LLM Inference

From Prompt to Token-by-Token Generation

INFERENCE PIPELINE

Topic: Deploying an LLM

- AI is eating the world.
- The current forms of AI use **large language models**.
- These are neural networks with billions of parameters that process text and images.
- We’re going to see how modern LLM **inference engines** trade space for time to achieve significant cost savings.
- This is a complicated but realistic example of the same principle behind the Fibonacci memoization.

LLM Inference

- **Inference** is the runtime process that turns an input **prompt** into output tokens from a trained model.
- Modern systems spend substantial resources on inference because deployed models answer requests far more often than they are retrained.
- An algorithms perspective matters because serving quality depends on how runtime cost grows with prompt length, output length, and model size.

Tensors as Inputs

- Inside the model, inputs are represented as **tensors**, which are numerical arrays with well-defined dimensions.
- A scalar has zero dimensions, a vector has one dimension, a matrix has two dimensions, and higher-dimensional arrays are also called tensors.
  - `7` is a scalar.
  - `[2, 5, 9]` is a vector.
  - A table of numbers with rows and columns is a matrix.
- In this lecture, tensor **shapes** such as `B x T x d` tell us how much data is being processed and which input-size parameters drive complexity.
- You can think of tensor shape as the structured size information that an algorithms analysis needs before counting work.

Batches and Batching

- A **batch** is a collection of multiple sequences processed together in one pass through the model.
- In training, batching is standard because many examples are available at once and parallel hardware is used to update the model efficiently.
  - A training step might process 64 or 128 examples together.
- In inference, batching is optional and depends on the serving setting: one request may run alone, or several requests may be grouped together to improve hardware utilization.
  - Interactive use may have `B = 1`.
  - A busy server may group many requests into one batch.
- This is why tensor shapes use `B` as a leading dimension: `B` counts how many sequences are being processed together at that moment.

Training vs Inference

- **Training** updates model parameters by processing many examples and computing gradients.
- Inference keeps the parameters fixed and computes predictions for one request or one batch of requests.
- *The central runtime question is not how to learn better weights, but how to produce outputs efficiently once the model already exists.*
  - What work must the model perform before it can emit the first token?
  - What work repeats every time a new token is generated?
  - Which parts of the runtime grow with the length of the existing context, and which tradeoffs improve speed by spending more memory?

INFERENCE PROCESS

Raw Text to Tokens

- Models do not read characters or words directly; they consume discrete integer identifiers called tokens.
- A **tokenization** scheme breaks text into pieces that are practical for the model’s vocabulary and training data.
  - Common tokens can correspond to whole words, subwords, punctuation marks, or whitespace patterns.
  - The string `unbelievable!` might be represented as one token in one tokenizer and several pieces in another.
- Runtime cost depends on how many tokens are produced, not on how many characters the original string contained.
  - Two prompts with similar visual length can therefore produce different inference costs.

Context as Input

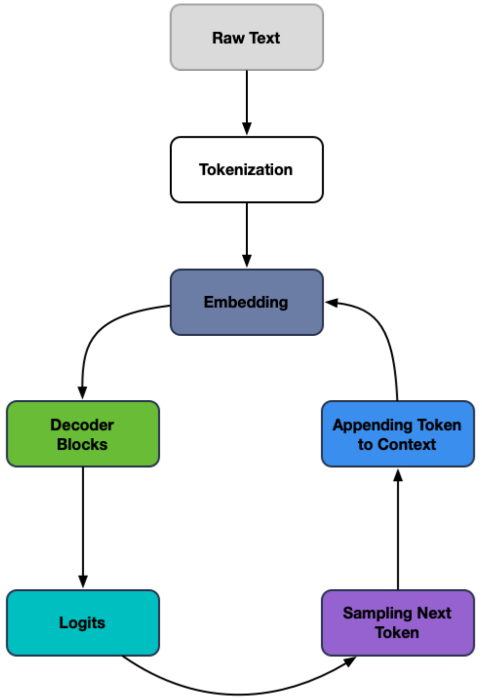
- The input to a decoder-only language model is an ordered token sequence called the **context window**.
    - Each model typically has a fixed maximum context length that is built into the architecture and reflected in how the model was trained.
    - As a representative current long-context example, OpenAI's `gpt-4.1` is documented with a `1,047,576` token context window, which is roughly `786,000` words or about `4.2` million characters using standard English token-count rules of thumb.
- At each generation step, the model uses the current context to predict one distribution for the next position.
- The context includes the original prompt plus every generated token that has been appended so far.
    - Prompt tokens are part of the context.
    - Previously generated tokens are also part of the context.

