CS498: Algorithmic Engineering

Lecture 25: Multi-Head Attention & Positional Encoding

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 14

Outline

1. Attention: From BPE to Predictions
2. What Does One Head Learn?
3. Multi-Head Attention
4. Positional Encoding

1 Attention: From BPE to Predictions

2 What Does One Head Learn?

3 Multi-Head Attention

4 Positional Encoding

The Full Pipeline So Far

Last lecture: BPE tokenization. We converted our entire training corpus into a sequence of token IDs:

$$y_1,\ y_2,\ y_3,\ \ldots,\ y_T \quad \text{(integers from 0 to vocab_size} - 1)$$

Now we need to go from integers to vectors that a neural network can process.

Embedding matrix E: a learnable table of shape [vocab_size $\times$ d].

Token y_i maps to row y_i of E: $\quad x_i = E[y_i] \in \mathbb{R}^d$

This is just a lookup table. Initially random. After training, similar tokens have similar vectors.

Now we have: $x_1, x_2, \ldots, x_T \quad$ (a sequence of d-dimensional vectors).

The Task: Predict the Next Token

Language modeling: predict y_{i+1} given $y_1, \ldots, y_i$.

We have embedding vectors $x_1, \ldots, x_i$. But each vector only knows about its own token.

Problem: to predict y_{i+1}, token i needs information from the other tokens.

Example: "The cat sat on the ____"

Token "the" (position 5) needs to know there's a "cat" at position 2 and "sat" at position 3. Without that context, it can't predict "mat".

The question: how does each token gather information from the other tokens in the sequence?

Attention: The Idea

Each token x_i computes three vectors from its embedding:

- $q_i = x_i W_Q$ (the query: "what context do I need?")
- $k_i = x_i W_K$ (the key: "what do I contain?")
- $v_i = x_i W_V$ (the value: "what information do I provide?")

W_Q, W_K, W_V are learned matrices of shape $[d \times d_k]$. Same for every position.

The databsase analogy:

Token i is about to help predict y_{i+1}.

It asks a query/question (q_i) about the context.

Each previous token j answers (k_j). The dot product $q_i \cdot k_j$ measures: "how relevant is token j's answer to token i's question?"

The most relevant tokens contribute their value (v_j).

A Concrete Example

Three tokens: "cat", "sat", "mat". Embedding dimension $d = 4$, projection dimension $d_k = 2$.

Token	x_1	x_2	x_3	x_4
cat	1.0	0.0	1.0	0.0
sat	0.0	1.0	0.0	1.0
mat	1.0	0.0	0.5	0.5

After multiplying by learned W_Q, W_K, W_V (each 4×2):

Token	q_1	q_2	k_1	k_2	v_1	v_2
cat	1.0	0.0	1.0	0.0	2.0	0.0
sat	0.0	1.0	0.0	1.0	0.0	2.0
mat	1.0	0.0	0.5	0.5	1.5	0.5

Now: compute the score $s_{ij} = q_i \cdot k_j$ for every pair.

Attention Scores

$q_i \cdot k_j$	cat (k)	sat (k)	mat (k)
cat (q)	$1{\cdot}1 + 0{\cdot}0 = 1.0$	$1{\cdot}0 + 0{\cdot}1 = 0.0$	$1{\cdot}0.5 + 0{\cdot}0.5 = 0.5$
sat (q)	$0{\cdot}1 + 1{\cdot}0 = 0.0$	$0{\cdot}0 + 1{\cdot}1 = 1.0$	$0{\cdot}0.5 + 1{\cdot}0.5 = 0.5$
mat (q)	$1{\cdot}1 + 0{\cdot}0 = 1.0$	$1{\cdot}0 + 0{\cdot}1 = 0.0$	$1{\cdot}0.5 + 0{\cdot}0.5 = 0.5$

"cat" (q) has highest score with "cat" (k) = 1.0. Also attends to "mat" (0.5, a similar noun). Low score with "sat" (0.0, a verb).

Higher score = this token's question matches that token's description.

