CIS 6200: Emerging Theory of LLMs **Fall 2026**

Topic 0: Overview of Large Language Models

Last updated: August 25, 2026 *Surbhi Goel*

Acknowledgements: *These notes draw on Jurafsky and Martin's* Speech and Language Processing *[JM26], the Stanford CS324 course notes [L+22], and the papers cited throughout.*

AI Disclosure: *These notes were drafted and revised with the assistance of AI tools. I have reviewed the content and take responsibility for it. If you find an error, please let me know.*

Most of you have used a system like ChatGPT, Claude, or Gemini. You type in a question and get a reply:

"What is the capital of Pennsylvania?" $\longrightarrow$ "Harrisburg."

These notes give a high-level view of how language models are defined, trained, and used to generate responses.

1 Defining a Language Model

Consider the prefix *The capital of Pennsylvania is*. It could be followed by *Harrisburg*, "a city on the Susquehanna River," or countless other continuations, but we should not treat these choices equally. Language modeling handles this by using a probability distribution over the possible continuations. A good model assigns more probability to *Harrisburg* than to *Philadelphia*, and we can use these probabilities either to compare candidate continuations or to generate one.

1.1 Tokens and Sequence Probabilities

To define this probability distribution, we first need to say what its outcomes are. Most contemporary language models use a **tokenizer** to map text into tokens drawn from a fixed, finite set called the **vocabulary** $\mathcal{V}$. Using whole words would require an enormous vocabulary and would still leave us with unfamiliar words, while using individual characters or bytes would make common words unnecessarily long. Subword tokenizers are a compromise: frequent words may remain intact, and less frequent words can be assembled from reusable pieces. The model then assigns probabilities to sequences $x_{1:T}$ with each $x_t \in \mathcal{V}$. Different tokenizers can represent the same text with different token sequences.

For example, the original GPT-3 models used the GPT-2 byte-level byte-pair encoding (BPE) tokenizer,[1] called `r50k_base` in `tiktoken`. It tokenizes "The capital of Pennsylvania is Harrisburg." as

$$\underbrace{\texttt{The}}_{464} \mid \underbrace{\texttt{␣capital}}_{3139} \mid \underbrace{\texttt{␣of}}_{286} \mid \underbrace{\texttt{␣Pennsylvania}}_{9589} \mid \underbrace{\texttt{␣is}}_{318} \mid \underbrace{\texttt{␣Harris}}_{10026} \mid \underbrace{\texttt{burg}}_{7423} \mid \underbrace{\texttt{␣.}}_{13}.$$

[1] Byte-level BPE starts from bytes and repeatedly adds tokens for frequent adjacent pairs. Common strings can therefore become single tokens while any byte string remains representable. See https://www.cs.cornell.edu/courses/cs4782/2026sp/demos/bytepair/ for an interactive walkthrough.

The visible space marks a space at the beginning of a token, while the number below is its identifier. Notice that *Harrisburg* is split even though *Pennsylvania* is not. A vocabulary may also contain reserved structural tokens, such as an end-of-sequence token used to signal that generation should stop.

The classic definition of a **language model** is a probability distribution p over token sequences. For any fixed length T, the model assigns a probability to each sequence $x_{1:T} = (x_1, \ldots, x_T) \in \mathcal{V}^T$:

$$p(x_{1:T}) \in [0, 1], \qquad \sum_{x_{1:T} \in \mathcal{V}^T} p(x_{1:T}) = 1.$$

This distribution also determines probabilities for the possible continuations of a prefix. After *The capital of Pennsylvania is*, a useful model assigns more probability to *Harrisburg* than to *Philadelphia*, just as it assigns more probability to a grammatical sentence than to a scrambled version. To make these distinctions, the model has to learn grammar, facts about the world, and common ways in which sentences and documents are organized.

1.2 Autoregressive Factorization

There are $|\mathcal{V}|^T$ possible sequences of length T, which is about 10^{470} when $|\mathcal{V}| = 50{,}000$ and $T = 100$. We cannot store a separate probability for every one of them. Instead, the probability **chain rule** expresses a sequence probability as a product of next-token probabilities. Let $x_{<t} = (x_1, \ldots, x_{t-1})$ denote the prefix before token t, with $x_{<1}$ the empty prefix. Then

$$p(x_{1:T}) = \prod_{t=1}^{T} p(x_t \mid x_{<t}). \tag{1}$$

Equation (1) is the ordinary chain rule of probability. It rewrites a joint sequence probability as the same prediction problem at every position: given the preceding tokens $x_{<t}$, predict the next token x_t. A model organized around these left-to-right conditional distributions is called **autoregressive** because it predicts each token from the earlier tokens in the same sequence.