Sequence Length Matters

- Longer contexts often improve usefulness because they carry more instructions, examples, and conversation history.
- Longer contexts also increase runtime because the model must process more positions before producing output.
- In inference, capability and cost are coupled through sequence length.

Pipeline Overview

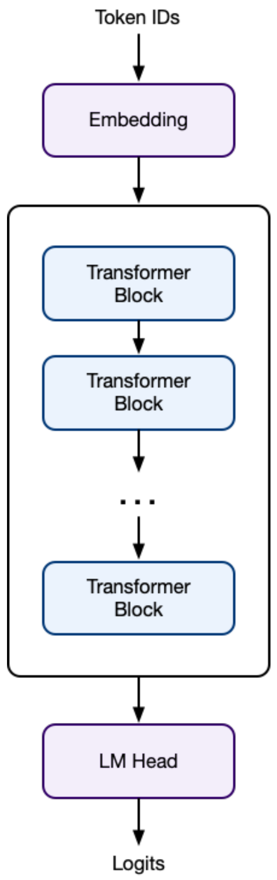
- A request begins as raw text, which is converted into **tokens** before the model sees it.
- Those tokens pass through embeddings and repeated transformer blocks to produce scores over the vocabulary.
- A decoding rule selects the next token, appends it to the context, and repeats the process.
- The loop ends when a stopping condition is met, such as an end token or a length limit.



MODEL COMPONENTS

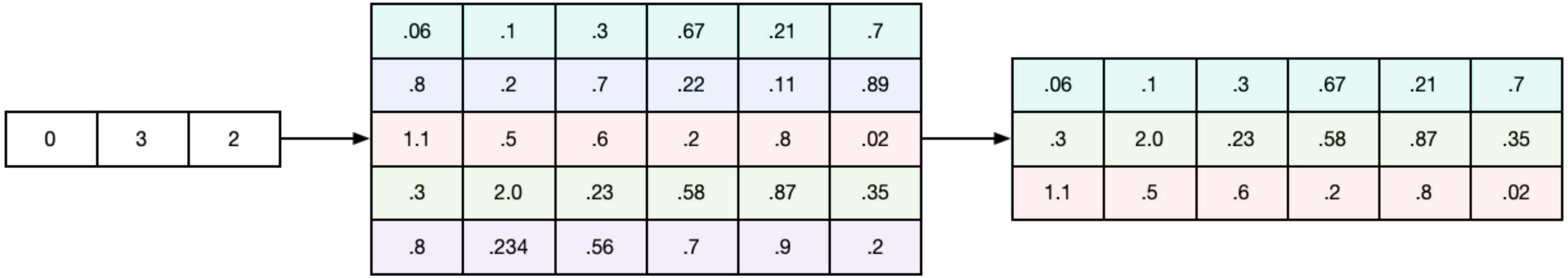
Decoder-Only View

- A **decoder-only transformer** is the dominant architecture for many large language models used for text generation.
- The model reads the existing token sequence and predicts the next token, then repeats that operation autoregressively.
- For this lecture, the key architectural question is how information flows through repeated blocks during inference.



Embeddings and Positions

- Each token id is mapped to a learned vector called an **embedding**, which gives the model a numerical representation of the token.
- Positional information is added so the model can distinguish `A then B` from `B then A`.
- If a batch has size `B`, sequence length `T`, and hidden width `d`, then token ids have shape `B x T` and the embedded sequence has shape `B x T x d`.
- The result is a sequence of vectors, one per token position, ready to enter the first transformer block.



