Transformers for Signal Prediction

EE269: Signal Processing and Quantization for Machine Learning
Winter 2026

The Question

Given a sequence of tokens $t_0, t_1, \ldots, t_{n-1}$,

predict the next token t_n.

- Tokens are discrete
- We can quantize our signals to get discrete tokens
- There are other methods for *tokenization*

The Question

Given a sequence of tokens $t_0, t_1, \ldots, t_{n-1}$,

predict the next token t_n.

- Tokens are discrete
- We can quantize our signals to get discrete tokens
- There are other methods for *tokenization*

Baselines

- Linear predictor (e.g. Least Squares)
- Deep Neural Network predictor

Transformer (GPT)

- Each position attends to previous positions
- Dynamically selects *which* past tokens matter
- "Programmable" retrieval + computation

Tokenizing Continuous Signals via Quantization

Goal: convert a real-valued signal into discrete tokens $t_0, t_1, \ldots$ where each t is an integer in $\{0, 1, \ldots, B-1\}$.

Uniform quantization (simplest tokenization).

- Choose clipping range $[x_{\min}, x_{\max}]$ and number of bins B.
- Define $\Delta = (x_{\max} - x_{\min})/(B-1)$.
  - First clip x to $x_c = \min(\max(x, x_{\min}), x_{\max})$.
  - Then compute the token:

$$t = \mathrm{round}\left(\frac{x_c - x_{\min}}{\Delta}\right) \quad \text{so that} \quad t \in \{0, 1, \ldots, B-1\}.$$

Alternative Tokenization Methods

- A. Scalar quantization
  - Uniform quantization applied to each sample.
  - Nonuniform quantization: Lloyd–Max Algorithm, compander, high-rate optimal quantizers
- B. Vector quantization
  - Group several samples into one vector and map it to a codebook index[1].
  - Used for blocks in audio and patches in images.
- C. Transform then quantize
  - Apply a transform such as DFT, DCT, or wavelets.
  - Quantize the transform coefficients to produce tokens.
- D. Learned tokenizers
  - Train a model to map signals into discrete codes from a learned codebook.
  - Produces tokens that can align better with perceptual or semantic structure[2].
- E. Event-based tokenization
  - Tokenize events such as threshold crossings, peaks, or onsets.
  - Useful when the signal changes only at a few time points.

[1] E.g. k-means algorithm

[2] E.g. KyutAI Mimi for audio and Nvidia Cosmos for images and video

From Attention to Transformers and the LLM Revolution

- **Before transformers:** sequence/signal prediction models struggled with long-range dependencies.
  - **Attention (2014–2016):** learn to focus on relevant parts of the input.
  - Key idea: compute weights over past states and take a weighted sum (soft retrieval).
- **Transformer (2017):** combine **self-attention** with fully connected MLP blocks.
  - Every position can attend to previous positions (or all positions).
  - Enables much better scaling on GPUs/TPUs: parallelism in training and efficient batching.
  - Becomes the dominant architecture for sequence prediction and language modeling.
- **GPT (2018): A transformer trained with next-token prediction**
  - Pretrain on large text, then fine-tune for tasks.
  - Unifies many NLP tasks as "predict the next token."
  - GPT-2 (2019)
    - **Scale + data:** much larger model and a broad web-scale dataset.
    - **Behavior shift:** strong **zero-shot** and **few-shot** generalization without task-specific training.
    - **Coherent long-form generation:** noticeably better fluency, consistency, and prompt-following.
    - This made it plausible that "just next-token prediction at scale" yields general capabilities.

- **The LLM revolution (2020–present):**
  - **Scaling laws:** larger models, more data, and more compute keep improving performance.
  - **Instruction tuning and reinforcement learning:** models become much more usable as assistants.
  - **Ecosystem:** open-source LLMs, tool use, retrieval augmentation, and multimodal LLMs/VLMs.

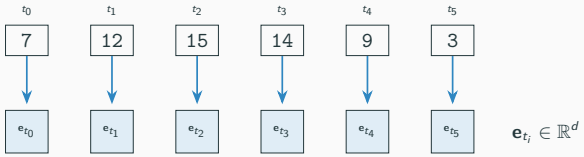
The Transformer Architecture

Step 1: Token Embeddings

Every token is an integer from a vocabulary $\{0, 1, \ldots, V-1\}$. A neural network cannot operate on integers directly — it needs **vectors**.

Token embedding: a learnable lookup table $\mathbf{W}_E \in \mathbb{R}^{V \times d}$.

$$t_i \in \{0, \ldots, 31\} \xrightarrow{\mathbf{W}_E} \mathbf{e}_{t_i} \in \mathbb{R}^d$$



- Each token gets its own d-dimensional vector (learned during training)
- Similar tokens can end up with similar embeddings
- In our setup: $V = 32$ tokens, $d = 64$ embedding dimensions (same dimensions as in the homework)

Step 2: Position Embeddings — Why Order Matters

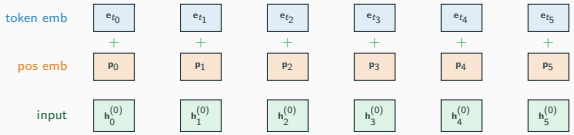
Token embeddings alone lose all **positional information**:

$$\mathbf{e}_{t_0}, \mathbf{e}_{t_1}, \ldots, \mathbf{e}_{t_5} \quad \text{— these are just a } \textit{set} \text{ of vectors!}$$

But for signal prediction, *where* a token appears is critical (e.g., "25 steps ago").

Solution: add a separate **position embedding** $\mathbf{p}_i \in \mathbb{R}^d$ to each position:

$$\mathbf{h}_i^{(0)} = \underbrace{\mathbf{e}_{t_i}}_{\text{what token}} + \underbrace{\mathbf{p}_i}_{\text{which position}}$$



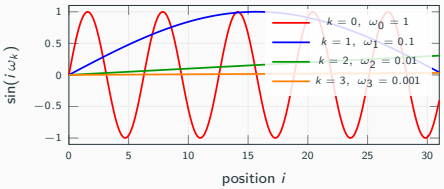
- $\mathbf{p}_i$ is a fixed or learnable vector for each position $i \in \{0, \ldots, K-1\}$
- The model learns relative-position structure through these vectors
- In our setup: $K = 64$ positions, each $\mathbf{p}_i \in \mathbb{R}^{64}$

Sinusoidal Position Embeddings (Vaswani et al., 2017)

Each position i is encoded using sinusoids at log-spaced frequencies.

