x) \rceil] \leq H(p) + 1.$$

Zero-probability strings can also be included without violating the bound.⁴ □

In practice, we do not know p , so we construct a code using a language model q . Assuming $q(x) > 0$ for every string being encoded, the preceding construction gives a prefix-free code C_q with lengths $\lceil -\lg q(x) \rceil$. Its expected length under p satisfies

$$H(p) + D_{\text{KL}}(p \parallel q) \leq \mathbb{E}_{x \sim p}[|C_q(x)|] \leq H(p) + D_{\text{KL}}(p \parallel q) + 1,$$

where the KL divergence is measured in bits. This follows from

$$\mathbb{E}_{x \sim p}[-\lg q(x)] = H(p) + D_{\text{KL}}(p \parallel q).$$

Thus, minimizing the negative log loss minimizes the expected ideal code length for data drawn from p . The KL divergence measures the excess expected number of bits from using q instead of p .

Conversely, a prefix-free code C induces a language model

$$q_C(x) = \frac{2^{-|C(x)|}}{\sum_y 2^{-|C(y)|}}.$$

Strings with shorter encodings receive higher probability. Since $\sum_y 2^{-|C(y)|} \leq 1$,

$$-\lg q_C(x) = |C(x)| + \lg \left(\sum_y 2^{-|C(y)|} \right) \leq |C(x)|.$$

Hence, a code with small expected length induces a language model with small expected log loss.

This connection also yields practical compression algorithms. Arithmetic coding uses a language model's next-token probabilities to encode a string x using close to

$$-\lg q(x) = -\sum_{i=1}^{|x|} \lg q(x_i \mid x_{<i})$$

⁴If the sum of 2^{-l_x} is less than 1, assign these strings sufficiently large lengths and apply the same construction. If the sum equals 1, then $l_x = -\lg p(x)$ for every positive-probability string. Increasing one such length by 1 leaves room for the remaining strings and increases the expected length from $H(p)$ by at most 1.

bits. Strong language models therefore yield strong text compressors: [Delétang et al. \(2024\)](#) show that pretrained language models combined with arithmetic coding can outperform standard compressors on text. These comparisons assume that the encoder and decoder already share the model; otherwise, its description must also be included in the compressed size.

1.3 k -Gram models

Having extensively justified the choice of the log loss, we now turn our attention to choosing the model class $\mathcal{Q}$. Before introducing more modern architectures like the transformer, we first introduce one of the earliest class of models, the k -gram model.

Definition 10 (k -Gram model). *A k -gram model predicts the next token using only the preceding $k - 1$ tokens:*

$$q(x_i \mid x_{<i}) = q(x_i \mid x_{\max\{1, i-k+1\}}, \dots, x_{i-1}).$$

The k -gram model is a useful but restrictive model of human language. First of all, the k -gram model does not model any long context dependencies that are present in typical human language, unless k is very large. However, this presents a different challenge: to store a k -gram model, we generally must store roughly $|V|^k$ numbers encoding the conditional distributions for each possible $(k - 1)$ -tuple of context.

Inductive biases. Shortly, we will introduce the transformer architecture. Compared to a hypothetical “matched-performance” k -gram model (by this, we mean a k -gram model with the same performance as a transformer on human language modeling tasks), a transformer will take much less space to describe.

This is an example of a mismatch between the *inductive bias* of the k -gram architecture and the structure of human language. A k -gram model assigns any $(k - 1)$ -tuple of context the same amount of storage, regardless of whether that context is a meaningful phrase like $x^{(1)} = (\text{My, favorite, class, at, Columbia, is})$ or a completely nonsensical phrase like $x^{(2)} = (\text{dog, yes, up, fun, algorithm, 3})$. In a human language corpus, the probability of ever seeing $x^{(2)}$ is (essentially) zero; thus, we can reduce the memory footprint of our k -gram model by simply not storing the conditional distribution corresponding to context $x^{(2)}$, at no cost to our model’s ability to represent typical human language.

In fact, human language is highly compressible: language concentrates probability on a small fraction of possible contexts, and distinct contexts may still have related continuations (for example, $x^{(1)}$ and $x^{(3)} = (\text{My, favorite, class, at, UPenn, is})$). This motivates architectures which make use of the natural compressibility of human language, such as the transformer.