Now: turn these scores into weights (probabilities) using softmax.

Step 3: Softmax → Weights

All three rows after softmax:

Weights	cat	sat	mat
cat →	0.51	0.19	0.31
sat →	0.19	0.51	0.31
mat →	0.51	0.19	0.31

Each row is a probability distribution.
"cat" attends 51% to itself, 31% to mat, 19% to sat.

Step 4: Compute Output

Recall our value vectors: $\mathrm{v}_{\mathrm{cat}} = (2.0, 0.0)$, $\mathrm{v}_{\mathrm{sat}} = (0.0, 2.0)$, $\mathrm{v}_{\mathrm{mat}} = (1.5, 0.5)$.

Output for "cat":

$$\mathrm{out}_{\mathrm{cat}} = 0.51 \times (2.0,\ 0.0) + 0.19 \times (0.0,\ 2.0) + 0.31 \times (1.5,\ 0.5)$$

$$= (1.02,\ 0.0) + (0.0,\ 0.38) + (0.46,\ 0.16) = (1.48,\ 0.53)$$

Output for "sat":

$$\mathrm{out}_{\mathrm{sat}} = 0.19 \times (2.0,\ 0.0) + 0.51 \times (0.0,\ 2.0) + 0.31 \times (1.5,\ 0.5)$$

$$= (0.38,\ 0.0) + (0.0,\ 1.02) + (0.46,\ 0.16) = (0.84,\ 1.17)$$

Each token's output is enriched with information from tokens it attends to.

That's Attention. The Formula.

What we just computed, in one equation:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(QK^\top)\, V$$

- $QK^\top$ = score matrix (how much each query matches each key)
- softmax = turn scores into probability weights (per row)
- $\times V$ = weighted sum of value vectors

Complexity: $O(T^2 \cdot d)$ where T = sequence length, d = dimension.

The T^2 comes from the score matrix: every token attends to every other token.

Note: We usually normalize by $\sqrt{d_k}$:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

Interpreting the Outputs

Before attention: $\mathrm{v}_{\mathrm{cat}} = (2.0, 0.0)$. Pure "cat" information.

After attention: $\mathrm{out}_{\mathrm{cat}} = (1.48, 0.53)$.

The output is a blend:

- Mostly cat-like (large first component: 1.48)
- Some mat-like and sat-like influence (second component: 0.53)

Token "cat" now knows about the other tokens in the sequence.

Before attention: each token is isolated. After attention: each token carries information about the whole sequence.

Why Scale by $\sqrt{d_k}$?

If $q, k \in \mathbb{R}^{d_k}$ with entries $\sim N(0, 1)$:

$$q \cdot k = \sum_{i=1}^{d_k} q_i k_i \quad \Rightarrow \quad \mathrm{Var}(q \cdot k) = d_k$$

For $d_k = 512$: dot products have std ≈ 23. Softmax saturates: one weight ≈ 1, rest ≈ 0. Gradients vanish.

Fix: divide by $\sqrt{d_k}$ before softmax.

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V$$

In our example $d_k = 2$ so $\sqrt{d_k} \approx 1.4$. The effect is small. For $d_k = 512$ it's critical.

Causal Masking

For language generation: when predicting x_{i+1}, token i should not see future tokens.

Set scores for $j > i$ to $-\infty$ before softmax → those weights become 0.

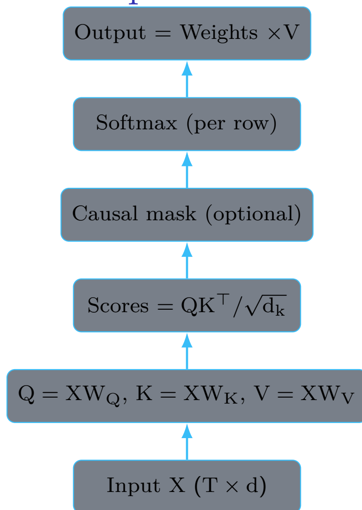
Scores	cat	sat	mat
cat	1.0	$-\infty$	$-\infty$
sat	0.0	1.0	$-\infty$
mat	1.0	0.0	0.5

After softmax:

Weights	cat	sat	mat
cat	1.00	0.00	0.00
sat	0.27	0.73	0.00
mat	0.51	0.19	0.31

Each position only attends to previous positions and itself.