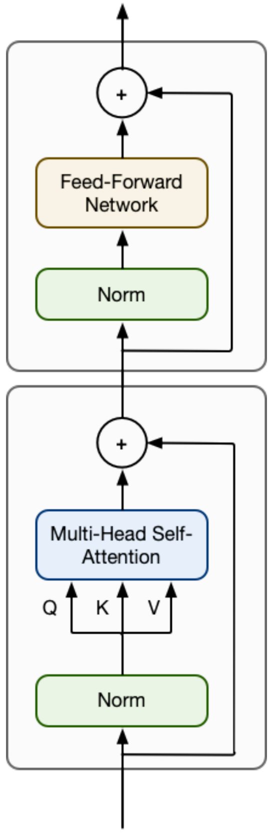
0	3	2

.06	.1	.3	.67	.21	.7
.8	.2	.7	.22	.11	.89
1.1	.5	.6	.2	.8	.02
.3	2.0	.23	.58	.87	.35
.8	.234	.56	.7	.9	.2

.06	.1	.3	.67	.21	.7
.3	2.0	.23	.58	.87	.35
1.1	.5	.6	.2	.8	.02

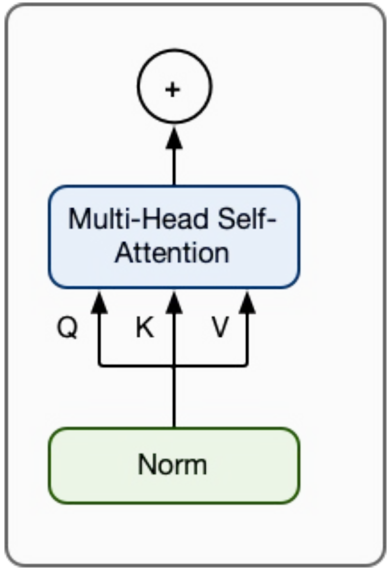
Repeated Blocks

- A transformer is built from a stack of repeated blocks rather than one giant monolithic computation.
- Each block applies the same high-level pattern: attention-style mixing across positions, followed by a position-wise feedforward transformation.
- Repetition matters algorithmically because the cost of one block is multiplied by the number of layers.
- At the level of tensor shape, one decoder block maps an input of shape `B x T x d` to an output of the same shape `B x T x d`.



Attention Purpose

- After embedding and positional encoding, the first major step inside a transformer block is the attention module.
- **Attention** lets each token position combine information from other positions in the existing context.
- This is the mechanism that allows later tokens to depend on earlier instructions, names, variables, or examples.
- Without attention, the model would struggle to condition flexibly on long sequences.

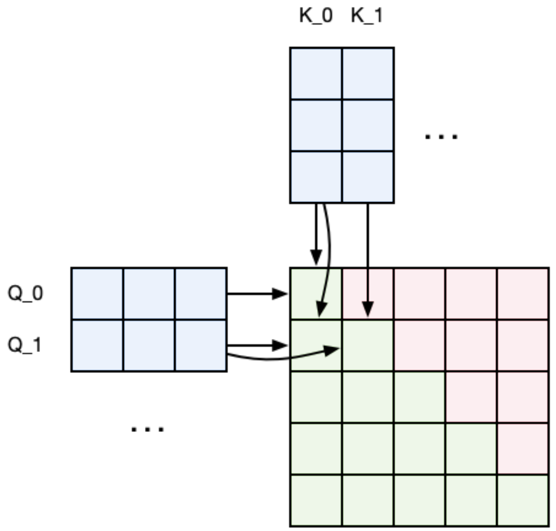


Attention Inputs

- At a high level, each position produces **query**, **key**, and **value** vectors from its current representation.
  - Query vectors ask which prior positions matter.
  - Key vectors describe what each position offers.
  - Value vectors carry content to be aggregated.
- The resulting weighted combination becomes part of the representation for that position.
- In a common **multi-head** notation with `nh` **attention heads** and per-head width `hs`, the projected tensors are often viewed as `Q, K, V` with shape `B x nh x T x hs`.