1.3 Prompts and Responses

In many applications, we give the model an input sequence, called a **prompt**, and ask it to continue. Writing X for the prompt and $Y = (y_1, \ldots, y_T)$ for its continuation, or **response**, the notation $p(Y \mid X)$ denotes the probability of that response given the prompt:

$$p(Y \mid X) = \prod_{t=1}^{T} p(y_t \mid X, y_{<t}). \tag{2}$$

In the opening example, X contains the question about Pennsylvania and Y contains the tokens spelling *Harrisburg*, followed by the period. Question answering, translation, summarization, and code generation all fit this form: the prompt specifies the request, and the response is built one token at a time.

1.4 Generating a Response

We can use Equation (2) to generate a response from left to right. At position t, the model supplies the next-token distribution $p(\cdot \mid X, y_{<t})$. We still need a rule for selecting a token from

this distribution. A **decoding rule** makes this choice, either by selecting a likely token or by sampling.

Suppose the prompt is *The capital of Pennsylvania is.* A decoding rule might select `␣Harris`, after which that token is appended to the prefix and the model is evaluated again. Its next distribution may place high probability on `burg`, followed by a period:

$$\texttt{␣Harris} \longrightarrow \texttt{burg} \longrightarrow \texttt{.}$$

Thus *Harrisburg* is assembled through repeated next-token predictions rather than produced in one step, with each selected token becoming part of the prefix used for the next prediction. The process continues until a stopping rule ends the response.

Greedy decoding. The simplest rule selects a token with the largest next-token probability:

$$y_t \in \arg\max_{w \in \mathcal{V}} p(w \mid X, y_{<t}). \tag{3}$$

With fixed tie-breaking, greedy decoding is deterministic. It chooses the most likely token at the current position without considering how that choice will affect later probabilities, so it need not produce the complete response with the largest value of $p(Y \mid X)$.

Sampling. Sampling instead draws the next token from the model's distribution, allowing repeated generations from the same prompt to differ. A **temperature** $\tau > 0$ modifies the distribution to

$$p_\tau(w \mid X, y_{<t}) = \frac{p(w \mid X, y_{<t})^{1/\tau}}{\sum_{v \in \mathcal{V}} p(v \mid X, y_{<t})^{1/\tau}}.$$

At $\tau = 1$, this is the original distribution. A temperature below one concentrates more probability on likely tokens, while a temperature above one makes the probabilities more even. As τ approaches zero, the distribution concentrates on the most likely token and approaches greedy decoding when that token is unique.

Restricting the candidates. **Top-k** decoding samples only from the k tokens with the largest probabilities. **Top-p** decoding instead orders tokens from most to least likely and retains the smallest set whose total probability reaches a chosen threshold [HBD+20]. After either cutoff, the retained probabilities are rescaled to sum to one before sampling.

Stopping. Generation ends when the model selects an end-of-sequence or end-of-turn token, when a designated stop string appears, or when a length limit is reached. The token-selection and stopping rules together determine the distribution over complete responses.

2 Representing Next-Token Probabilities

A language model still needs a way to compute these probabilities. Given a prefix, it must assign a probability to every candidate next token $w \in \mathcal{V}$. We will first see how this works in count-based models and then how neural language models share what they learn across different prefixes.

2.1 Count-Based Models and Their Limits

The simplest way to estimate a next-token distribution is to count how often each token appears after a given context. The context matters because language is not formed by choosing each symbol independently: Markov demonstrated this by showing that the chance of seeing a vowel or consonant in a passage of Pushkin depended on the preceding symbol [Mar13]. Shannon later turned the same observation into a method for generating text. He estimated continuation frequencies after short character or word contexts and sampled from them. As he used longer contexts, the generated text became more recognizable [Sha48].

For example, Shannon's first-order word approximation sampled words independently according to their overall frequencies and produced

representing and speedily is an good apt.

When he instead conditioned each word on the preceding word, one sample began:

the head and in frontal attack on an English writer.

Neither passage is coherent, but the second contains more recognizable local phrases because adjacent words are no longer chosen independently.

n-gram models. An n-gram model formalizes this idea by retaining the most recent $n-1$ tokens as its context. For $t \geq n$, it predicts x_t from the suffix $x_{t-n+1:t-1} = (x_{t-n+1}, \ldots, x_{t-1})$, discarding all earlier tokens. The first $n-1$ positions use their available shorter prefixes.

For any context value $c \in \mathcal{V}^{n-1}$, let $N(c, w)$ count the corpus positions at which c is followed by token w, and let $N(c) = \sum_{v \in \mathcal{V}} N(c, v)$. Counts remain within document boundaries. For $N(c) > 0$, the empirical next-token distribution is

$$\widehat{p}(w \mid c) = \frac{N(c, w)}{N(c)}. \tag{4}$$

Equation (4) is the maximum-likelihood estimate for context c: among all next-token distributions, it assigns the greatest likelihood to the observed continuations. Used directly, however, it assigns probability zero to a token that was never observed after c, even if that token is a valid continuation. Count estimates are therefore usually *smoothed*, which gives some probability to events not observed in the corpus [JM26].[2]

Choosing the context length. A small n provides many observations per context but assumes that earlier tokens are irrelevant, an assumption that often fails. For example, consider the following sentence,

The location of the Pennsylvania records stored at the Philadelphia archive ____ unknown.