2 Transformers

In this section, we will introduce the transformer architecture, first introduced in [Vaswani et al. \(2017\)](#). In doing so we will first abstract out the fundamental components: token embeddings, attention, and nonlinearities. Then, we will describe typical implementations of these components.

A transformer computes next-token probabilities for a context $x_{1:n}$ by maintaining a vector representation at each position. It repeatedly updates these representations using two operations: *attention*, which combines information across positions, and a *multilayer perceptron* (MLP), which transforms each position separately. We describe a decoder-only transformer, as used in autoregressive language models.

Throughout, let d denote the dimension of the vector representation. We refer to d as the *dimension* of the model. Let L denote the number of transformer blocks. For $l \in [L]$, we write $h_i^{(l)} \in \mathbb{R}^d$ for the

representation at position i entering block l , and $\tilde{h}_i^{(l)}$ for its representation after attention. The MLP then produces $h_i^{(l+1)}$. Thus, $h_i^{(1)}$ is the initial embedding, while $h_i^{(L+1)}$ is the final representation.

Token and positional embeddings. A *token embedding* $E : V \rightarrow \mathbb{R}^d$ maps each token $z \in V$ to a vector. A *positional embedding* $P : \mathbb{N} \rightarrow \mathbb{R}^d$ maps each position $i \in \mathbb{N}$ to a vector. The *embedding* of token x_i in an input string $x_{1:n}$ is typically taken to be the sum of its positional and token embeddings:

$$h_i^{(1)} = E(x_i) + P(i), \quad i \in [n].$$

Self-attention and masking. The most iconic component of the transformer architecture is self-attention, which provides a mechanism to mix information across different token positions. For the current layer $l \in [L]$, let $h_{1:n}^{(l)}$ denote the current vector representations for each input position.

A single-head attention layer is parameterized by a set of matrices $W_Q^{(l)}, W_K^{(l)} \in \mathbb{R}^{r \times d}$, $W_V^{(l)} \in \mathbb{R}^{s \times d}$, and $W_O^{(l)} \in \mathbb{R}^{d \times s}$. For each position i , the head computes a *query*, *key*, and *value*:

$$q_i = W_Q^{(l)} h_i^{(l)}, \quad k_i = W_K^{(l)} h_i^{(l)}, \quad v_i = W_V^{(l)} h_i^{(l)}.$$

The query at position i is compared with each key to determine the attention weights:

$$\alpha_{ij} = \frac{\exp(\langle q_i, k_j \rangle / \sqrt{r})}{\sum_{t=1}^n \exp(\langle q_i, k_t \rangle / \sqrt{r})}.$$

The head then outputs the attention representation

$$a_i^{(l)} = W_O^{(l)} \sum_{j=1}^n \alpha_{ij} v_j. \tag{2}$$

Thus, the attention weights determine which positions contribute information, while the value and output matrices determine how that information is transformed.

Similarly, a k -head attention layer is parameterized by k single-head attention layers; following (2), let us write $a_i^{(l)(j)} \in \mathbb{R}^d$ to denote the post-attention vector representation of the i th input token at layer l and head j . Then, the output of the corresponding k -head attention layer is simply given by

$$\tilde{h}_i^{(l)} = \sum_{j=1}^k a_i^{(l)(j)}.$$

For autoregressive language modeling, position i is used to predict token x_{i+1} and must therefore not access later tokens. We enforce this using *causal masking*: in every head and layer, we replace the attention weights by

$$\alpha_{ij} = \begin{cases} \frac{\exp(\langle q_i, k_j \rangle / \sqrt{r})}{\sum_{t=1}^i \exp(\langle q_i, k_t \rangle / \sqrt{r})}, & j \leq i, \\ 0, & j > i. \end{cases}$$

Equivalently, we set the scores of future positions to $-\infty$ before applying softmax. This ensures that the representation at position i depends only on $x_{1:i}$ in every layer.

Multilayer perceptrons. In typical implementations, attention is followed by a *multilayer perceptron* (MLP), applied independently at each position:

$$h_i^{(l+1)} = W_2^{(l)} \sigma(W_1^{(l)} \tilde{h}_i^{(l)} + b_1^{(l)}) + b_2^{(l)},$$

where $W_1^{(l)} \in \mathbb{R}^{m \times d}$, $W_2^{(l)} \in \mathbb{R}^{d \times m}$, and σ is a coordinatewise nonlinear function, such as ReLU. The same MLP is applied to each input position’s representation independently. The resulting representations $h_{1:n}^{(l+1)}$ are passed to the next attention layer. In practice, these layers are often implemented with residual connections and normalization layers, but we omit these for simplicity.

