From Attention to Transformers

Evan Dramko

March 2025

Impact of Attention and Transformers

The "T" in ChatGPT

They are the driving force of LLMs, AlphaFold, (and almost everything else in neural networks)

Impact across fields:

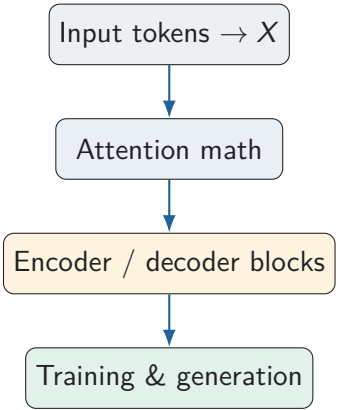
- Natural language processing
- Vision
- Protein structure prediction
- Materials modeling

Agenda and Roadmap

Core idea

Attention is the computational primitive; the Transformer is the architecture built around it.

- Start with a sequence of token embeddings.
- Build Q, K, and V from linear projections.
- Use attention to compute every-to-every interactions.
- Wrap that mechanism with residual paths, normalization, and MLPs.
- Add masking and caching to make decoding practical.



Notation and shapes

Single sequence view (used to explain attention)

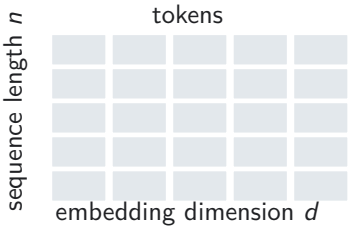
$$X \in \mathbb{R}^{n \times d_x}$$

- n: sequence length
- d_x: token embedding dimension

Batched Transformer view

$$X \in \mathbb{R}^{B \times n \times d_{model}}$$

- B: batch size
- same attention math, with an extra leading batch dimension



Throughout the deck, dimensions are the quickest way to see what a tensor means.

Step 1: build Q, K, and V

Start with token matrix

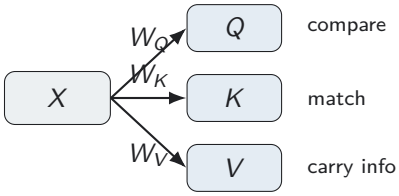
$$X \in \mathbb{R}^{n \times d_{model}}$$

Project it three different ways:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V$$

$$Q, K \in \mathbb{R}^{n \times d_k}, \qquad V \in \mathbb{R}^{n \times d_v}$$

- same n: still one row per token
- new feature spaces: each projection has a different role
- attention uses Q and K to decide weights, then applies them to V



Step 2: raw attention scores are $QK^\top$

Before softmax, attention forms the raw score matrix

$$S = QK^\top$$

Shape check:

$$Q \in \mathbb{R}^{n \times d_k}, \qquad K^\top \in \mathbb{R}^{d_k \times n} \qquad \Longrightarrow \qquad S \in \mathbb{R}^{n \times n}$$

Each entry is a dot product:

$$S_{ij} = q_i \cdot k_j$$

- row i: scores produced by query token i
- column j: how strongly key token j matches
- these are **logits**, not probabilities yet

raw attention logits $S = QK^\top$

keys

queries

$$\begin{array}{c|cccc} & & & & \\ \hline & q_1 k_1^\top & q_1 k_2^\top & \cdots & q_1 k_n^\top \\ & q_2 k_1^\top & q_2 k_2^\top & \cdots & q_2 k_n^\top \\ & \vdots & \vdots & \ddots & \vdots \\ & q_n k_1^\top & q_n k_2^\top & \cdots & q_n k_n^\top \end{array}$$

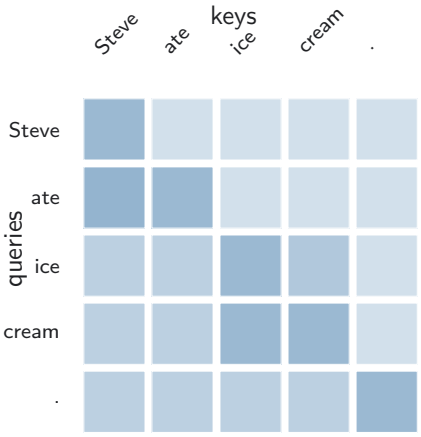
Why the raw score matrix is $n \times n$