Definition. For position i and dimension pair index k:

$$\omega_k = 10000^{-2k/d}, \qquad \mathbf{p}_i[2k] = \sin(i\,\omega_k), \qquad \mathbf{p}_i[2k+1] = \cos(i\,\omega_k).$$



- A sinusoid $\sin(i\,\omega_k)$ completes one cycle when its phase increases by 2π, so the wavelength (in tokens) is
$$\lambda_k = \frac{2\pi}{\omega_k}.$$
- With $\omega_k = 10000^{-2k/d}$ for $k = 0, 1, \ldots, d/2-1$, the wavelengths are $\lambda_k = 2\pi \cdot 10000^{2k/d}$.
- Range: $\omega_{\max} = 1$ at $k = 0$ gives $\lambda_{\min} = 2\pi \approx 6$ tokens. At $k = d/2-1$, $\omega_{\min} = 10000^{-1+2/d}$ gives $\lambda_{\max} = 2\pi \cdot 10000^{1-2/d} \approx 2\pi \cdot 10000$.
- High ω_k (small λ_k) resolves local positions; low ω_k (large λ_k) encodes long-range position.

- The log-spaced frequencies give each position a unique multi-resolution Fourier feature

Why Sin and Cos for Positions: Relative Shifts

- **Encode position on a unit circle (one frequency).**
  - For frequency ω and position i, define

$$\mathbf{p}_\omega(i) = \begin{bmatrix} \sin(\omega i) \\ \cos(\omega i) \end{bmatrix}.$$

- **Key property: shifting by k is a rotation that depends only on k.**
  - Use the trigonometric identities

$$\sin(\omega(i+k)) = \sin(\omega i)\cos(\omega k) + \cos(\omega i)\sin(\omega k),$$

$$\cos(\omega(i+k)) = \cos(\omega i)\cos(\omega k) - \sin(\omega i)\sin(\omega k).$$

  - Stack them into a 2D vector:

$$\mathbf{p}_\omega(i+k) = \begin{bmatrix} \cos(\omega k) & \sin(\omega k) \\ -\sin(\omega k) & \cos(\omega k) \end{bmatrix} \mathbf{p}_\omega(i).$$

  - So, for each frequency plane, a relative offset k corresponds to a fixed 2D rotation.

Why Sin and Cos for Positions: Dot Products Encode Offsets

- **Dot product depends only on the relative offset.**
  - For one frequency ω and positions i and j:

$$\mathbf{p}_\omega(i)^T \mathbf{p}_\omega(j) = \sin(\omega i)\sin(\omega j) + \cos(\omega i)\cos(\omega j).$$

  - Apply the cosine difference identity:

$$\sin(\omega i)\sin(\omega j) + \cos(\omega i)\cos(\omega j) = \cos(\omega(i-j)).$$

  - Therefore

$$\mathbf{p}_\omega(i)^T \mathbf{p}_\omega(j) = \cos(\omega(i-j)).$$

- **Multiple frequencies give a multi-scale signature of distance.**
  - Concatenate frequency planes $\omega_0, \omega_1, \ldots$ into a d-dimensional vector $\mathbf{p}_i$.
  - The total dot product becomes a sum of terms like $\cos(\omega_k(i-j))$.
  - High ω_k captures small offsets; low ω_k captures large offsets.

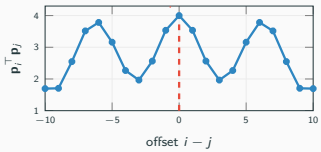
Sinusoidal Embeddings: Dot Products Select Positions

The position embedding at position i is a stack of $(\sin, \cos)$ pairs. The dot product between two positions has a beautiful closed form:

$$\mathbf{p}_i^\top \mathbf{p}_j \;=\; \sum_{k=0}^{d/2-1} \big[\sin(i\omega_k)\sin(j\omega_k) + \cos(i\omega_k)\cos(j\omega_k)\big] \;=\; \sum_{k=0}^{d/2-1} \cos\big((i-j)\,\omega_k\big)$$

It depends **only on the offset** $i-j$, not on absolute positions.

Dot product kernel (d=8):



Role of attention weights: shift the peak to offset τ.
If $\mathbf{W}_K$ rotates each $(\sin, \cos)$ pair by angle $\tau\omega_k$:

$$\mathbf{q}_i^\top \mathbf{k}_j = \sum_k \cos\big((i-j-\tau)\,\omega_k\big)$$

Now the dot product peaks at $i-j=\tau$.

Shifted kernel (τ=3):

This is exactly how attention will retrieve a token at a specific offset — the query–key dot product acts as a position-selective filter.

Self-Attention: The Core Idea

Key Insight
Each position **queries** all previous positions and retrieves a weighted combination of their **values**, based on how well their **keys** match the query.

For each position i, compute three vectors from the hidden state $\mathbf{h}_i$:

$$\mathbf{q}_i = \mathbf{W}_Q \mathbf{h}_i \quad \text{(query: "what am I looking for?")}$$
$$\mathbf{k}_i = \mathbf{W}_K \mathbf{h}_i \quad \text{(key: "what do I contain?")}$$
$$\mathbf{v}_i = \mathbf{W}_V \mathbf{h}_i \quad \text{(value: "what should I output?")}$$

Attention weights (how much position i attends to position j):

$$\alpha_{ij} = \text{softmax}_j\left(\frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d_k}}\right), \qquad j \le i \text{ (causal mask)}$$

Softmax definition (over index j):

$$\text{softmax}_j(s_j) = \frac{e^{s_j}}{\sum_{j'} e^{s_{j'}}}.$$

Output at position i:

$$\text{out}_i = \sum_{j \le i} \alpha_{ij} \, \mathbf{v}_j$$

Peaked Attention: Retrieving a Specific Position

Goal: Position i should attend strongly to position $j = i - \tau$ (e.g., $\tau = 3$).

Trick: Use position embeddings so that $\mathbf{q}_i^\top \mathbf{k}_j$ peaks at $j = i - \tau$.

Construction ($d = 2$, $\tau = 3$): Set sinusoidal position embeddings

$$\mathbf{p}_i = \begin{pmatrix} \cos(i\omega) \\ \sin(i\omega) \end{pmatrix}, \quad \mathbf{W}_Q = \mathbf{I}, \quad \mathbf{W}_K = \underbrace{\begin{pmatrix} \cos(3\omega) & \sin(3\omega) \\ -\sin(3\omega) & \cos(3\omega) \end{pmatrix}}_{\text{rotation matrix by } 3\omega}$$

Then:

$$\mathbf{q}_i^\top \mathbf{k}_j = \cos\big((i - j - 3)\omega\big) \quad \begin{cases} = 1 & \text{if } j = i - 3 \\ < 1 & \text{otherwise} \end{cases}$$

After softmax, attention concentrates on position $i - 3$:

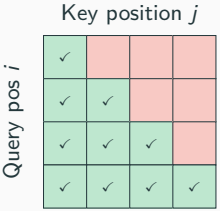
$$\boldsymbol{\alpha} \approx [\,0,\ 0,\ \mathbf{0.95},\ 0.02,\ 0.02,\ 0.01\,] \qquad \text{(peaked at offset } -3)$$

$\Rightarrow$ One head can implement a fixed-offset lookup.

