Understanding Attention: From Signals to Learning

Berkay Guler

Center for Pervasive Communications and Computing
University of California, Irvine

October 28, 2025

Outline

1. Motivation: The Problem We are Solving
2. Core Intuition: Adaptive Weighted Combinations
3. Scaling Attention
4. Multi-Head Attention: Parallel Processing
5. Self-Attention: When Everything Comes from One Source
6. Why Attention Works: A Computational View
7. Applications Across Domains
8. Limitations and Conclusions

Goal: Build understanding, not memorize formulas.

Part 1: Motivation

The Fundamental Problem

Classic Signal Processing: Given n signals $x_1, x_2, \ldots, x_n$, compute:

$$y = \sum_{i=1}^{n} w_i x_i$$

where $x_i \in \mathbb{R}^d$ (signal vector), $w_i \in \mathbb{R}$ (scalar weight), $y \in \mathbb{R}^d$ (output).

Key observation: The weights w_i are usually *fixed* and predetermined.

But what if:

- The relevant signals depend on *context*?
- We want weights w_i that *adapt* based on what we're looking for?
- We want to learn which signals matter?

Reformulating the Problem

Instead of fixed weights, let's make them *data-dependent*:

$$w_i = f(\text{context}, x_i)$$

where:

- context $\in \mathbb{R}^d$: query vector (what we're looking for)
- $x_i \in \mathbb{R}^d$: the i-th signal vector
- $w_i \in \mathbb{R}$: scalar weight for signal i

Questions:

1. What is "context"? How do we represent it?
2. How should context and x_i interact to determine w_i?
3. How do we ensure weights are meaningful (e.g., sum to 1)?
4. Can we compute this efficiently?

Part 2: Core Intuition

Three Representations: Query, Key, Value

Consider n elements with key-value pairs: $\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$

- $i \in \{1, 2, \ldots, n\}$: **Index** — element identifier
- $k_i \in \mathbb{R}^d$: **Key** — descriptor/fingerprint of element i
- $v_i \in \mathbb{R}^d$: **Value** — actual content of element i
- $q \in \mathbb{R}^d$: **Query** — what we're looking for

Goal: Weighted combination of values, where weights reflect key-query relevance.

Note: The value dimension can be different than that of queries and keys, i.e. d, but keys and queries must have the same dimension due to the dot product computation in the next page.

Intuition (information retrieval):

- Index i: Book number in library catalog
- Query q: "I'm searching for documents about machine learning"
- Key k_i: Document metadata/summary for book i
- Value v_i: Full document content of book i

Step 1: Compute Relevance Scores

How relevant is element i to our query? Measure via dot product:

$$s_i = q^T k_i \in \mathbb{R}$$

Why dot product?

- Correlation-based (from signal processing)
- s_i is large when q and k_i are aligned
- s_i is small when orthogonal/opposite
- Differentiable and efficient

Example scores:

$$s = [2.1, -0.5, 3.8, 0.2] \in \mathbb{R}^4$$

Element 3 is most relevant (highest score).

Step 2: Normalize via Softmax

Convert scores $s_i \in \mathbb{R}$ to weights $w_i \in \mathbb{R}$ via softmax:

$$w_i = \text{softmax}(s)_i = \frac{\exp(s_i)}{\sum_{j=1}^{n} \exp(s_j)}$$

Properties:

- $0 < w_i < 1$ for all i
- $\sum_i w_i = 1$ — probability distribution
- $w_i > w_j$ iff $s_i > s_j$ — preserves ordering
- Differentiable everywhere

Alternatives: Any function that normalizes to a probability distribution works:

- Sparsemax: produces sparse weights (some exactly zero)
- Sigmoid + normalize: $w_i = \frac{\sigma(s_i)}{\sum_j \sigma(s_j)}$
- Hardmax: $w_i = 1$ if $i = \arg\max_j s_j$, else $w_i = 0$ (not differentiable)

Softmax is standard due to smoothness and gradient properties.

Step 3: Compute Weighted Sum

Combine values using computed weights:

$$y = \sum_{i=1}^{n} w_i v_i \in \mathbb{R}^d$$

where $w_i \in \mathbb{R}$ (weight), $v_i \in \mathbb{R}^d$ (value), $y \in \mathbb{R}^d$ (output).

Complete process:

1. Score: $s_i = q^T k_i \in \mathbb{R}$
2. Weight: $w_i = \text{softmax}(s)_i \in \mathbb{R}$
3. Output: $y = \sum w_i v_i \in \mathbb{R}^d$

Result: An *adaptive superposition* of values, weighted by query-key relevance.