Interpretation

Each row corresponds to one query token.
Each column corresponds to one key token.
The entry S_{ij} tells us how strongly token i matches token j **before normalization**.

$$S = QK^\top \in \mathbb{R}^{n \times n}$$

- every token compares against every token
- one row = one token asking about all other tokens
- values need not be symmetric



Step 3: scale the logits, then apply row-wise softmax

Raw attention scores:

$$S = QK^\top$$

Scaled scores:

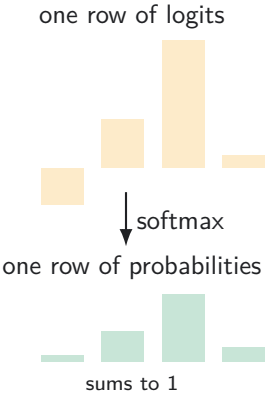
$$\widetilde{S} = \frac{QK^\top}{\sqrt{d_k}}$$

Normalized attention weights:

$$A = \mathrm{softmax}\left(\widetilde{S}\right)$$

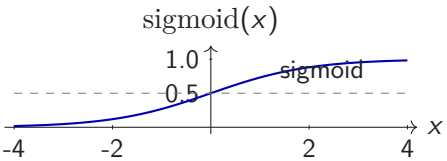
Softmax is applied row-wise, so each row becomes a distribution:

$$A_{ij} \geq 0, \qquad \sum_j A_{ij} = 1$$



Sigmoid vs. Softmax

$$\sigma(x) = \frac{1}{1 + e^{-x}} (sigmoid) \qquad \sigma(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}} (softmax)$$



- **Sigmoid** acts on a single scalar
- **Softmax** acts on a vector quantity
- Sigmoid maps one value into the range $(0, 1)$
- Softmax turns a vector into a probability distribution (sum to 1)
- In attention, **softmax** is used so each row of attention weights sums to 1

Why divide by $\sqrt{d_k}$, and why softmax after that?

What goes wrong without scaling

As d_k grows, raw dot products tend to grow in variance. Large logits make softmax very sharp, so one token dominates and the rest get almost zero weight.

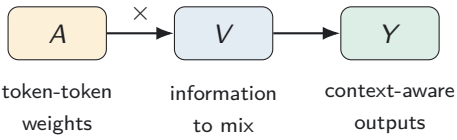
$$S = QK^\top, \qquad \widetilde{S} = \frac{S}{\sqrt{d_k}}, \qquad A = \mathrm{softmax}(\widetilde{S})$$

- $\sqrt{d_k}$ keeps logits in a reasonable range
- softmax then converts each row into a probability distribution
- this preserves graded, every-to-every interactions
- avoids attention becoming one-hot

Step 4: use attention weights to mix the values

Once we have attention weights A, we apply them to the value matrix:

$$Y = AV$$



For token i:

$$y_i = \sum_{j=1}^{n} A_{ij} v_j$$

- row i of A says how much token i should use each value vector
- same number of tokens in, same number of tokens out

$$A \in \mathbb{R}^{n \times n}, \qquad V \in \mathbb{R}^{n \times d_v} \qquad \Longrightarrow \qquad Y \in \mathbb{R}^{n \times d_v}$$

Attention pipeline: logits → weights → outputs

The whole computation can be read in three stages:

$$Q, K, V = XW_Q, XW_K, XW_V$$

$$S = QK^\top$$

$$A = \text{softmax}\left(\frac{S}{\sqrt{d_k}}\right)$$

$$Y = AV$$

- S: raw compatibility logits
- A: normalized attention weights
- Y: mixed, context-aware token representations