Attention: The Complete Pipeline



Every step is a matrix operation. Fully parallelizable on GPUs.

Efficient Attention: FlashAttention

Standard attention: A "naive" implementation of Attention stores full $T \times T$ matrix in GPU memory.

Memory: $O(T^2)$. For $T = 8192$: that's 256 MB just for the attention matrix (fp32).

FlashAttention: tiles the computation, keeps everything in fast on-chip SRAM.

- Same mathematical output (not an approximation).
- $O(T)$ memory instead of $O(T^2)$.
- 2–4× faster in practice (fewer memory reads).

In PyTorch: F.scaled_dot_product_attention(Q, K, V, is_causal=True) uses FlashAttention automatically when available. Same math, faster, less memory.

1. Attention: From BPE to Predictions
2. What Does One Head Learn?
3. Multi-Head Attention
4. Positional Encoding

One Head = One Type of Relationship

The matrices W_Q, W_K, W_V define:

- What "question" each token asks (W_Q)
- What "answer" each token advertises (W_K)
- What information each token provides (W_V)

A single set of W_Q, W_K, W_V learns one type of relationship.

Example: maybe this head learned "Do we both interact together often?"
$\rightarrow$ cat attends to mat, dog attends to bone.
But it cannot simultaneously learn "am I the verb's subject?"

Language Has Many Relationship Types

- Syntactic: subject-verb agreement (“The cats are sleeping”)
- Semantic: meaning similarity (“bank” near “river” vs “money”)
- Positional: nearby words often relate
- Coreference: pronouns $\rightarrow$ nouns (“it” $\rightarrow$ “the mat”)
- Functional: in code, “def foo():” $\rightarrow$ “foo” is called later

One attention head can only capture one of these. We need more.

Analogy to CNNs

In a CNN:

- One kernel detects one spatial pattern (e.g., vertical edges).
- We use 32 or 64 kernels to detect many patterns.

In attention:

- One head captures one type of relationship.
- We use multiple heads to capture many relationship types.

Just like CNN kernels, each attention head specializes in something different.

Key insight: multi-head attention = multiple parallel attention operations, each with its own W_Q, W_K, W_V.

1 Attention: From BPE to Predictions

2 What Does One Head Learn?

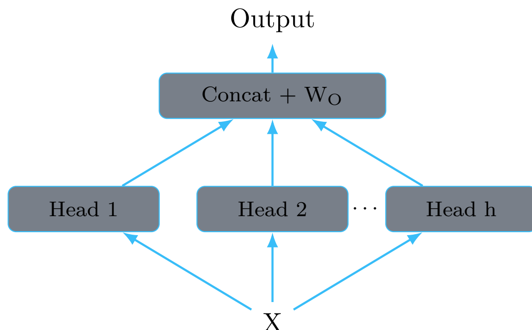
3 Multi-Head Attention

4 Positional Encoding

Multi-Head Attention: The Idea

Instead of one set of (W_Q, W_K, W_V) with $d_k = d$:

- Use h heads, each with $d_k = d/h$.
- Each head independently computes attention.
- Concatenate all heads' outputs.
- Project back to d dimensions with W_O.



Concrete Example

$d = 6$, $h = 3$ heads, so $d_k = 6/3 = 2$ per head.

Head 1: W_Q^1, W_K^1, W_V^1 each 6×2. Might learn: "are we both nouns?"
Head 2: W_Q^2, W_K^2, W_V^2 each 6×2. Might learn: "am I the verb and are you my subject?"
Head 3: W_Q^3, W_K^3, W_V^3 each 6×2. Might learn: "are we in the same sentence?"

Each head produces a 2-dim output per token.
Concatenate: $[\text{head1_out};\ \text{head2_out}; \text{head3_out};] = $ 6-dim vector.
Multiply by W_O (6×6) to mix the heads.