The Causal Mask: No Peeking at the Future

The attention score matrix *before* softmax:

$$\mathbf{S} = \frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}} = \begin{pmatrix} s_{00} & -\infty & -\infty & -\infty \\ s_{10} & s_{11} & -\infty & -\infty \\ s_{20} & s_{21} & s_{22} & -\infty \\ s_{30} & s_{31} & s_{32} & s_{33} \end{pmatrix}$$



$-\infty$ entries become 0 after softmax.

Why causal? We're doing *next-token prediction*:

- Position i can only use information from positions $0, 1, \ldots, i$
- This is autoregressive: each prediction conditions only on the past
- Same mask used in all autoregressive models (GPT series and others) ...

Multi-Head Attention: Parallel Retrieval

Instead of one set of $(\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V)$, use H heads in parallel:

$$\text{head}_h = \text{Attn}(\mathbf{W}_Q^{(h)}\mathbf{h},\ \mathbf{W}_K^{(h)}\mathbf{h},\ \mathbf{W}_V^{(h)}\mathbf{h}) \qquad h = 1, \ldots, H$$

$$\text{MultiHead}(\mathbf{h}) = \text{concat}(\text{head}_1, \ldots, \text{head}_H) \cdot \mathbf{W}_O$$

Why multiple heads?

- Head 1: "attend to position $n - 25$" (fixed delay)
- Head 2: "attend to the nearest key token" (content match)
- Head 3: "copy from position $n - 1$" (local prediction)
- Head 4: "attend to a specific token" (symbol/value matching)

In our homework: $d = 64$, $H = 4$ heads, $d_k = 16$ per head.

Each head is a different retrieval program running in parallel.

Layer Normalization (LN): Definition + Tiny Example

LayerNorm normalizes *features within a token* (across the d channels), then applies a learned scale and bias.

Definition for a token vector $x \in \mathbb{R}^d$:

$$\mu = \frac{1}{d}\sum_{k=1}^{d} x_k, \quad \sigma^2 = \frac{1}{d}\sum_{k=1}^{d}(x_k - \mu)^2, \quad \mathrm{LN}(x) = \gamma \odot \frac{x-\mu}{\sqrt{\sigma^2+\varepsilon}} + \beta,$$

where $\gamma, \beta \in \mathbb{R}^d$ are learned and ε is a small constant.

Tiny numeric example (with $d = 4$):

$$x = [2,\ 0,\ 0,\ 2], \quad \mu = 1, \quad \sigma^2 = 1.$$

If $\gamma = \mathbf{1}$ and $\beta = \mathbf{0}$:

$$\mathrm{LN}(x) = \frac{x-\mu}{\sqrt{\sigma^2+\varepsilon}} \approx [1,\ -1,\ -1,\ 1].$$

Why LN helps: it keeps activations in a stable range so attention/MLPs can train reliably across many layers.

Feed-Forward MLP: Token-Wise Computation

- attention mixes information *across tokens*; the multi layer perceptron (MLP) computes *within each token*
- input to the MLP is a token vector $h_i \in \mathbb{R}^d$ (one row of the hidden-state matrix)
- standard 2-layer MLP (position-wise feed-forward network):

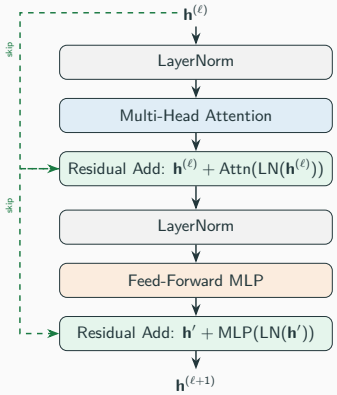
$$\mathrm{MLP}(h_i) = W_2\,\phi(W_1 h_i + b_1) + b_2,$$

  where $W_1 \in \mathbb{R}^{d_{\mathrm{ff}} \times d}$ and $W_2 \in \mathbb{R}^{d \times d_{\mathrm{ff}}}$, and ϕ is a nonlinearity (e.g., ReLU/GELU)
- applied independently to every token:

$$\mathrm{MLP}(H) = \begin{bmatrix} \mathrm{MLP}(h_1)^T \\ \vdots \\ \mathrm{MLP}(h_n)^T \end{bmatrix}$$

- **attention = communication (where to look), MLP = local compute (what to do)**
- examples of what the MLP can do *after* attention selects relevant tokens:
  - **feature construction**: combine retrieved features into new ones (e.g., AND/OR)
  - **gating**: amplify/suppress channels based on context (content-dependent filtering)
  - **arithmetic / comparisons**: compute functions of retrieved values (e.g., add, subtract)
  - **nonlinear denoising**: map a noisy data into a cleaner representation (e.g. soft-threshold)

The Transformer Block: Attention + MLP



- **Attention**: retrieves information from other positions (where to look)
- **MLP**: processes the retrieved information (what to compute)
- **Residual adds**: preserve information while enabling incremental updates

Example: MLP Stores a Fact

- **Task (next-token prediction).**

  $$[\texttt{The capital of France is}]$$

  Goal: predict the next token `Paris`.
- **Attention: select and aggregate the subject token.** Let i be the final position (after `is`). Self-attention computes

  $$q_i = h_i W_Q, \qquad k_j = h_j W_K, \qquad v_j = h_j W_V,$$

  $$a_{ij} = \mathrm{softmax}_j\left(\frac{1}{\sqrt{d_k}} q_i^T k_j\right), \qquad z_i = \sum_j a_{ij} v_j.$$

  Training can make the weights concentrate on the position j^* where the token is `France`:

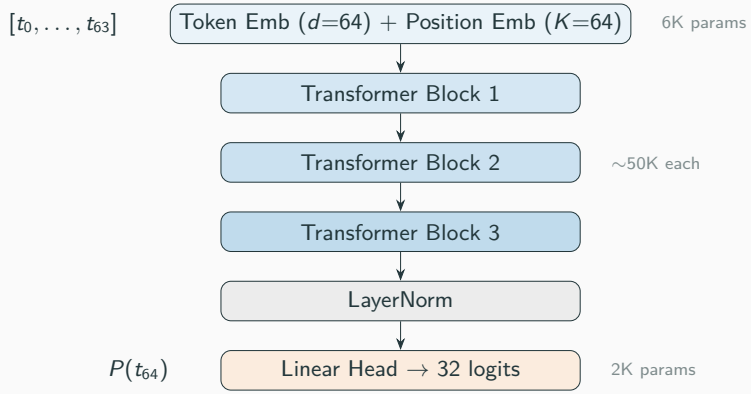
  $$a_{ij^*} \approx 1, \qquad a_{ij} \approx 0 \text{ for } j \neq j^*, \qquad z_i \approx v_{j^*}.$$

  So z_i becomes a transformed embedding of `France` (as seen from the query position).
- **MLP: map the retrieved subject to the correct object.** A token-wise MLP applies the same nonlinear map at every position:

  $$u_i = W_2\, \phi(W_1 z_i + b_1) + b_2.$$

- Attention: **retrieves the relevant subject** (`France`) into the prediction position.
- MLP and output head: **store the association** `France` $\rightarrow$ `Paris` as parametric memory.