Encoder block = attention + residual + norm + FFN

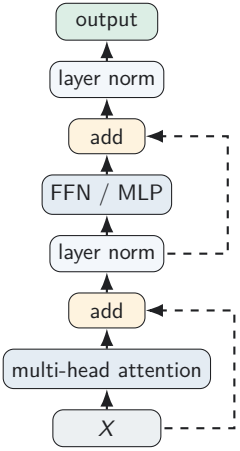
$$\tilde{X} = X + \mathrm{MHA}(X)$$

$$X' = \mathrm{LayerNorm}(\tilde{X})$$

$$\hat{X} = X' + \mathrm{FFN}(X')$$

$$X_{out} = \mathrm{LayerNorm}(\hat{X})$$

- residuals preserve information and help gradients flow
- layer norm stabilizes the feature scale per token
- FFN mixes information within each token after attention mixed information across tokens



LayerNorm and FFN: what happens inside a block

LayerNorm per token

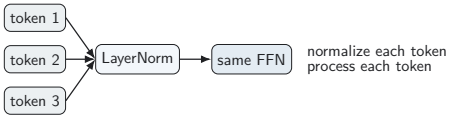
For batched data $X \in \mathbb{R}^{B \times n \times d_{model}}$, normalize each token independently across its feature dimension:

$$\text{LayerNorm}(x) = \gamma \odot \frac{x - \mu}{\sigma + \varepsilon} + \beta$$

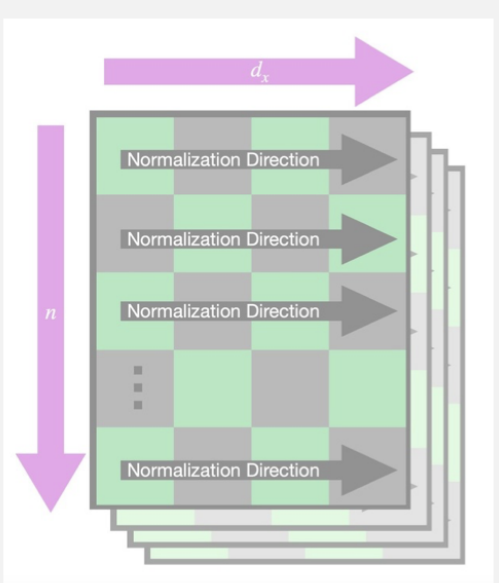
Feed-forward network per token

$$\text{FFN}(x) = W_2\, \sigma(W_1 x + b_1) + b_2$$

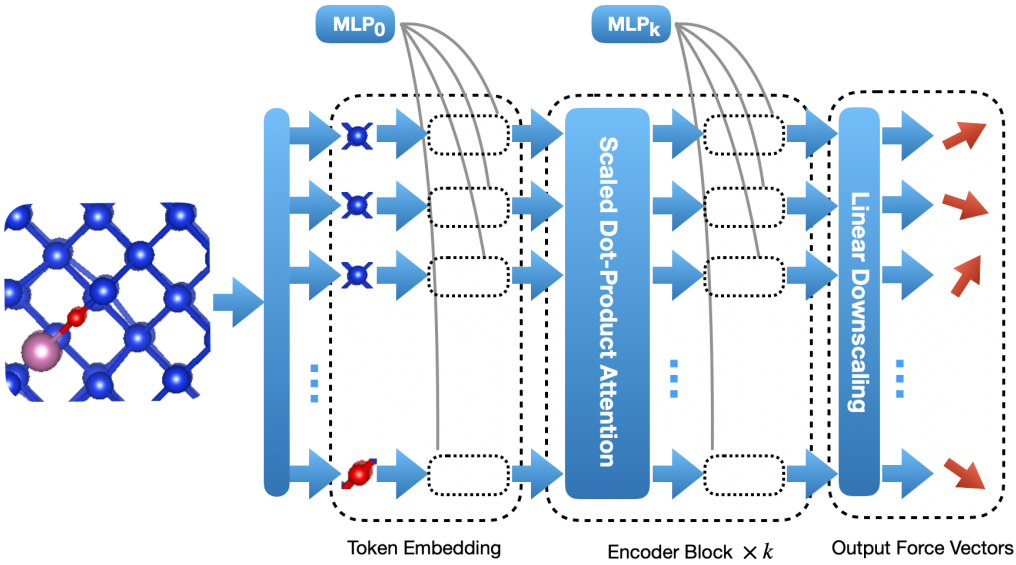
Same weights are reused at every token position.