The correct continuation is *is*, not *are*. The relevant number information comes from *location*, which may lie outside a short window.

Increasing n retains more context, but it also gives us fewer repeated observations of each context. Since each of the $n-1$ positions can contain any token in $\mathcal{V}$, there are $|\mathcal{V}|^{n-1}$ possible full-length contexts; when $n = 3$, for example, the two positions give $|\mathcal{V}|^2$ possibilities. This number grows exponentially with n, so most long contexts appear only once in a finite corpus and Equation (4) simply memorizes their observed continuations. This is the familiar bias–variance trade-off: short contexts may discard relevant information, while long contexts provide too few examples for reliable probability estimates.

[2] Add-α smoothing uses $(N(c, w) + \alpha)/(N(c) + \alpha|\mathcal{V}|)$ for $\alpha > 0$, giving every token positive probability.

2.2 Neural Language Models: Sharing Across Contexts

Count tables treat every complete context as a separate entry. For example, "Pennsylvania's largest city is" and "The largest city in Pennsylvania is" lead to different rows of a count table even though both can be followed by *Philadelphia*. Observing one context therefore does not directly improve the estimate for the other. A **neural language model** instead uses learned representations and one prediction rule whose parameters are shared across contexts and positions. We write this rule as p_θ, where θ denotes its parameters. Evidence from one context can then affect predictions in other contexts that the model represents similarly.

Figure 1 compares the three neural architectures we discuss below. They differ in how they use the prefix to construct a representation for the next prediction.

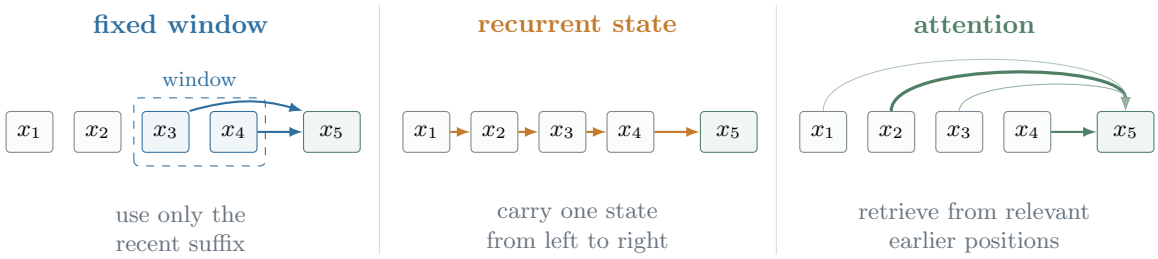

Figure 1: Three high-level strategies for using a prefix. A fixed-window model uses a suffix, a recurrent model carries a state, and attention retrieves information from earlier positions.

Fixed-window neural models. Bengio et al. [BDVJ03] proposed an influential model that retained the n-gram idea of using the previous $n-1$ words but replaced the count table with a feed-forward neural network. The model represents the words in the window as vectors, combines them into a representation of the context, and uses that representation to score every possible next word.

Token identifiers are only labels. The identifiers assigned to *Philadelphia* and *Pennsylvania*, for example, do not tell the network that the words are related. An **embedding matrix** $E \in \mathbb{R}^{|\mathcal{V}| \times d_e}$ associates every word w with a learned vector $e_w \in \mathbb{R}^{d_e}$. Training adjusts these vectors along with the other model weights, so words that are useful in similar predictions can acquire representations that the network handles similarly.

At position t, the recent embeddings are concatenated into

$$z_t = (e_{x_{t-n+1}}, \ldots, e_{x_{t-1}}) \in \mathbb{R}^{(n-1)d_e},$$

and a feed-forward network computes a contextual representation

$$h_t = f_\theta(z_t) \in \mathbb{R}^{d_h}.$$

Concatenation preserves the position of each word because the embedding from each relative position occupies a different part of z_t. If the window contains the four words *at the Philadelphia archive*, then z_t places their four embeddings in that order. The network combines them into h_t, a summary used for the current prediction. The embedding dimension d_e and contextual dimension d_h describe different representations and need not be equal.