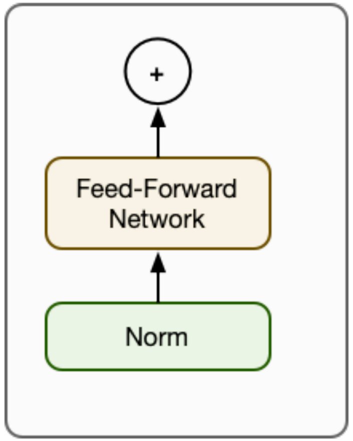
Masked Attention

- In a decoder-only model, generation uses **causal** or **masked self-attention**.
- A token at position `i` may depend on positions `1` through `i`, but not on future positions that have not been generated yet.
- This masking preserves the next-token prediction structure required for autoregressive generation.
- Before masking, attention score tensors are naturally shaped like `B x nh x T x T`, since each position is compared against every accessible position in the sequence.



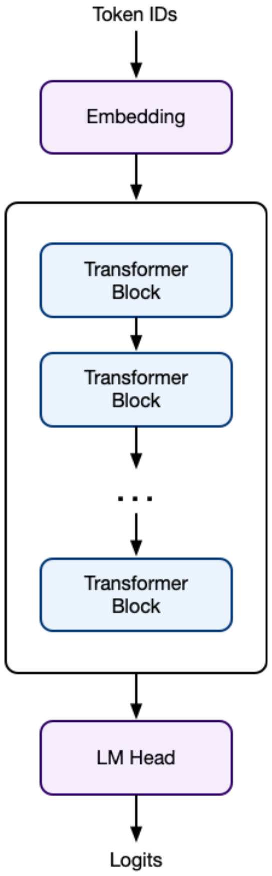
Feedforward Layers

- A **feedforward network** acts independently at each position after attention has mixed information across positions.
  - Its job is to transform the position representation through learned nonlinear mappings, not to communicate across time steps.
  - Attention mixes information across positions; feedforward layers enrich the representation at each position.
- **Residual connections** allow each block to add refined information without discarding the incoming representation.
  - The block keeps the old representation.
  - It adds a learned update on top of that representation.
- Normalization and residual paths stabilize the repeated application of many layers.
- For inference reasoning, the important point is that every block consumes and returns a full sequence of vectors.
- A common shape view is `B x T x d -> B x T x d_ff -> B x T x d`, where `d_ff` is the larger hidden width inside the feedforward sublayer.



Logits and Next-Token Prediction

- After the final block, the model projects the last-position representation into one score per vocabulary item.
  - These raw scores are called **logits**.
- For the full sequence, logits have shape `B x T x V`, and next-token generation usually reads the last-position slice with shape `B x V`.
- A higher logit means the model currently considers that token more compatible with the observed context.
- The model does not generate an entire sentence in one shot; it generates one next-token distribution, selects a token, appends it, and then solves a slightly larger next-token problem.
- This serial dependence is the reason generation behaves differently from highly parallel matrix operations inside a single forward pass.



INFERENCE PROCESS

Prefill Phase

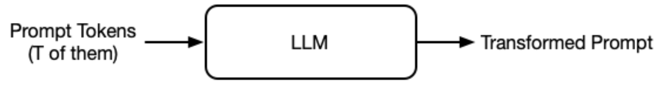
- **Prefill** is the phase that processes the initial prompt before the first generated token appears.
- Because the prompt already exists in full, many computations across prompt positions can be organized in parallel.
- Prefill often dominates the **time to first token** for long prompts.