Final output: 6-dim per token (same dimension as input).

Multi-Head Attention: The Math

For each head $i \in \{1, \ldots, h\}$:

$$Q_i = XW_Q^i, \quad K_i = XW_K^i, \quad V_i = XW_V^i$$
$$\text{head}_i = \text{Attention}(Q_i, K_i, V_i)$$

Combine all heads:

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

Parameter count: Same total parameters as single-head with $d_k = d$. But distributed across h independent attention computations.

What Different Heads Learn (Empirically)

In trained transformers, researchers found specialized heads:

- Positional heads: attend to the previous word
- Syntactic heads: attend to the subject of a verb
- Delimiter heads: attend to the end of the sentence
- Rare-word heads: attend to unusual or important words
- Induction heads: detect and continue repeated patterns

The model discovers these patterns automatically through gradient descent. Nobody hand-designs which head learns what.

1. Attention: From BPE to Predictions
2. What Does One Head Learn?
3. Multi-Head Attention
4. Positional Encoding

The Missing Ingredient

Attention is permutation-invariant:

The score $q_i \cdot k_j$ depends on the content of tokens i and j, not their positions.

"The cat sat" and "sat cat the" produce the same attention scores (just reordered rows/columns).

But word order matters!

- "Dog bites man" $\neq$ "Man bites dog"
- "You should never trust politicians" $\neq$ "Politicians should never trust you"

We need to inject position information into the model.

Option 1: Learned Position Embeddings

Add a learnable position vector:

$$x_i = E[id(i)] + \text{position_embedding}[i]$$

- One learnable vector per position: $p_1, p_2, \ldots, p_T \in \mathbb{R}^d$.
- Simple. Used in GPT-2. The model learns that position 1 "feels" different from position 100.

Problem: fixed max sequence length; if position 1025 was never seen during training $\rightarrow$ no embedding for it. Cannot generalize.

Option 2: RoPE (Modern)

Rotary Position Embedding (Su et al., 2021).

Instead of adding position to the embedding, rotate Q and K by a position-dependent angle.

For each pair of dimensions $(2i, 2i+1)$:

$$\begin{pmatrix} q'_{2i} \\ q'_{2i+1} \end{pmatrix} = \begin{pmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{pmatrix} \begin{pmatrix} q_{2i} \\ q_{2i+1} \end{pmatrix}$$

where m = position and $\theta_i = 10000^{-2i/d}$.

Used in Llama, Qwen, Mistral, and most modern LLMs.

RoPE: Why Rotation Gives Relative Position

Key mathematical property:

If $q'_i = R(i) \cdot q_i$ and $k'_j = R(j) \cdot k_j$ where $R(\cdot)$ is a rotation matrix:

$$q'_i \cdot k'_j = q_i^\top R(i)^\top R(j)\, k_j = q_i^\top R(j-i)\, k_j$$

The score depends only on $j - i$ (the distance between tokens).

Example:

- Tokens at positions 3 and 7 (distance 4)
- Tokens at positions 100 and 104 (also distance 4)
- Same rotational effect on the attention score

The attention score $q'_i \cdot k'_j$ depends on relative position $|i - j|$, not absolute positions. This is why RoPE captures relative position naturally.

Where Position Info Gets Added

Learned Positional Embeddings:

- Add to the embedding before attention (at the input).
- Position information is added once. Must survive through many layers.

RoPE:

- Apply to Q and K inside each attention layer.
- Every layer gets fresh position information.
- Does not affect V, so the content is position-free.

With RoPE, every layer gets fresh position information. With additive, it's added once and must survive through many layers.

Why Not Just Concatenate Position?

Alternative idea: append position as an extra feature. $x_i = [\text{embedding}; i]$.

Problems:

- Increases dimension by 1. Seems small, but breaks symmetry of the embedding space.
- Position i is an integer. Embedding dimensions are continuous. Scale mismatch.
- The model must learn to "ignore" the position dimension for content and "ignore" content for position.