To turn h_t into a next-word distribution, the model assigns each candidate word w a score $u_w^\top h_t + b_w$. Here u_w is the row for w in an output matrix $U \in \mathbb{R}^{|\mathcal{V}| \times d_h}$, and b_w is a learned bias. The

embedding e_w represents w when it appears in the context, whereas u_w helps score w as a possible continuation. A softmax converts all of the scores into probabilities:

$$p_\theta(w \mid x_{<t}) = \frac{\exp(u_w^\top h_t + b_w)}{\sum_{v \in \mathcal{V}} \exp(u_v^\top h_t + b_v)}. \tag{5}$$

The softmax produces nonnegative probabilities that sum to one across the vocabulary. A candidate with a larger score receives a larger probability. Because the same embeddings and network weights are used for every prediction, learning that *is* commonly follows contexts involving a singular location can affect more than one exact context. However, a model whose window contains only *at the Philadelphia archive* cannot use the earlier word *location* when choosing between *is* and *are*.

Why not simply choose a much larger window? The limitation comes from feeding the entire concatenated window into a feed-forward network as one fixed-size vector. Every additional position adds another d_e coordinates to z_t and requires more weights in the layers connected to that input. In the architecture of Bengio et al., the number of parameters therefore grew linearly with the window length, increasing the memory and computation required for training. A larger window was possible, but more expensive, and its length was built into the shape of the network.

Recurrent neural models. A recurrent model avoids choosing a fixed window by carrying a representation of the prefix from left to right. Starting from a fixed or learned initial state, it combines each word's embedding with the previous representation to produce a new one:

$$h_t = g_\theta(h_{t-1}, e_{x_{t-1}}) \in \mathbb{R}^{d_h}, \qquad t \geq 2,$$

where the same update rule g_θ is used at every position. The representation h_t summarizes the prefix $x_{<t}$ and supplies the input to the same next-word softmax in Equation (5). In the running example, reading *location* changes the recurrent state, and every subsequent word updates that state again. When the model reaches the blank before *unknown*, its choice between *is* and *are* can still depend on information carried forward from *location*.

This removes a fixed cutoff, but it does not make long-range dependence effortless. Information about *location* must survive every update caused by the intervening words, so it can be weakened or overwritten before it is needed. Improved recurrent architectures, particularly long short-term memory (LSTM) networks, made such information easier to preserve and became the dominant neural approach to sequence modeling during much of the 2010s [Elm90, HS97, MKB+10, SVL14]. Recurrent models nevertheless process tokens one after another, which limits parallelism during training and makes the computational path between distant words grow with their separation.

Attention-based models. Attention provides a more direct way to use earlier information. Instead of requiring information to pass through every intervening recurrent update, the model considers the earlier positions when constructing the representation for the current prediction. It uses a learned comparison rule to give each earlier position a relevance score, then normalizes these scores into weights. For $t \geq 2$, a single attention operation forms

$$a_t = \sum_{s<t} \alpha_{ts} v_s, \qquad \alpha_{ts} \geq 0, \qquad \sum_{s<t} \alpha_{ts} = 1.$$

Here $v_s \in \mathbb{R}^{d_h}$ contains information from position s, while α_{ts} is its attention weight for the prediction at position t. The weights are nonnegative and sum to one, so a_t is a weighted combination of the information available at the earlier positions.

In the running example, the model can assign a large weight to the position containing *location* when it predicts the word before *unknown*. The information from that position then contributes directly to a_t, even though many words appear between *location* and the blank. Recall that the fixed-window network receives the concatenated window as one input, so adding another context position changes the input dimension. Attention instead applies the same learned comparison rule to every available position. The same parameters can process prefixes of different lengths, and increasing the context length does not require adding parameters for the new positions. It still requires more computation and memory, and implementations impose a maximum supported context length.

Attention also avoids the sequential updates required by a recurrent model. When a complete training sequence is available, the model can compute representations for many positions in parallel and make effective use of GPUs. Modern neural language models usually use the **Transformer** architecture, which builds on attention and adds representations of token positions and feed-forward neural-network computations [V+17]. For now, we will treat a Transformer as a shared neural prediction rule that gathers information from the available prefix and produces a next-token distribution. We will study its attention heads and other components later in the course.

Today, the term **large language model** (LLM) usually refers to a neural language model with many parameters trained on a large collection of text, although there is no fixed parameter count or data threshold at which a model becomes "large."

3 Pretraining

The architectures above compute next-token probabilities once the parameters θ are fixed. We now need to learn those parameters.

The pretraining corpus. Pretraining uses a large collection of documents assembled from sources such as web pages, books, Wikipedia, news, and code, then filtered, deduplicated, mixed, and tokenized. GPT-3, for example, used 300 billion tokens sampled from filtered Common Crawl, WebText2, two book corpora, and English-language Wikipedia [BMR+20]. Disclosure varies across models: the GPT-4 report names publicly available internet data and licensed data but does not describe the dataset construction or exact mixture [Ope23].