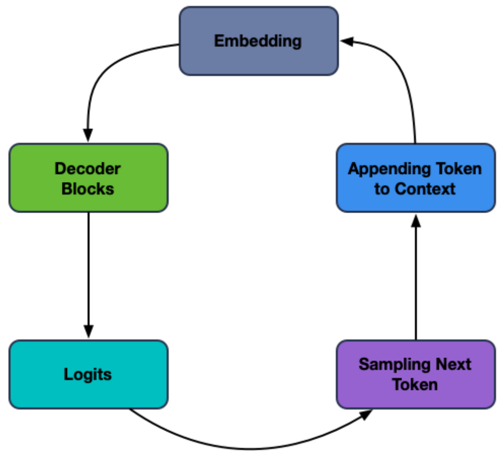
Decode Phase

- **Decode** is the phase that begins once generation starts and the model emits tokens one at a time.
- Each decode step depends on the token chosen at the previous step, so the process is *inherently sequential*.
- Decode therefore exposes a different performance bottleneck from prefill.

Prefill vs Decode

- Prefill handles a known input sequence and exploits more parallelism across positions.
- Decode handles a growing sequence and must wait for each new token before starting the next prediction.
- Good inference systems optimize these phases differently because they stress hardware and memory differently.





Greedy Decoding

- **Greedy decoding** chooses the highest-scoring token at each step.
- It is simple and deterministic once the logits are fixed.
- Greedy decoding can be fast to describe algorithmically, but it may produce repetitive or low-diversity outputs.

Sampling Choices

- **Sampling-based decoding** draws from a controlled distribution instead of always taking the top-scoring token.
- **Temperature**, **top-k**, and **top-p sampling** change which candidates remain plausible at each step.
  - These choices affect output variety and can also affect runtime details, especially in serving implementations.
  - The same prompt can produce different outputs when stochastic decoding is used.
- From an algorithms perspective, the main runtime structure is unchanged: the system still computes logits and selects one token per step.
- The policy for choosing that token changes the output distribution more than the asymptotic shape of the loop.

Stopping Rules

- Generation stops when the model emits a designated end token, reaches a maximum length, or hits an application-specific stopping rule.
- The runtime cost of one request therefore depends on both the input length and the generated output length.
- End-to-end analysis must account for both quantities.

Latency Measures

- One request can be decomposed into prompt arrival, tokenization, prefill, repeated decode steps, and completion.
- **Latency** measures how long one request takes from arrival to visible progress or completion.
- Important latency views include:
  - time to first token, which captures how expensive prompt processing and initial model setup feel to a user
  - time per output token, which captures how quickly the sequential decode loop proceeds once generation starts

Throughput and Batching

- **Throughput** measures aggregate work completed per unit time, such as requests per second or tokens per second.
  - A system can increase throughput by keeping hardware busy even if some individual requests wait longer.
- **Batching** combines multiple requests so matrix operations can use hardware more efficiently.
  - Larger batches often improve throughput but can increase queueing delay and per-request latency.
    - Better utilization helps the machine.
    - Longer waiting time can hurt one user’s request.
- This is a classic systems tradeoff: better aggregate efficiency may worsen the experience of one individual request.

Memory Traffic

- Inference cost is not only about arithmetic counts; moving model weights and activations through memory can be a major bottleneck.
- Large models repeatedly read substantial parameter data during generation.
- Practical performance therefore depends on both asymptotic operation counts and the memory system that supports those operations.