LayerNorm


d_x
n
Normalization Direction
Normalization Direction
Normalization Direction
Normalization Direction

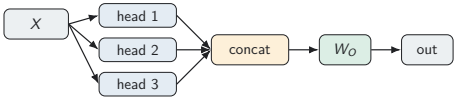
Transformer encoder



Multi-head attention: one mechanism, many subspaces

$$\text{head}_i = \text{softmax}\left(\frac{Q_i K_i^\top}{\sqrt{d_k}}\right) V_i \qquad i = 1, \ldots, h$$

$$\text{MHA}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W_O$$



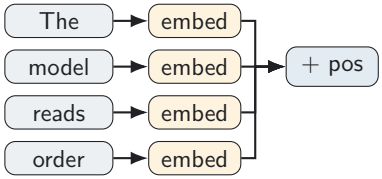
- split d_{model} across h heads, usually with $d_k = d_{model}/h$
- each head can specialize to a different interaction pattern
- concatenate the heads, then project back to d_{model}

Decoders and LLMs

Input stage: embeddings and positional information

- Transformers operate on sequences of continuous vectors.
- Native data is tokenized into discrete units, then embedded into $\mathbb{R}^{d_{model}}$.
- Positional encoding is added so the model can tell tokens apart by order.

$$X_0 = E(\text{tokens}) + P$$



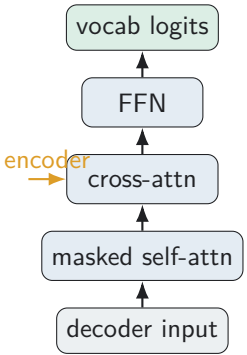
Why this matters

Pure attention is permutation-equivariant. Without positional information, the model sees a bag of tokens rather than an ordered sequence.

Decoder block: what changes

1. masked self-attention on decoder tokens
2. residual + layer norm
3. cross-attention to encoder outputs (if an encoder exists)
4. residual + layer norm
5. FFN, then projection to vocabulary logits

$$X \in \mathbb{R}^{B\times n\times d_{model}} \quad \rightarrow \quad XW_V \in \mathbb{R}^{B\times n\times |\mathcal{V}|}$$



Decoder-only models

Skip cross-attention entirely. Masked self-attention + FFN is enough for next-token prediction.

What the decoder is trained to do

Given a sequence:

`Steve went to the store`

The decoder is trained with a shifted input:

input: ⟨*sos*⟩ `Steve went to the store`

target: `Steve went to the store` ⟨*eos*⟩

What each position learns

- ⟨*sos*⟩ → Steve
- ⟨*sos*⟩ `Steve` → went
- ⟨*sos*⟩ `Steve went` → to
- ...

Key idea

The model always predicts the **next token given the full prefix**.

- not pairwise ("Steve → went")
- always **prefix → next token**

From decoder outputs to next-token probabilities

Decoder output:

$$X \in \mathbb{R}^{B \times n \times d_{model}}$$

Project to vocabulary:

$$Z = XW_{out} \in \mathbb{R}^{B \times n \times |\mathcal{V}|}$$

- each position produces logits over the vocabulary

Convert to probabilities:

$$P = \text{softmax}(Z)$$

We only use the **last position**: $P(x_t \mid x_{1:t-1})$

Interpretation

- logits = unnormalized scores
- softmax = probability distribution
- size = vocabulary size

Key detail

The distribution is conditioned on **all previous tokens**.

Autoregressive generation (inference loop)

Start with:

$$[\langle sos \rangle]$$

Loop:

1. Run decoder on current sequence
2. Take logits at **last position**
3. Compute probabilities with softmax
4. **Sample or argmax** next token
5. Append token to sequence

Repeat until:

- $\langle eos \rangle$ is produced
- or max length reached

Example

- $\langle sos \rangle \rightarrow$ Steve
- $\langle sos \rangle$ Steve $\rightarrow$ went
- $\langle sos \rangle$ Steve went $\rightarrow$ to

Key idea

Each prediction becomes part of the next input.