Output probabilities. A final affine map produces vocabulary logits, which define the next-token probabilities via a softmax transform:

$$o_i = W_U h_i^{(L+1)} + b_U, \quad q_\theta(x_{i+1} = v \mid x_{\leq i}) = \frac{\exp(o_{i,v})}{\sum_{w \in V} \exp(o_{i,w})}.$$

2.1 Autoregressive generation with a transformer

Given a prefix $x_{1:t}$, a *forward pass* first computes its embeddings, then applies the L transformer blocks. The output map at position t converts $h_t^{(L+1)}$ into the distribution $q(\cdot \mid x_{1:t})$. We sample x_{t+1} from this distribution, append it to the prefix, and repeat until sampling EOS. When generating without a prompt, a special beginning-of-sequence token provides the initial context.

Figure 1 illustrates the forward pass through two transformer blocks. Each column corresponds to a token position, and information flows upward. During generation, the output at the final position predicts the next token.

KV caching. A straightforward implementation recomputes the entire prefix at every step. However, causal masking implies that appending a token cannot change the representations of earlier positions. Their keys and values therefore remain unchanged in every layer.

We can avoid this repeated computation by storing the keys and values in a *KV cache*. At each layer, we compute the query, key, and value only for the new position. Its query is compared with the cached keys and its own key, and the corresponding values are combined to produce its attention representation. We then apply the MLP at the new position and pass its representation to the next layer. The new key and value at each layer are added to the cache.

Time complexity of autoregressive generation. For simplicity, suppose the head dimensions satisfy $kr = ks = d$, and the MLP width is $m = O(d)$. Without caching, recomputing attention on a prefix of length t costs $O(Lt^2d)$. Thus, generating a sequence of length n requires attention computation totaling

$$\sum_{t=1}^n O(Lt^2d) = O(Ln^3d). \quad (3)$$

With KV caching, only the new input position requires fresh computation to generate its vector representations, and hence the total attention cost is reduced to