Source	Tokens in corpus	Sampling weight
Filtered Common Crawl	410 billion	60%
WebText2	19 billion	22%
Books1	12 billion	8%
Books2	55 billion	8%
English Wikipedia	3 billion	3%

Table 1: The GPT-3 pretraining mixture, adapted from Table 2.2 of Brown et al. [BMR+20]. Sampling weights determine how often training examples are drawn from each source and are not proportional to the number of available tokens. The reported percentages are rounded.

A sequence supplies many training examples. Once tokenized, each document supplies many prediction problems: the model receives a prefix and is trained to predict the token that the document actually contains next, which is called the **target**. Suppose that each word in

Philadelphia is in Pennsylvania is one token. Moving through the sequence gives

$$\begin{aligned} \textit{Philadelphia} &\longrightarrow \textit{is}, \\ \textit{Philadelphia is} &\longrightarrow \textit{in}, \\ \textit{Philadelphia is in} &\longrightarrow \textit{Pennsylvania}. \end{aligned}$$

The prefix appears on the left and the observed target on the right. Every retained corpus position can provide an example, and no separate annotation is needed because the target already appears in the original text. This is why the training is called **self-supervised**.[3]

From token losses to a training objective. Training evaluates each observed target x_t using the **log loss** $-\log p_\theta(x_t \mid x_{<t})$. Assigning probabilities 0.9 and 0.01 gives losses of about 0.11 and 4.61, respectively, so a confident mismatch incurs a large loss.

The loss of a training sequence is the sum of its token losses. The chain rule makes this sum equal to the negative log probability of the sequence:

$$L(x_{1:T};\theta) = \sum_{t=1}^{T} -\log p_\theta(x_t \mid x_{<t}) = -\log p_\theta(x_{1:T}). \tag{6}$$

The left side adds the loss at every prediction point. By the chain rule, this is also the negative log probability that the model assigns to the complete sequence, as shown on the right.

For a corpus of D documents $x^{(i)} = x^{(i)}_{1:T_i}$, pretraining minimizes the sum of their sequence losses:

$$\min_\theta \ \mathcal{L}_{\text{pre}}(\theta), \qquad \mathcal{L}_{\text{pre}}(\theta) = \sum_{i=1}^{D} L(x^{(i)};\theta). \tag{7}$$

Because each sequence loss is a negative log probability, minimizing Equation (7) is equivalent to maximizing the log probabilities of the observed training sequences. This is the **maximum-likelihood** objective. An optimization algorithm repeatedly adjusts θ to reduce $\mathcal{L}_{\text{pre}}$, producing a pretrained model, also called a **base model**.

What is learned from the objective. The objective itself only measures how well the model predicts the corpus. Since the same parameters are used at every position, learning a recurring pattern can improve many predictions. Such patterns include grammatical regularities, facts expressed repeatedly in the documents, and common ways of organizing text or code. Grammar, knowledge, and programming are not separate targets in the loss; the model learns them when they help predict tokens across many examples.

Predictive fit is not the same as truth. The corpus supplies examples of text rather than claims labeled true or false, so false or contradictory statements in the data can also receive probability. A low loss on the finite training corpus likewise does not guarantee equally good predictions on new text.

Mid-training. After broad pretraining, some models continue next-token training on a more targeted mixture chosen for data quality, particular domains, or longer contexts. This optional stage is often called **mid-training**. The data may still be ordinary text, and each observed next

[3] Using the observed prefix at each position is called **teacher forcing**. Real systems usually divide documents into finite chunks or combine text from several documents into packed sequences, using masks to control which tokens each position may access. Each retained position still contributes a next-token target.

token remains the target, so the training looks much like pretraining. The mixture may also include examples generated by language models and then filtered or verified. These are commonly called **synthetic data**, and they can also appear during post-training. The boundary between pretraining, mid-training, and post-training is not standardized.

4 Post-Training: Learning from Targeted Feedback

A pretrained model has learned which continuations resemble its training text, but this alone does not tell it to answer an instruction. Consider the prompt

"Name the largest city in Pennsylvania."

The answer *Philadelphia* follows the instruction. A quiz document containing the same sentence might instead continue with another question, such as "Name the capital of Pennsylvania." Both continuations can resemble text in the corpus, but they serve different purposes. We need targeted feedback to make the answer more likely when the input is intended as an instruction.

Post-training continues to change θ using data chosen for particular capabilities and behaviors. It can teach instruction following, strengthen performance in domains such as mathematics or programming, adapt responses to human preferences, or discourage unwanted behavior. This feedback can take several forms: a response to imitate, a comparison between candidate responses, or a numerical evaluation of a generated response.