MiniGPT Architecture from the Homework



Total: 158,272 parameters (L=3 layers, d=64, H=4 heads)

Only the **last position** $\mathbf{h}_{63}^{(L)}$ is used for prediction (next-token).

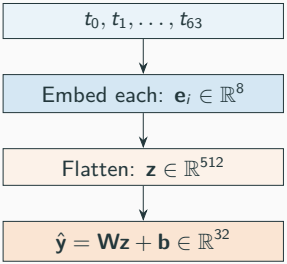
The Knobs: What You Can Change in a Transformer

Transformers have a few **architectural parameters** that directly control what sequence tasks are solvable.

Knob	Symbol	What it buys you
Context length	K	How far back you can possibly look. If the dependency is at offset τ, you need $K > \tau$.
Depth (layers)	L	How many sequential reasoning steps you can compose (retrieve $\rightarrow$ transform $\rightarrow$ retrieve again).
Heads	H	How many different retrieval patterns you can run in parallel.
Width	d	Feature capacity per token: sharper attention, richer representations.
MLP expansion	d_{ff}	Compute power for arithmetic / piecewise functions after retrieval.

Other Baseline Models for Prediction: Linear & MLP

Linear Predictor: A Flat View of Context

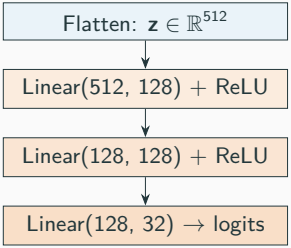


Key limitation: The model sees context as a *fixed flat vector*.

- Position information only through which slice of the 512-dim vector
- Cannot *dynamically select* which position to read from
- If the relevant past position changes (variable delay), the model is stuck

Good for: periodic signals, fixed patterns. Bad for: variable lookups.

MLP Predictor: Nonlinear but Still Flat



Advantage over Linear: can learn nonlinear functions.

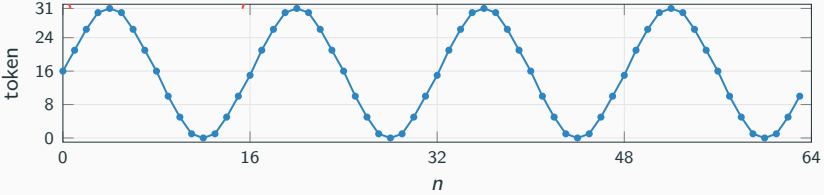
Still limited:

- Input is a flattened vector — no per-position processing
- Cannot route information based on content
- The same weights apply regardless of which key token appeared

Application: Signal Prediction

Signal 1: Sine Wave

$$x[n] = \sin\left(\frac{2\pi n}{16}\right), \qquad \text{quantized to } Q = 32 \text{ levels}$$



Only 9 unique tokens. **Perfectly periodic**: $\text{token}[n] = \text{token}[n + 16]$.

Result: All models = 100% accuracy. If any fails, you have a bug!

Signal 1: Concrete Next-Token Examples + Parameter Choices

Because the signal is exactly periodic with $P=16$, next-token prediction can be solved by copying the token from 16 steps ago.

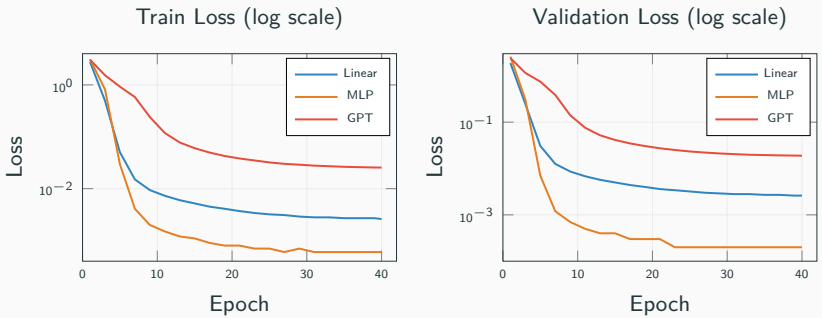
Example windows (each row shows the end of a length-K window):

Context suffix (last 24 tokens) $\rightarrow$ target
$[21, 26, 30, 31, 30, 26, 21, 16, 10, 5, 1, 0, 1, 5, 10, 16, 21, 26, 30, 31, 30, 26, 21, 16] \rightarrow 10$
$[26, 30, 31, 30, 26, 21, 16, 10, 5, 1, 0, 1, 5, 10, 16, 21, 26, 30, 31, 30, 26, 21, 16, 10] \rightarrow 5$

Minimal architecture that works:

- Context length: $K \geq P$ (need one full period; $K=16$ works)
- Depth: $L=1$ is enough (no multi-step reasoning)
- Heads: $H=1$ is enough (single periodic copy pattern)
- Baselines work too: linear/MLP can learn a fixed position-to-output mapping

Signal 1: Training Curves



- All three models converge rapidly — the periodic pattern is easy
- MLP converges fastest (lowest loss by epoch 5) due to its nonlinear capacity
- GPT converges slowest — its 158K parameters are overkill for this trivial task
- All reach 100% token-level accuracy despite different loss values

Background: The Mackey-Glass Equation

The Mackey-Glass equation models **physiological control systems** — originally the regulation of white blood cell production (hematopoiesis):

$$\frac{dx}{dt} = \frac{\beta\, x_\tau}{1 + x_\tau^{10}} - \gamma\, x(t), \qquad x_\tau \equiv x(t-\tau)$$

- **Delayed negative feedback**: the body adjusts production based on levels from τ time units ago
- **Sigmoidal saturation**: $\frac{x}{1+x^{10}}$ prevents unbounded growth
- For large τ: dynamics become **chaotic** (sensitive dependence on initial conditions)
- We use $\tau = 25$ (chaotic regime), discretized:

$$x[n{+}1] = x[n] + \frac{0.3 \cdot x[n{-}25]}{1 + x[n{-}25]^{10}} - 0.1 \cdot x[n]$$

The key challenge: predicting $x[n{+}1]$ requires knowing $x[n{-}25]$ — the value **exactly 25 steps back**. A model that cannot look back 25 steps will fail.