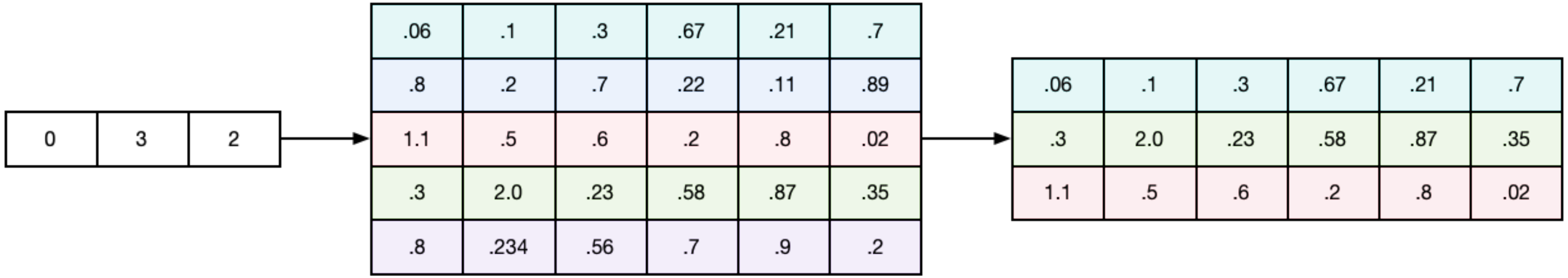
COMPLEXITY ANALYSIS

Complexity Parameters

- To analyze inference formally, define parameters clearly before deriving costs.
- Let `T` denote the current sequence length, `G` denote the number of generated output tokens, `L` denote the number of transformer blocks, `d` denote hidden width, and `V` denote vocabulary size.
  - `B` is batch size.
  - `nh` is the number of attention heads.
  - `hs` is the width of one attention head.
- The exact constants depend on implementation details, but the growth trends follow from these structural parameters and from **asymptotic complexity** analysis.

Embedding Cost

- Token lookup and positional encoding are comparatively simple operations applied once per input position or generated position.
- A coarse analysis treats this stage as linear in the number of positions being processed.
- These costs matter, but they are usually not the dominant asymptotic term in large transformer inference.



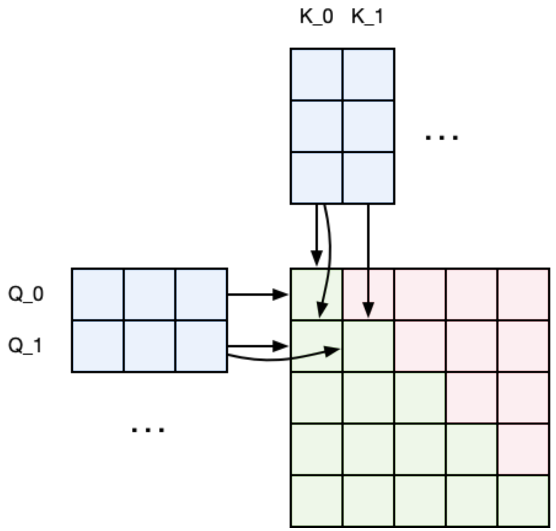
0	3	2

.06	.1	.3	.67	.21	.7
.8	.2	.7	.22	.11	.89
1.1	.5	.6	.2	.8	.02
.3	2.0	.23	.58	.87	.35
.8	.234	.56	.7	.9	.2

.06	.1	.3	.67	.21	.7
.3	2.0	.23	.58	.87	.35
1.1	.5	.6	.2	.8	.02

Attention Cost

- Within one block, attention compares positions against other accessible positions in the sequence.
- For a sequence of length `T`, the pairwise interaction structure gives a time cost on the order of `O(T^2)` for the attention score pattern in a straightforward implementation.
- Attention also requires memory for sequence-dependent intermediate data, so its space cost grows with sequence length as well.

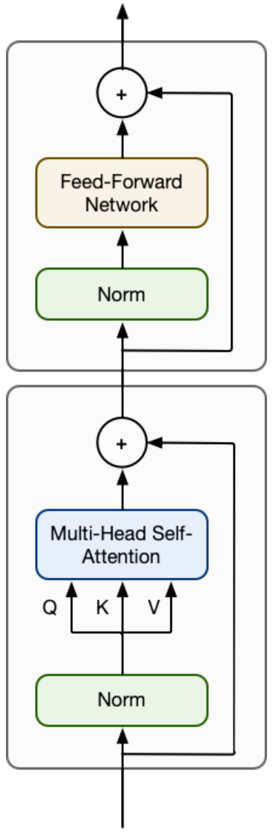


Feedforward Cost

- The feedforward sublayer processes each position independently once attention has produced updated representations.
- For fixed hidden widths, this contribution scales linearly with sequence length across positions.
- In coarse asymptotic terms, feedforward work is `O(T)` in sequence length per block when model width is treated as fixed, or `O(T d^2)` when width remains explicit.