4.1 Demonstrations and Supervised Fine-Tuning

The most direct feedback supplies a response that the model should imitate:

$$\underbrace{\text{"Name the largest city in Pennsylvania."}}_{\text{prompt}} \quad \longrightarrow \quad \underbrace{\text{"Philadelphia."}}_{\text{desired response}}$$

A prompt paired with a selected target response is a **demonstration**. Given m demonstrations (X_i, Y_i), $i = 1, \ldots, m$, **supervised fine-tuning** (SFT) maximizes

$$\max_{\theta} \sum_{i=1}^{m} \log p_\theta(Y_i \mid X_i). \tag{8}$$

The calculation is the same teacher-forced next-token training used in pretraining. Here the prompt supplies the initial prefix, and the selected response tokens supply the targets that follow it. Thus SFT is continued language-model training on a smaller, deliberately constructed dataset. We will call it post-training because the examples are selected to teach a task or behavior, although some taxonomies place part of this training under mid-training. For a general-purpose model, SFT on many instruction–response pairs is called **instruction tuning**.

Writing a complete target response can be expensive, and an open-ended prompt may admit many good responses. A demonstration also says little about the relative quality of other candidates. Comparisons provide another form of feedback.

4.2 Preferences, Rewards, and RLHF

Comparisons are especially useful when a prompt has many reasonable responses. Consider the prompt

"In one sentence, explain why Philadelphia is historically important."

Suppose the model produces "Philadelphia was a meeting place for the Continental Congress and the Constitutional Convention" and "Philadelphia is a historic city with many important places." An evaluator may prefer the first response because it is more specific and informative, without having to write an ideal response from scratch. A **preference** example records the prompt, the preferred response, and the rejected response as a triple (X, Y^+, Y^-). This is a relative judgment: it tells us that Y^+ was preferred to Y^-, but it does not assign either response a numerical score.

From comparisons to a reward. Given many preference examples, we can train a **reward model** $r_\phi(X, Y)$ that assigns a numerical score to response Y for prompt X. Its parameters ϕ are trained so that preferred responses receive larger scores than rejected responses. The evaluators supply comparisons rather than these scores; the numerical scale is learned by the reward model. The resulting score estimates the preferences represented in the comparison data and should not be read as a direct measure of truth or quality.

The reward model can score responses that did not appear in the comparisons. Some tasks instead provide rewards directly: unit tests evaluate generated code, exact-answer checkers evaluate calculations, and environments evaluate tool calls. These are called **verifiable rewards** because they come from a specified procedure rather than learned preferences.

Using rewards to update the model. Unlike SFT, a reward does not provide a target response whose tokens the model can imitate. The model instead generates responses from its current distribution, the reward rule scores them, and training changes θ so that responses receiving larger scores become more likely. This form of learning is called **reinforcement learning**. Let $\mathcal{D}$ denote a distribution over prompts, and treat the reward rule as fixed while updating θ. The expected-reward objective is

$$\max_\theta \ \mathbb{E}_{X\sim\mathcal{D}}\mathbb{E}_{Y\sim p_\theta(\cdot|X)}[r(X, Y)]. \tag{9}$$

In **reinforcement learning from human feedback** (RLHF), we first apply SFT, collect human comparisons, train a reward model from them, and then use reinforcement learning to increase the learned reward [CLB+17, OWJ+22]. RLHF is often used for **alignment**, adapting the model to better match human intentions and constraints. Because the reward only approximates the behavior we want, optimization may favor unintended responses that score well, a failure called **reward hacking**. Rewards from answer checkers, unit tests, or other specified procedures instead lead to **reinforcement learning from verifiable rewards** (RLVR).

RL is not the only way to use preference data, since other methods train directly from preferred and rejected responses without first fitting a reward model. For now, the important distinction is that demonstrations provide responses to imitate, comparisons indicate which response is preferred, and rewards score newly generated responses for optimization.

Pretraining, mid-training, and post-training all change the model parameters, using different sources of supervision. Inference begins after those parameters have been fixed.

5 Inference with a Fixed Model

Training produces a particular setting of the parameters θ. During **inference**, a system uses those fixed parameters to make predictions or generate responses. The model supplies next-token distributions, while the system chooses the prompt, a decoding rule such as those in Section 1.4, the number of model calls, and any external computation. These choices can change the output without changing θ.

5.1 Prompting and In-Context Learning

The model receives the prompt as the initial part of its prefix, so every instruction and example in the prompt can change the response distribution even though θ stays fixed. An instruction can describe the task, while examples can demonstrate the pattern that the response should follow. Suppose the prompt gives the county containing each of two Pennsylvania cities and asks for the county containing a third:

```
City:  Philadelphia          County:  Philadelphia
City:  Pittsburgh               County:  Allegheny
City:  Harrisburg                County:  _________
```

The likely completion is *Dauphin*. The two completed rows demonstrate the desired city–county mapping, while the final row supplies the new input. Using examples in a prompt to establish a task in this way is called **in-context learning** [BMR+20]. Although the name includes "learning," no training step occurs: the parameters remain fixed, and the examples affect the response by becoming part of the prefix, so removing them can remove their effect. A prompt with no examples is called **zero-shot**, while one with a small number is called **few-shot**.