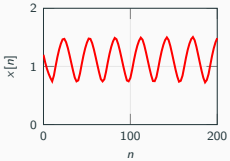
Mackey-Glass: Small Delay, Edge, and Chaotic Regimes

- Same discrete update rule, different delay τ:

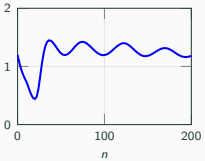
$$x[n+1] = x[n] + \frac{0.3 \cdot x[n-\tau]}{1 + x[n-\tau]^{10}} - 0.1 \cdot x[n].$$

- As τ increases: dynamics go from smooth to irregular to chaotic.

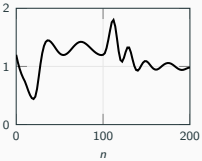
$\tau = 5$ (smooth, nonchaotic)



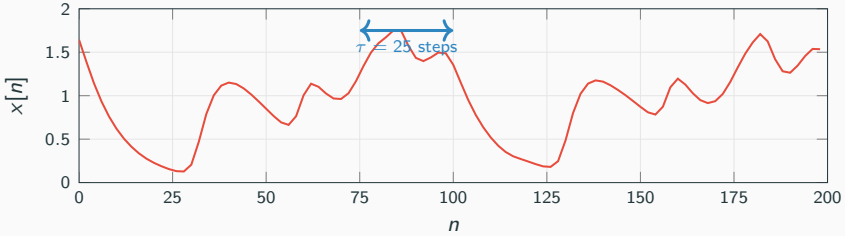
$\tau = 17$ (near the edge)



$\tau = 25$ (chaotic)



Signal 2: Mackey-Glass — Chaotic Dynamics



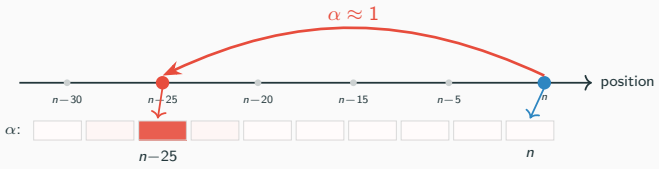
- Aperiodic oscillations with amplitude variations — **chaotic**
- To predict $x[101]$, the model *must* access $x[76]$ (25 steps back)
- Quantized to $Q = 32$ tokens for the prediction task

Signal 2: How Attention Solves Mackey-Glass

Construction: One attention head dedicated to delay-25 lookup.

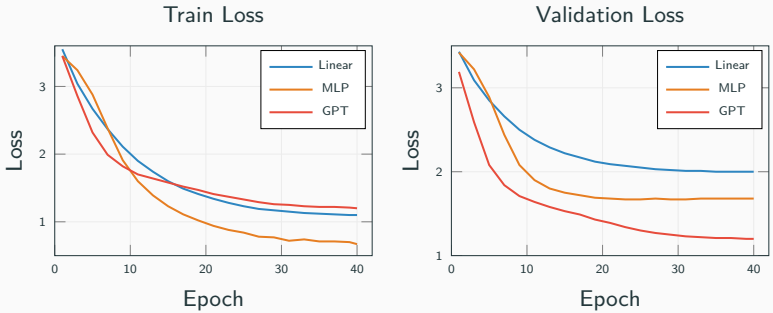
Set $\mathbf{W}_Q = \mathbf{I}$, $\mathbf{W}_K = \mathbf{R}_{25\omega}$ (rotation by 25ω):

$$\mathbf{q}_i^\top \mathbf{k}_j = \cos\big((i-j-25)\omega\big) \xrightarrow{\text{softmax}} \alpha_{ij} \approx \begin{cases} 1 & j = i-25 \\ 0 & \text{otherwise} \end{cases}$$



- **Attention** retrieves $x[n-25]$ via peaked attention weights
- **MLP** computes the nonlinear recurrence term after retrieval
- An MLP baseline sees a flat 512-dim vector — must discover position $n-25$ implicitly

Signal 2: Training Curves



- MLP achieves lowest train loss but validation plateaus (overfitting)
- GPT achieves the lowest validation loss (attention enables generalization)
- Linear underfits (cannot learn the nonlinear recurrence)

Signal 2: Parameter Choices That Make/Break the Task

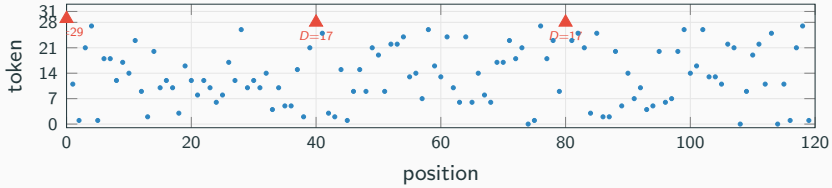
This task is almost a diagnostic test for context length.

Choice	What it implies for Mackey-Glass ($\tau = 25$)
$K \leq 25$	Impossible: the window does not contain $x[n-25]$.
$K \approx 32$	Barely enough: can see the needed point but with little extra context.
$K = 64$	Comfortable: contains $x[n-25]$, $x[n-1]$, etc.
$H \geq 1$	At least one head can specialize to offset-25 retrieval.
$L = 1$	Often sufficient: retrieve and compute in one block.
Large d_{ff}	Helps approximate the nonlinearity after retrieval.

Interpretation: attention is the "read" operation; the MLP is the "nonlinear compute" operation.

Signal 3: Switching Delay — Variable-Offset Retrieval

Key tokens $\{28, 29, 30, 31\}$ select delay $D \in \{17, 29, 41, 53\}$. Value tokens follow: $x[n] = (3 \cdot x[n - D] + 7) \bmod 28$.



▲ = key tokens setting the active delay. To predict each ●, the model must: (1) find the active key → determine D, then (2) look back D steps and compute $(3x + 7) \bmod 28$.

Signal 3: Worked Token Example (Key $\Rightarrow$ Delay $\Rightarrow$ Retrieval)

Think of the sequence as long stretches of values (0–27), occasionally interrupted by a key token (28–31) that changes the delay rule.