$$\sum_{t=1}^n O(Ltd) = O(Ln^2d). \quad (4)$$

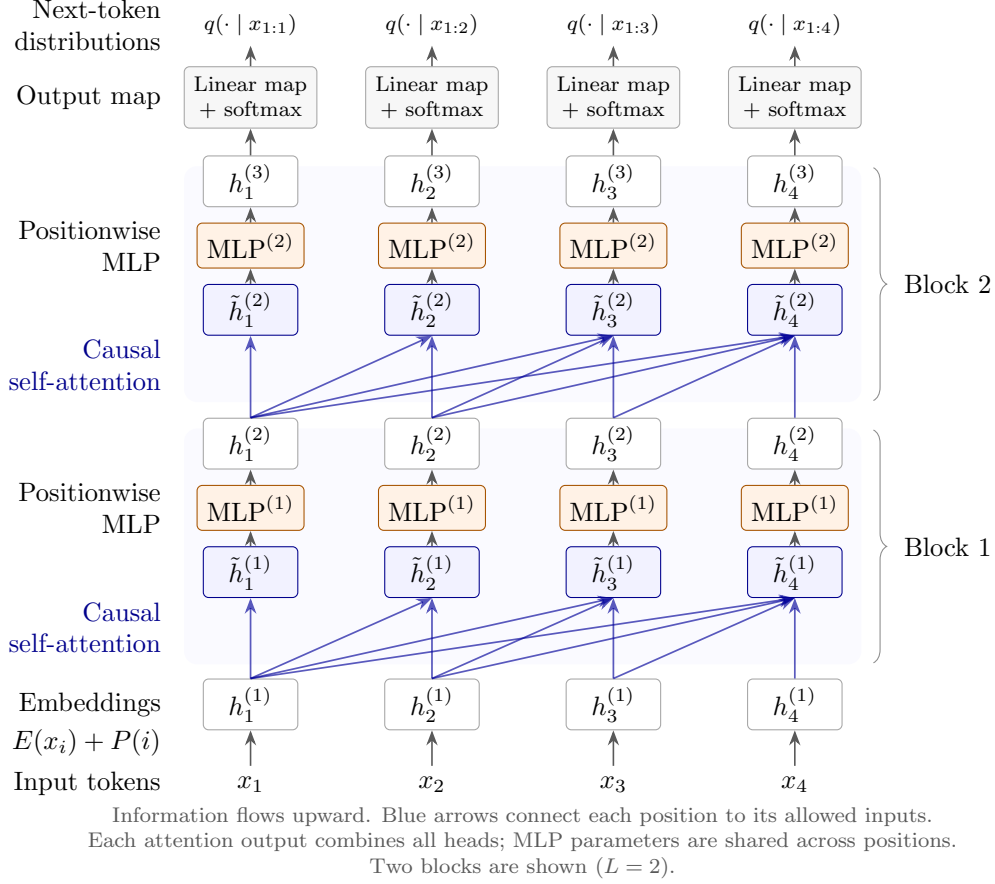


Figure 1: A transformer with two blocks. Attention mixes information across positions subject to causal masking, while each MLP acts independently at each position.

The cost of the self-attention mechanism is the dominant computational cost of autoregressive generation with a transformer for long contexts, so comparing (3) and (4) demonstrates huge a benefit of causal masking.

We stress that *causal masking is what makes these cached representations reusable*. Without masking, adding a token can change earlier representations, invalidating their cached keys and values in subsequent layers. The speedup therefore comes from combining causal masking with KV caching.

3 State space models

Transformers are not the only architecture used for modern language modeling. A *state space model* (SSM) processes a sequence by maintaining a state that is updated as each token arrives. In contrast to transformers, where self-attention allows the transformer to "attend" directly to earlier positions in the context, a SSM must compress its entire context into a fixed state $s \in \mathbb{R}^m$. This leads to a tradeoff: SSMs are generally worse than transformers at long-context tasks, but since they only need to maintain a fixed-dimension state, they do not require more computation as the length of the conversation grows.

Classical state space models. A basic discrete-time linear SSM is parameterized by matrices A , B , C , D and takes input vectors $h_1, \dots, h_n \in \mathbb{R}^d$ and computes

$$s_i = As_{i-1} + Bh_i, \quad y_i = Cs_i + Dh_i, \quad (5)$$

where $s_i \in \mathbb{R}^m$ is the state, $y_i \in \mathbb{R}^d$ is the output, and typically $s_0 = 0$. Here, $A \in \mathbb{R}^{m \times m}$, $B \in \mathbb{R}^{m \times d}$, $C \in \mathbb{R}^{d \times m}$, and $D \in \mathbb{R}^{d \times d}$. For language modeling, the h_i 's could represent a token embedding $h_i = E(x_i)$.

The state s_i summarizes the inputs seen so far, while y_i supplies the representation passed to subsequent computations. An SSM-based language model combines these layers with nonlinear transformations and a final output map giving next-token probabilities. Each layer maintains its own state during generation.

Unrolling the recurrence (5) gives

$$s_i = \sum_{j=1}^i A^{i-j} B h_j, \quad y_i = \sum_{j=1}^i C A^{i-j} B h_j + D h_i.$$

Thus, an earlier input h_j contributes through the matrix $C A^{i-j} B$. This matrix depends only on the distance $i - j$, so the sequence of outputs can also be computed as a causal convolution. Unlike attention, the coefficients of this convolution do not depend on the contents of the input.

Modern state space models. Recent work adapts this classical framework to deep learning. The *S4* architecture (Gu et al., 2021) uses structured state space layers designed to capture long-range dependencies while admitting efficient computation. Its parameterization gives the transition matrix additional structure, allowing the convolution to be evaluated efficiently during training. During generation, the same layer can instead be evaluated using its recurrent form.

The *Mamba* architecture (Gu and Dao, 2023) introduces *selective* state space models, whose update depends on the current input. Schematically, these take the form

$$s_i = A(h_i)s_{i-1} + B(h_i)h_i, \quad y_i = C(h_i)s_i + Dh_i,$$

where the input-dependent matrices $A(\cdot), B(\cdot), C(\cdot), D(\cdot)$ are computed using shared learned functions. This allows the model to control which information to retain or forget based on the input. Although the update is linear in the preceding state, it is no longer linear in the input sequence.