5.2 Inference-Time Computation

At inference, a system can spend additional computation by producing a longer response, generating and selecting among several responses, or incorporating information and calculations supplied by retrieval or tools. Each option requires more generated tokens or more evaluations of the fixed model before the system returns an answer.

Longer generations. In **chain-of-thought** generation, the model writes intermediate steps before its final answer [WWS+22]. For the prompt in Figure 2, it may write $67 \times 3 = 60 \times 3 + 7 \times 3$ before producing 201. Each step becomes part of $y_{<t}$, so later predictions can use the intermediate calculation, at the cost of more sequential model evaluations.

Generating and selecting candidates. Another approach uses repeated sampling to produce several complete responses. In **best-of-N** generation, the model samples N candidates, and a **verifier** selects the highest-scoring one [CKB+21]. The verifier may be a rule or another model that scores candidate answers. If the samples are independent and each is acceptable with probability α, the probability that at least one candidate is acceptable is $1-(1-\alpha)^N$. Larger N makes it more likely that the candidates contain a good answer, although returning that answer still depends on the correctness of the verifier.

Retrieval and tools. Sometimes the useful addition is information or a calculation rather than another model-generated response. Retrieval can supply a passage stating that Harrisburg is in Dauphin County, while a calculator can return 201 for 67×3. Either result becomes part of a later prefix, allowing subsequent predictions to use it without changing θ.

Figure 2 summarizes these possibilities for one arithmetic prompt.

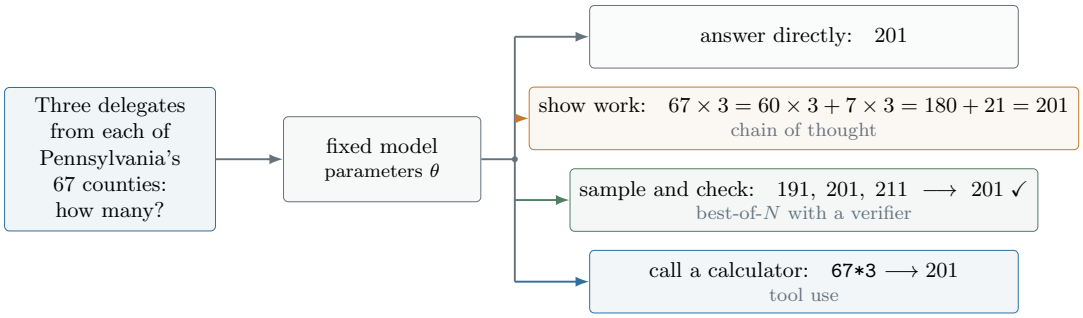

Figure 2: Four inference procedures for the same prompt: direct decoding, intermediate steps, candidate verification, and tool use.

The common point is that the model parameters stay fixed. The prompt, the length and number of generated responses, the selection rule, and any external tools can still change. Two systems built around the same language model can therefore produce different response distributions.

6 The Language-Model Pipeline

Figure 3 brings together the four stages described above. Broad pretraining produces a base model, optional mid-training continues next-token learning on a targeted mixture, and post-training uses demonstrations or feedback chosen for particular behaviors. Inference uses the resulting model without changing its parameters. The boundaries between these stages are not standardized, and some pipelines have no separate mid-training stage.

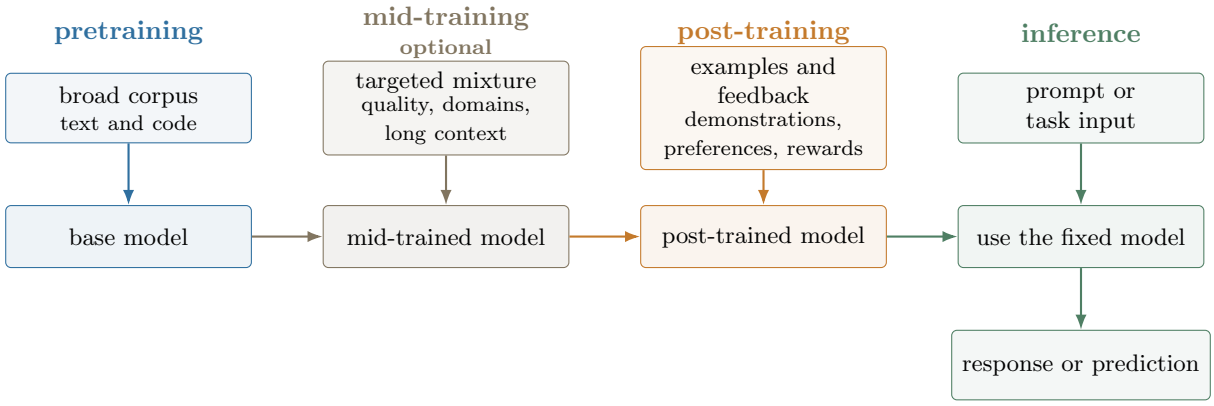

Figure 3: A high-level language-model pipeline. Mid-training is optional.