Illustrative example:

$$[\,29,\ 11,\ 1,\ 21,\ 27,\ 1,\ \ldots\,] \quad \Rightarrow \quad D = 29.$$

Later at time n (a value position), the rule is:

$$x[n] = (3 \cdot x[n-29] + 7) \bmod 28.$$

If $x[n-29] = 9$, then:

$$x[n] = (3 \cdot 9 + 7) \bmod 28 = 34 \bmod 28 = 6.$$

What the model must do:

1. Read the most recent key token to infer D.
2. Retrieve the value at offset D (content-conditioned offset lookup).
3. Apply the arithmetic program (MLP).

Signal 3: Multi-Head Solution

Head 1: "Find the most recent key token"

- Key tokens $\{28\text{–}31\}$ produce distinctive key vectors (via token embedding)
- Value vector encodes which delay is active ($D = 17, 29, 41, 53$)
- $\Rightarrow$ After attention, position n knows which delay is active

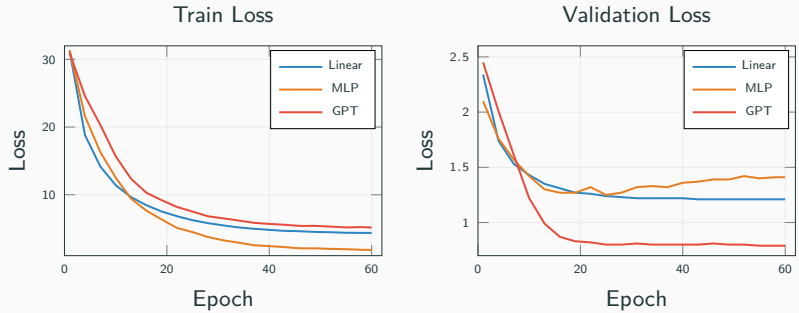
Heads 2–4: "Look back D positions" (each head specializes on one delay)

- Head 2: peaked at offset -17, Head 3: offset -29, Head 4: offset -41
- Each uses the rotation-matrix construction from Signal 2

MLP layer: Computes $(3x + 7) \bmod 28$ from the retrieved value.

Why MLP fails: A flat vector cannot dynamically select *which* of the 64 positions to read based on a key token's value.

Signal 3: Training Curves



- GPT validation loss drops to **0.79** — far below baselines
- MLP validation rises after epoch 15 (overfitting to background tokens)
- High GPT train loss reflects the 10× weighting on retrieval steps

Attention as Associative Memory

Key Analogy
Self-attention = a differentiable, content-addressable memory.

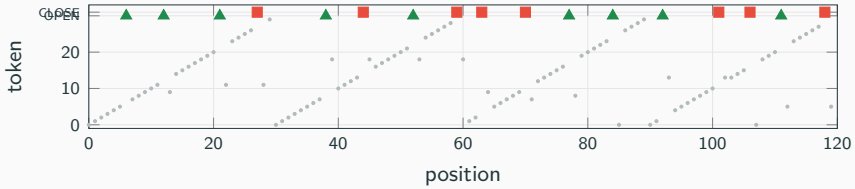
Memory Operation	Attention	Signal 3 Example
Address (query)	$\mathbf{q}_i = \mathbf{W}_Q \mathbf{h}_i$	"I need the value at delay D"
Memory slots	$\mathbf{k}_j = \mathbf{W}_K \mathbf{h}_j$	Key tokens produce matching keys
Stored values	$\mathbf{v}_j = \mathbf{W}_V \mathbf{h}_j$	Token content at each position
Read operation	$\sum_j \alpha_{ij} \mathbf{v}_j$	Weighted retrieval

Contrast with a look-up table:

- Attention: soft match, weighted combination (differentiable)
- This makes it trainable by gradient descent

Signal 4: Bracket Matching — Why Depth Matters

OPEN (token 30) pushes a tag; CLOSE (token 31) expects the matching tag:



▲ OPEN ■ CLOSE • background counting. At each CLOSE, predict the tag of the **matching unmatched** OPEN — requires stack tracking!

Signal 4: A Tiny Worked Example (Stack Tracking)

Consider this short sequence (spaces are just for readability):

30 7 30 2 31 _ 31 _

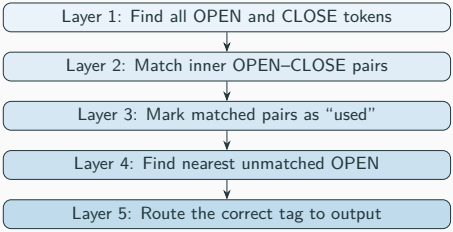
Interpretation:

- See OPEN 7 $\Rightarrow$ stack $= [7]$
- See OPEN 2 $\Rightarrow$ stack $= [7, 2]$
- See CLOSE $\Rightarrow$ must output **2** (pop) $\Rightarrow$ stack $= [7]$
- Next CLOSE $\Rightarrow$ must output **7**

Why depth matters: "nearest unmatched OPEN" is a stateful property that changes after each CLOSE.

Signal 4: Multi-Layer Stack Simulation

Why increasing number of layers suddenly improves performance (depth threshold effect):

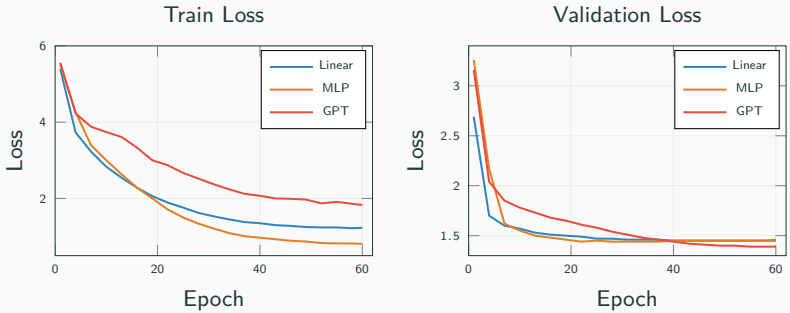


Depth sweep (conditional accuracy on retrieval steps):

	$L = 1$	$L = 3$	$L = 5$
Bracket Matching	10.6%	9.4%	24.7%

Sharp threshold: $L \leq 3$ is near random. At $L = 5$: enough layers to compose a multi-step stack simulation.

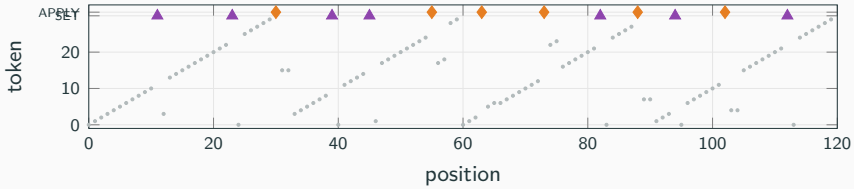
Signal 4: Training Curves



- GPT starts slow (high train loss) but achieves lowest validation loss
- Linear and MLP plateau early on validation (cannot solve stack matching)
- GPT keeps improving through all epochs (depth helps)

Signal 5: Conditional Routing — Read-Then-Compute

SET_OP (token 30) selects an operation; APPLY (token 31) tests it:



▲ SET_OP (sets operation 0–3) ♦ APPLY (tests with argument).