Training vs inference (why they differ)

Training (teacher forcing)

- full sequence given at once
- predictions made at all positions
- parallel computation

$$[\langle sos \rangle, x_1, x_2, x_3] \to [x_1, x_2, x_3, x_4]$$

Inference (generation)

- no ground truth future tokens
- must generate step-by-step
- sequential loop

$$\begin{aligned} [\langle sos \rangle] &\to x_1 \\ [\langle sos \rangle, x_1] &\to x_2 \\ [\langle sos \rangle, x_1, x_2] &\to x_3 \\ &\vdots \end{aligned}$$

Consequence

Errors can compound over time.

Cross-attention and causal masking

Cross-attention

Use different sources for queries vs. keys/values:

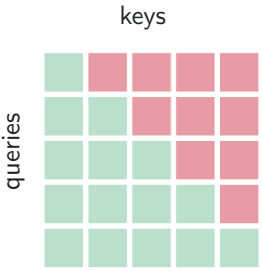
$$Q = X_{dec} W_Q, \qquad K = X_{enc} W_K, \qquad V = X_{enc} W_V$$

The decoder asks questions; the encoder provides the memory to read from.

Causal masking

Prevent token t from attending to future tokens $> t$ by adding a mask before softmax.

$$A = \text{softmax}\left(\frac{QK^\top + M}{\sqrt{d_k}} \right)$$



green: allowed
red: masked future

Autoregressive decoding with KV cache

At time step t, only the new token needs fresh projections:

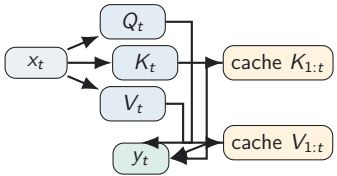
$$Q_t = x_t W_Q, \qquad K_t = x_t W_K, \qquad V_t = x_t W_V$$

Cache all previous keys and values, then append the new ones:

$$K_{1:t} \in \mathbb{R}^{B\times h\times t\times d_k}, \qquad V_{1:t} \in \mathbb{R}^{B\times h\times t\times d_v}$$

$$A_t = \mathrm{softmax}\left(\frac{Q_t K_{1:t}^\top}{\sqrt{d_k}}\right), \qquad y_t = A_t V_{1:t}$$

- softmax now runs over the t past positions
- caching avoids recomputing old K and V every step



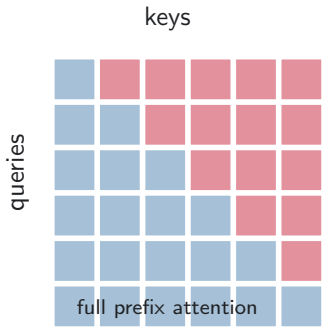
Prefill phase: full triangular attention

During the prompt (prefill), we process the entire sequence at once:

$$A = \text{softmax}\left(\frac{QK^\top + M}{\sqrt{d_k}}\right)$$

- Q, K, V computed for **all tokens**
- causal mask M removes future tokens
- produces a **lower triangular attention matrix**

cost: $O(n^2)$



Naive autoregressive decoding (no cache)

At each step t, we would recompute everything:

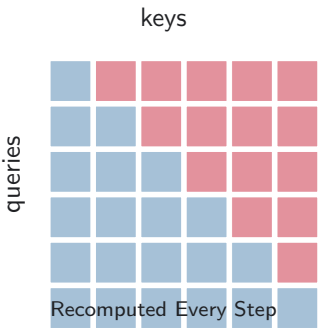
$$Q_{1:t}, K_{1:t}, V_{1:t}$$

- recompute attention over entire prefix
- repeated work for earlier tokens

total cost: $O(n^2)$ per step $\Rightarrow O(n^3)$

Key inefficiency

We keep recomputing the same K and V for past tokens.