The main dividing line is between the first three stages, which change θ, and inference, which leaves θ fixed. In the opening example, training determines whether *Harrisburg* and an instruction-style answer are likely. Inference changes the prompt, decoding rule, amount of computation, or available tools to determine how the response is constructed and selected.

The pipeline describes what is done, but it does not explain why these stages produce particular capabilities, when those capabilities persist, or how they fail. Experiments can establish that a behavior occurs for a particular model, while theory seeks the mechanism and the conditions under which the behavior extends to new settings. This matters for safety because understanding the mechanism may help us anticipate failures that testing has not revealed.

Modern language models also raise new technical questions about how next-token prediction produces broader capabilities, how generated tokens provide computation, and how feedback changes a pretrained model. These problems are interesting in their own right, and theory lets us compare possible explanations and identify both possibilities and limits. A first-principles understanding can also guide the design of new algorithms, mathematical frameworks, training objectives, architectures, and inference methods. The rest of the course will study these questions:

- **Language generation and hallucination.** What can be learned from data that only shows valid text? What should we demand of a generator beyond correctness, and why can a model that fits its data well still hallucinate?
- **Next-token prediction and pretraining.** What does the next-token objective measure, when can it produce broader capabilities, and what does the training loss leave unmeasured?
- **Transformer architecture and computation.** What computations can a Transformer represent? How do resources such as depth, precision, sequence length, and generated intermediate tokens change its computational power?
- **Scaling laws.** Why does loss vary predictably with model size, data, and compute? What compute-optimal tradeoff does this imply, and what might explain these empirical laws?
- **In-context learning.** How can a fixed pretrained model infer a new task from a few examples in its prompt? What mechanisms can explain this behavior?
- **Reasoning, search, and verification.** When do intermediate reasoning steps, repeated sampling, and verification improve the use of a fixed model, and what limits remain?
- **Post-training and learning from feedback.** How much can feedback change a model after pretraining? What can demonstrations, preference comparisons, reward models, and reinforcement learning teach, and when can optimizing imperfect feedback produce unintended behavior?

Taken together, these questions ask how the choices made across the pipeline interact to determine what a language model can do, where it fails, and which conclusions extend beyond a particular model or dataset.

References

[BDVJ03] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003. https://www.jmlr.org/papers/v3/bengio03a.html.

[BMR+20] Tom B. Brown, Benjamin Mann, Nick Ryder, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, 2020. https://arxiv.org/abs/2005.14165.

[CKB+21] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, et al. Training verifiers to solve math word problems, 2021. arXiv:2110.14168. https://arxiv.org/abs/2110.14168.

[CLB+17] Paul F. Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. In *Advances in Neural Information Processing Systems*, 2017. https://arxiv.org/abs/1706.03741.

[Elm90] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990. https://doi.org/10.1207/s15516709cog1402_1.

[HBD+20] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020. https://arxiv.org/abs/1904.09751.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. https://doi.org/10.1162/neco.1997.9.8.1735.

[JM26] Daniel Jurafsky and James H. Martin. Speech and language processing, 2026. Third edition draft. https://web.stanford.edu/~jurafsky/slp3/.

[L+22] Percy Liang et al. CS324: Large language models. Stanford University course notes, 2022. https://stanford-cs324.github.io/winter2022/lectures/.

[Mar13] Andrey A. Markov. An example of statistical investigation of the text *Eugene Onegin* concerning the connection of samples in chains. *Bulletin of the Imperial Academy of Sciences of St. Petersburg*, 7(3):153–162, 1913. https://www.mathnet.ru/eng/im6612.

[MKB+10] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Proceedings of Interspeech*, pages 1045–1048, 2010. https://doi.org/10.21437/Interspeech.2010-343.

[Ope23] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. https://arxiv.org/abs/2303.08774.

[OWJ+22] Long Ouyang, Jeff Wu, Xu Jiang, et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, 2022. https://arxiv.org/abs/2203.02155.

[Sha48] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948. https://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, 2014. https://arxiv.org/abs/1409.3215.

[V+17] Ashish Vaswani et al. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017. https://arxiv.org/abs/1706.03762.

[WWS+22] Jason Wei, Xuezhi Wang, Dale Schuurmans, et al. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022. https://arxiv.org/abs/2201.11903.