4 operations: identity, increment $(x+1)$ mod 30, double $(2x)$ mod 30, complement $(29-x)$ mod 30.

Signal 5: Tiny Worked Example (Read Then Compute)

Example program in tokens (spaces for readability):

$$30\ 2\ \cdots\ 31\ 18\ \_$$

Interpretation:

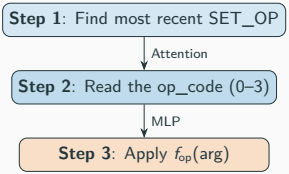
- SET_OP 2 sets operation to **double**
- APPLY with argument 18 must output $(2 \cdot 18) \bmod 30 = 36 \bmod 30 = \mathbf{6}$

If instead the active op were complement ($o = 3$):

$$(29 - 18) = \mathbf{11}.$$

So the model must retrieve the opcode from context before deciding the correct mapping.

Signal 5: The Transformer Pipeline



Query: "where is SET_OP?"
Key: SET_OP tokens have distinctive embeddings
$\Rightarrow$ attention peaks at SET_OP position

MLP implements a piecewise function:
if op=0: output arg | if op=1: output $(x+1) \bmod 30$
if op=2: output $(2x) \bmod 30$ | if op=3: output $(29-x) \bmod 30$

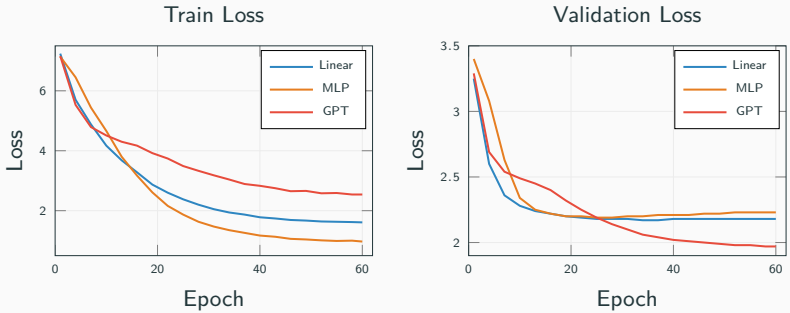
Same argument (18) gives different answers depending on active operation:

Operation	arg	answer	computation
identity	18	18	$f(18) = 18$
double	18	6	$(2 \times 18) \bmod 30 = 6$
complement	18	11	$(29 - 18) \bmod 30 = 11$

$\Rightarrow$ MLP cannot learn a fixed mapping. Must retrieve context first.

Result: GPT **27.9%** vs MLP 6.1% (**+21.8 pp**)

Signal 5: Training Curves

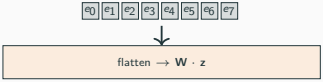


- GPT validation loss keeps decreasing (it learns the routing)
- MLP validation increases after about epoch 15 (overfitting to background)

Why MLP Fails on These Tasks

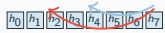
The Fundamental Limitation of Flat Models

MLP/Linear (flat input)



- All positions mixed into one vector
- Same weights for every input
- Cannot adapt "where to look"

Transformer (per-position + attention)

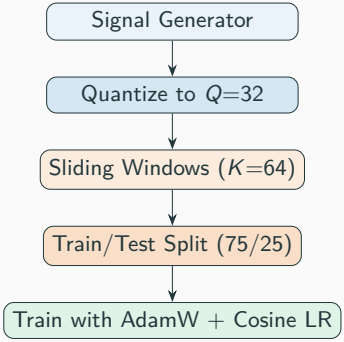


- Each position processed independently
- Attention selects relevant positions
- Input-dependent routing

Capability	MLP	Transformer
Fixed patterns (sine)	✓	✓
Fixed offset lookup	Partially	✓
Variable offset lookup	✗	✓
Content-based retrieval	✗	✓
Multi-step reasoning	✗	✓(multi-layer)

Training Pipeline

From Raw Signal to Training Data



Token-space signals skip this step
— tokens are generated directly

Loss Weighting (Events vs. Background)

For next-token prediction, the (weighted) cross-entropy loss is:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} w_i \cdot \left(-\log P_\theta(t_i \mid t_{<i}) \right).$$

In Signals 3–5, most tokens are **background** (easy) and a minority are **events** (hard):
Loss weighting for token-space signals (Signals 3–5):

$$w_i = \begin{cases} 10.0 & \text{if event\_mask}[i] = \text{True (retrieval step)} \\ 1.0 & \text{otherwise (background)} \end{cases}$$

Without this, gradient is dominated by easy background tokens.

- Background tokens can be predicted with shallow heuristics.
- Event tokens require retrieval (matching OPEN, last SET_OP, correct delay).

Upweighting events forces the optimizer to learn the attention patterns that actually solve the task.

GPT-2 in One Slide: What Changes vs. MiniGPT?

MiniGPT here is a toy model (V=32, K=64, L=3, d=64, H=4). GPT-2 keeps the same core block but scales up and uses a different tokenizer.

Same core ideas

- Causal self-attention (triangular mask)
- Learned token + positional embeddings
- Residual connections and LayerNorm
- Autoregressive training: predict next token

GPT-2 specifics

- BPE tokenizer (Byte Pair Encoding): $V \approx 50{,}257$
- Context length: $K = 1024$
- MLP expansion: $d_{\text{ff}} = 4d$ with GELU nonlinearity
- Weight tying: output projection shares weights with token embeddings

GPT-2 is the same recipe scaled until it generalizes across many sequence tasks.

GPT-2 Configurations (Classic Sizes)

GPT-2 comes in a few standard sizes (same architecture, different L, H, d).