One Block Cost

- One decoder block combines an attention term with a feedforward term.
- If sequence length is the dominant variable and width is treated as fixed, the quadratic attention contribution dominates for long contexts.
- If width and layer count are kept explicit, one block cost should be written in terms that preserve both sequence growth and model-size growth.



Full Prefill Cost

- Prefill runs all `L` blocks over the entire prompt of length `T`.
- A coarse sequence-length view gives `O(L T^2)` time when attention dominates and width is treated as fixed.
- A more detailed expression can keep width terms explicit, but the key lesson is that longer prompts *substantially increase prefill cost*.

One Decode Step

- During decode step `i`, the new token attends to the previously available context rather than reintroducing an entirely separate problem.
- Even so, the amount of relevant prior context grows as generation continues, so the work per step increases with the current sequence length in a naive implementation.
- *This is the structural reason decode can become expensive over long generations.*

Full Generation Cost

- Suppose the initial prompt has length `T_0` and the system generates `G` new tokens.
- A naive analysis sums the cost of the decode steps over contexts of lengths `T_0, T_0+1, ..., T_0+G−1`, which produces growth larger than a single linear pass.
  - Early decode steps run on shorter contexts.
  - Later decode steps run on longer contexts.
- *This cumulative cost is exactly what makes reuse of prior computation such an important optimization target.*

Example Setup

- Consider a prompt of length `T_0 = 100` tokens and a response of length `G = 20` tokens.
- The system first performs one prefill pass over 100 positions.
- It then performs 20 decode steps over contexts whose lengths grow from 100 up to 119 tokens.

Example

- Prefill
  - In the prefill phase, all 100 prompt positions participate in the forward pass through every block.
  - The attention part of that work depends on interactions among those prompt positions, so the dominant term grows quadratically with prompt length.
  - This phase determines how much useful state the model builds before emitting the first output token.
- Decode
  - The first decode step uses the 100-token context, the next step uses 101 tokens, and the pattern continues.
  - Even when each step seems locally small, the growing sequence length makes the total decode work accumulate significantly.
  - Summing increasing costs over many steps reveals why runtime can grow quickly.

Reuse as a Goal

- The runtime repeatedly needs information derived from earlier tokens in order to produce later ones.
- A natural algorithmic question is whether the system can retain useful intermediate results instead of rebuilding them from scratch at every step.
- Once that question is visible, inference starts to look like a time-space tradeoff problem rather than only a black-box prediction task.

Naive Repetition

- In a naive mental model, every decode step seems to revisit a growing prefix and redo work that is closely related to previous steps.
- That repetition is expensive because later steps depend on nearly the same history plus one new token.
- Repeated dependence on an only slightly changed state is often a signal that caching may help.

Caching Intuition

- Caching is useful when past work remains relevant to future queries and can be stored more cheaply than it can be recomputed.
- The challenge is deciding which intermediate results remain valid as the sequence grows.
- The memory used for caching is not free, so any speedup must be evaluated as a tradeoff rather than as a pure improvement.
    - More saved state can reduce repeated work.
    - More saved state also increases memory usage.

Time-Space Tradeoffs

- Many algorithmic optimizations reduce running time by storing additional state.
- **Dynamic programming**, **memoization**, and lookup tables all follow this pattern in familiar settings.
- LLM inference creates a modern systems instance of the same classical idea.

Main Takeaways

- Inference is a repeated next-token prediction process built on tokenization, transformer blocks, and a decoding loop.
- Prefill and decode have different performance structures, and both must be analyzed to understand serving cost.
- Attention over longer contexts is a major source of runtime growth, which makes asymptotic reasoning directly relevant.
- Once repeated dependence on prior computation is visible, caching becomes the natural next question.

Next Lecture

- The next step is to ask exactly what information from prior steps can be retained safely and reused.
- That question leads to memoization and the **KV cache** as a concrete **time-space tradeoff** inside decoder-only transformer inference.
- The key idea will be to reduce repeated work during decode by storing structured intermediate results from earlier tokens.