KV caching: from quadratic to linear decoding

Instead of recomputing everything:

$$K_{1:t} = [K_{1:t-1}, K_t], \quad V_{1:t} = [V_{1:t-1}, V_t]$$

Only compute:

$$Q_t = x_t W_Q$$

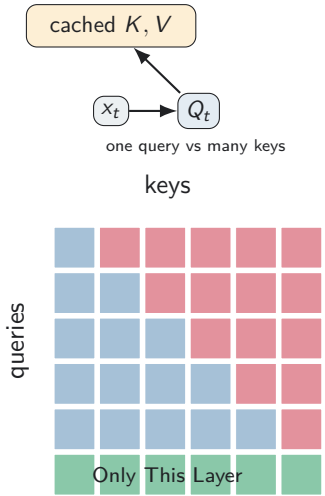
Then:

$$A_t = \text{softmax}\left(\frac{Q_t K_{1:t}^\top}{\sqrt{d_k}}\right)$$

$$y_t = A_t V_{1:t}$$

- attend over cached keys/values
- only one new query per step

cost per step: $O(t) \quad \Rightarrow \quad O(n)$



How context is built across layers

In a **single isolated layer**, Q, K, and V are just separate linear projections of the same input:

$$Q^{(l)} = X^{(l)} W_Q^{(l)}, \qquad K^{(l)} = X^{(l)} W_K^{(l)}, \qquad V^{(l)} = X^{(l)} W_V^{(l)}$$

Within that one layer, Q does **not** directly modify K or V. The dependency appears because Transformers are **stacks of layers**.

The chain reaction

1. **Layer 1 input:** raw token embeddings
2. **Layer 1 attention:** $Q^{(1)}$ and $K^{(1)}$ determine which tokens interact
3. **Layer 1 output:** tokens become *contextualized*
4. **Layer 2 input:** that contextualized output becomes the new input

So if token i uses its query to attend to nearby context in Layer 1, that context gets mixed into its output representation. By Layer 2, the new K and V are built from a representation that already knows more about the sentence.

Why Layer 1 queries matter for the KV cache

Let $X^{(l)}$ be the input to layer l. The attention output is

$$Z^{(l)} = \text{softmax}\left(\frac{Q^{(l)}(K^{(l)})^\top}{\sqrt{d_k}}\right) V^{(l)}$$

and this becomes the next layer's input:

$$X^{(l+1)} = Z^{(l)}$$

Why $Q^{(l)}$ matters

The query affects the **attention weights**, which determine how much of each value vector gets mixed into the output $Z^{(l)}$.

Since the next layer is built from that output,

$$K^{(l+1)} = X^{(l+1)} W_K^{(l+1)}, \qquad V^{(l+1)} = X^{(l+1)} W_V^{(l+1)}$$

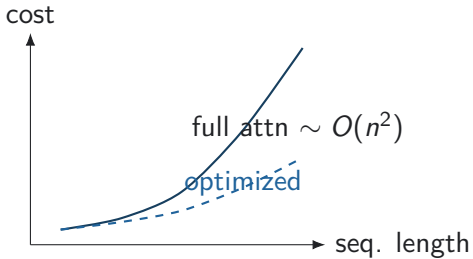
the Keys and Values in later layers inherit the effects of earlier attention.

Efficiency and common modern variants

Why efficiency matters

Full self-attention computes n^2 token-token interactions. Long contexts make both memory and compute expensive.

- FlashAttention: better memory/computation scheduling
- Multi-query attention: cheaper KV storage
- Ring / distributed variants: help large-scale multi-GPU training
- low-rank, sparse, or Nyström-style approximations: reduce cost of mixing. SSMs also may be used.



Takeaways

1. Attention computes weighted information exchange across tokens.
2. The $n \times n$ map is the heart of the mechanism: who reads from whom.
3. Scaling, masking, and multi-head structure make the primitive numerically stable and expressive.
4. A Transformer block wraps attention with residual paths, normalization, and a token-wise FFN.
5. Decoder training and inference differ: teacher forcing is parallel; generation is sequential and cache-heavy.

Attention is the operation. Transformers are the system built around it.