Model	**Params**	**Layers** L	**Heads** H	**Width** d
GPT-2 Small	$\sim$124M	12	12	768
GPT-2 Medium	$\sim$355M	24	16	1024
GPT-2 Large	$\sim$774M	36	20	1280
GPT-2 XL	$\sim$1.5B	48	25	1600

Shared settings:

- Context length $K = 1024$; learned positional embedding $\mathbf{W}_P \in \mathbb{R}^{1024 \times d}$
- MLP hidden size $d_{\text{ff}} = 4d$ (e.g. 3072 for d=768)
- Per-head dimension $d_k = d/H$ (e.g. 64 for Small)

GPT-2 Block: Exact Computation (Pre-LN, GELU, Residual)

One GPT-2 transformer block takes $\mathbf{h}^{(\ell)} \in \mathbb{R}^{K \times d}$ and outputs $\mathbf{h}^{(\ell+1)}$:

$$\begin{aligned}
\tilde{\mathbf{h}} &= \mathrm{LN}\big(\mathbf{h}^{(\ell)}\big) \\
\mathbf{h}' &= \mathbf{h}^{(\ell)} + \mathrm{MHA}\big(\tilde{\mathbf{h}}\big) \\
\tilde{\mathbf{h}}' &= \mathrm{LN}\big(\mathbf{h}'\big) \\
\mathbf{h}^{(\ell+1)} &= \mathbf{h}' + \mathrm{MLP}\big(\tilde{\mathbf{h}}'\big)
\end{aligned}$$

where

$$\mathrm{MLP}(x) = \mathbf{W}_2\,\mathrm{GELU}(\mathbf{W}_1 x + b_1) + b_2, \quad \mathbf{W}_1 \in \mathbb{R}^{d \times 4d}, \ \ \mathbf{W}_2 \in \mathbb{R}^{4d \times d}.$$

Minimal definitions used here

- **MHA** = multi-head attention (the parallel heads slide earlier).
- **GELU** is a smooth nonlinearity; one common form is $\mathrm{GELU}(x) = x\Phi(x)$ where Φ is the standard normal CDF.

GPT-2 Embeddings, Output Head, and Weight Tying

GPT-2 turns a token sequence into logits over the vocabulary:

$$\mathbf{h}_i^{(0)} = \mathbf{W}_E[t_i] + \mathbf{W}_P[i] \quad \text{for } i = 0, \ldots, K-1$$

After L blocks and a final LayerNorm:

$$\text{logits}_i = \mathbf{h}_i^{(L)} \mathbf{W}_E^\top \quad \in \mathbb{R}^V$$

(weight tying: output projection uses the same $\mathbf{W}_E$ as input embeddings).

Tokenizer note: GPT-2 tokens are subword units (BPE), not full words. In the homework, our signals use a toy vocab ($V = 32$) where tokens already represent the discrete symbols of the task.

Rotary Position Embeddings (RoPE, Su et al., 2021)

Additive PE: form $h_i = x_i + p_i$ before attention. **RoPE:** rotate q_i, k_i after projection.

Where it acts. Given token embedding x_i,

$$q_i = x_i W_Q, \qquad k_i = x_i W_K, \qquad v_i = x_i W_V.$$

RoPE applies a position-dependent rotation (block-diagonal over 2D pairs):

$$\tilde{q}_i = R(i)\, q_i, \qquad \tilde{k}_i = R(i)\, k_i,$$

where for pair k with $\theta_{i,k} = i\,\omega_k$ and $\omega_k = 10000^{-2k/d}$,

$$R(i)\big|_{2k,\,2k+1} = \begin{pmatrix} \cos\theta_{i,k} & -\sin\theta_{i,k} \\ \sin\theta_{i,k} & \cos\theta_{i,k} \end{pmatrix}.$$

Key property (relative positions). Because rotations compose as $R(i)^T R(j) = R(j-i)$ (pairwise),

$$\tilde{q}_i^\top \tilde{k}_j = q_i^\top R(i)^T R(j)\, k_j = q_i^\top R(j-i)\, k_j,$$

so position enters the attention score via the **offset** $j - i$.

Numerical example (d=4, k=0, ω_0=1). Take $q = (1, 0)$ and $k = (1, 0)$. If i=5 and j=2 then

$$\tilde{q} = (\cos 5,\ \sin 5) = (0.284,\ -0.959), \qquad \tilde{k} = (\cos 2,\ \sin 2) = (-0.416,\ 0.909),$$

and

$$\tilde{q}^\top \tilde{k} = \cos(5-2) = \cos(3) = -0.990.$$

RoPE is widely used in modern decoder-only LLMs (e.g., Qwen, LLaMA, Gemma, Mistral...). Su et al., 2021

Results & Takeaways

Main Results: Where Attention Wins

Signal	Linear	MLP	GPT	Gap
Global accuracy (%)				
1. Sine	100.0	100.0	100.0	—
2. Mackey-Glass	37.5	44.1	**54.0**	+9.9
Conditional accuracy[3] *on retrieval steps (%)*				
3. Switching Delay	68.5	71.9	**78.7**	+6.8
4. Bracket Matching	5.9	7.1	**9.4**	+2.3
5. Conditional Routing	4.8	6.1	**27.9**	+21.8

Largest gap: Conditional Routing (+21.8 pp) — read then compute is native to transformers.

Smallest gap at L=3: Bracket Matching (+2.3 pp) — needs deeper models.

[3] accuracy restricted to "event" (retrieval) positions only.

Global vs. Conditional Accuracy

Note. In our toy signals, GPT's global accuracy can be lower than MLP's.

Signal	Linear	MLP	GPT
4. Bracket Matching (global)	**68.4%**	68.0%	60.5%
4. Bracket Matching (conditional)	5.9%	7.1%	24.7%
5. Conditional Routing (global)	**51.1%**	53.0%	44.6%
5. Conditional Routing (conditional)	4.8%	6.1%	27.9%

Why? Token-space signals have about 85% easy background tokens.

- MLP/Linear optimize for the easy majority $\rightarrow$ high global accuracy
- GPT allocates capacity to the hard retrieval task $\rightarrow$ lower global, higher conditional
- The $10\times$ loss weight on retrieval steps helps steer GPT's gradient

Conditional accuracy is the right metric for attention-requiring tasks.

Depth Sweep: More Layers, More Reasoning

Signal (cond. acc.)	$L = 1$	$L = 3$	$L = 5$
3. Switching Delay	76.4%	78.7%	**82.1%**
4. Bracket Matching	10.6%	9.4%	24.7%
5. Conditional Routing	23.8%	**27.9%**	26.5%

Key observations:

1. **Bracket Matching**: sharp depth threshold at L=5.
2. **Switching Delay**: steady improvement with depth.
3. **Conditional Routing**: saturates early (single-hop retrieval).

Rule of thumb: k-step reasoning often needs about k transformer layers.

Summary: When Does Attention Help?

✗	Fixed periodic patterns
✓	Fixed-offset delay lookups
✓	Variable-offset retrieval
✓	Content-addressable memory
✓	Read-then-compute pipelines
✓✓	Hierarchical / multi-step reasoning (needs depth)

In your homework: Run 5 signals, compare 3 models, sweep MiniGPT depth.

The script (`signal_gpt_v4.py`) runs in about 2 min on GPU or about 48 min on CPU.

Questions?