Input dependence prevents us from using a single fixed convolution kernel. However, these affine state updates can still be composed efficiently using a parallel scan algorithm. Thus, the recurrent description does not require training to process the positions one at a time. Mamba-2 (Dao and Gu, 2024) further develops connections between SSMs and forms of linear attention to obtain more efficient algorithms.

3.1 Copying and retrieval

An explicit limitation appears in Jelassi et al. (2024), which studies the task of reading a string and then reproducing it. Let $|V| \geq 2$. The model receives

$$\text{BOS}, x_1, \dots, x_n, \text{COPY},$$

where the two special tokens are outside V , and succeeds if its next n outputs equal $x_{1:n}$. Throughout, decoding is deterministic.

Theorem 11 (Recurrent memory lower bound). *Suppose a recurrent model retains at most b bits of state across all layers. For a uniformly random string $x \in V^n$,*

$$\Pr[\text{the model copies } x \text{ correctly}] \leq \frac{2^b}{|V|^n}.$$

Consequently, success probability at least $1/2$ requires $b \geq n \lg |V| - 1$.

Proof. After reading the prompt, the model is in one of at most 2^b states. Each state determines a single continuation, so two distinct inputs that lead to the same state cannot both be copied correctly. Thus, the model succeeds on at most 2^b of the $|V|^n$ equally likely inputs. The final claim follows by rearranging $2^b/|V|^n \geq 1/2$. $\square$

This bound applies to any recurrent update rule, including selective SSMS. It concerns stored bits, so applying it to real-valued states requires accounting for their precision.

Theorem 12 (Transformer upper bound). *Let $\varepsilon \in (0, 1/2)$ and $n \geq C \lg(1/\varepsilon)$ for a sufficiently large universal constant C . There exists a transformer with two blocks and dimension*

$$d = O(\lg(n/\varepsilon) \lg |V|)$$

that copies a uniformly random $x \in V^n$ with probability at least $1 - \varepsilon$ under greedy decoding.

Proof sketch. Choose

$$k = \left\lceil \frac{\lg(n^2/\varepsilon)}{\lg |V|} \right\rceil.$$

The construction copies correctly whenever the input has no repeated length- k substring.

The first block uses local attention to encode length- k windows, using $O(k \lg |V|)$ coordinates. The second block matches the most recently generated window to its occurrence in the input and retrieves the following token. Special handling of the boundary tokens initializes the first k outputs. The attention scores are scaled so that greedy decoding selects the retrieved token.

For any two distinct starting positions, the probability that their length- k substrings agree is $|V|^{-k}$, including when they overlap. Indeed, for an overlap at distance $t < k$, equality forces the $k + t$ symbols to be periodic with period t , leaving only t symbols free. A union bound therefore gives

$$\Pr[\text{some length-}k \text{ substring repeats}] \leq n^2 |V|^{-k} \leq \varepsilon.$$

The dimension bound follows from the choice of k . $\square$

For a fixed alphabet, the transformer dimension is therefore $O(\lg(n/\varepsilon))$. Its total memory still grows with the input length, since it retains representations at earlier positions. The comparison illustrates the limitation of compressing the entire context into a fixed-size state. The transformer result establishes that suitable parameters exist; it does not establish that training finds them.

References

José M. Bernardo. Expected information as expected utility. *The Annals of Statistics*, 7(3):686–690, 1979.

- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. [Attention is all you need](#). *Advances in Neural Information Processing Systems*, 30, 2017.
- Grégoire Delétang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, et al. [Language modeling is compression](#). In *International Conference on Learning Representations*, 2024.
- Samy Jelassi, David Brandfonbrener, Sham M. Kakade, and Eran Malach. [Repeat after me: Transformers are better than state space models at copying](#). *arXiv preprint arXiv:2402.01032*, 2024.
- Albert Gu, Karan Goel, and Christopher Ré. [Efficiently modeling long sequences with structured state spaces](#). *arXiv preprint arXiv:2111.00396*, 2021.
- Albert Gu and Tri Dao. [Mamba: Linear-time sequence modeling with selective state spaces](#). *arXiv preprint arXiv:2312.00752*, 2023.
- Tri Dao and Albert Gu. [Transformers are SSMS: Generalized models and efficient algorithms through structured state space duality](#). *arXiv preprint arXiv:2405.21060*, 2024.
- DeepSeek-AI et al. [DeepSeek-V3 technical report](#). *arXiv preprint arXiv:2412.19437*, 2024.