Example: Setup

Toy example: $d = 2$ (dimension), $n = 3$ (elements)

Query: $q = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \in \mathbb{R}^2$

Keys: $k_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad k_2 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}, \quad k_3 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$

Values: $v_1 = \begin{bmatrix} 10 \\ 0 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 0 \\ 10 \end{bmatrix}, \quad v_3 = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$

Step 1: Scores $s_i = q^T k_i$

$$s_1 = 1, \quad s_2 = 1, \quad s_3 = -2$$

Score vector: $s = [1, 1, -2]^T \in \mathbb{R}^3$

Example: Softmax

Step 2: Softmax
Exponentials:

$$\exp(1) \approx 2.718, \quad \exp(1) \approx 2.718, \quad \exp(-2) \approx 0.135$$

Partition: $Z = 2.718 + 2.718 + 0.135 = 5.571$
Weights:

$$w_1 = \frac{2.718}{5.571} \approx 0.488, \quad w_2 \approx 0.488, \quad w_3 \approx 0.024$$

Check: $0.488 + 0.488 + 0.024 = 1.000$
Weight vector: $w = [0.488, 0.488, 0.024]^T$

Example: Output

Step 3: Weighted sum

$$\text{output} = w_1 v_1 + w_2 v_2 + w_3 v_3$$

$$= 0.488 \begin{bmatrix} 10 \\ 0 \end{bmatrix} + 0.488 \begin{bmatrix} 0 \\ 10 \end{bmatrix} + 0.024 \begin{bmatrix} 5 \\ 5 \end{bmatrix}$$

$$= \begin{bmatrix} 4.88 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 4.88 \end{bmatrix} + \begin{bmatrix} 0.12 \\ 0.12 \end{bmatrix} = \begin{bmatrix} 5.00 \\ 5.00 \end{bmatrix}$$

Interpretation: Output is dominated by v_1 and v_2 (weights 0.488 each), element 3 barely contributes.

Key Insight: Adaptivity

Same data, different query: $q' = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$

Scores:

$$s'_1 = -1, \quad s'_2 = -1, \quad s'_3 = 2$$

After softmax:

$$w'_1 \approx 0.045, \quad w'_2 \approx 0.045, \quad w'_3 \approx 0.909$$

Now element 3 dominates!

Key point: Same data, different queries yield different attention patterns. This adaptivity makes attention powerful.

Attention: Compact Formula

Given:

- Query $q \in \mathbb{R}^d$
- Keys $K = [k_1, \ldots, k_n] \in \mathbb{R}^{d \times n}$
- Values $V = [v_1, \ldots, v_n] \in \mathbb{R}^{d \times n}$

Compact formula:

$$\text{Attention}(q, K, V) = V \cdot \text{softmax}(K^T q)$$

Expanded:

$$\text{output} = \sum_{i=1}^{n} \frac{\exp(q^T k_i)}{\sum_{j=1}^{n} \exp(q^T k_j)} v_i$$

Everything is differentiable and efficient via matrix operations.

Part 3: Scaling

The Scaling Issue

When d is large, dot products $q^T k_i$ have large magnitude.
Problem: Softmax becomes very peaked (one weight 1, others 0).

Softmax gradient: For $w_i = \frac{\exp(s_i)}{\sum_j \exp(s_j)}$, we have:

$$\frac{\partial w_i}{\partial s_i} = w_i(1 - w_i)$$

Derivation: $\frac{\partial}{\partial s_i}\left[\frac{\exp(s_i)}{Z}\right] = \frac{\exp(s_i)\cdot Z - \exp(s_i)\cdot\exp(s_i)}{Z^2} = w_i - w_i^2 = w_i(1 - w_i)$
When softmax is peaked ($w_i \approx 1$), gradient $\to 0$ (vanishing gradients).

Solution: Scale dot products by $1/\sqrt{d}$:

$$s_i = \frac{q^T k_i}{\sqrt{d}}$$

This keeps s_i in reasonable range regardless of d.

Why Scale by $\sqrt{d}$?

Variance analysis:

Assume q, k_i have i.i.d. components with mean 0, variance 1.
The dot product is:

$$q^T k_i = \sum_{j=1}^{d} q_j k_{i,j}$$

Since components are independent:

$$\text{Var}(q^T k_i) = \sum_{j=1}^{d} \text{Var}(q_j k_{i,j}) = \sum_{j=1}^{d} \text{Var}(q_j)\text{Var}(k_{i,j}) = d$$

Therefore: $\text{Std}(q^T k_i) = \sqrt{d}$

After scaling: $\text{Var}\left(\frac{q^T k_i}{\sqrt{d}}\right) = \frac{d}{d} = 1$

Result: Scores have unit variance (standard deviation = 1) regardless of dimension d. This keeps softmax outputs balanced rather than extremely peaked, which maintains healthy gradients for learning.

Extending to Multiple Queries (1/3)

So far: Single query vector $q \in \mathbb{R}^d$ produces single output.

$$y = \text{Attention}(q, K, V) = V \cdot \text{softmax}\left(\frac{K^T q}{\sqrt{d}}\right) \in \mathbb{R}^d$$

In practice: We often need to process m different queries **in parallel**.

Question: Can we compute attention for multiple queries efficiently?
Answer: Yes! Stack them into a matrix and use matrix operations.

Extending to Multiple Queries (2/3)

Stack queries: $Q = [q_1; q_2; \ldots; q_m] \in \mathbb{R}^{m \times d}$ (each row is a query)
Matrix formulation:

$$Y = \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right) V \in \mathbb{R}^{m \times d}$$

Output: $Y = [y_1; y_2; \ldots; y_m] \in \mathbb{R}^{m \times d}$ (each row y_i is output for query q_i)

Dimensions:

- $QK^T \in \mathbb{R}^{m \times n}$: all query-key scores
- Softmax applied row-wise: each query gets its own distribution
- $Y \in \mathbb{R}^{m \times d}$: one output row per query row

Extending to Multiple Queries (3/3)

CRITICAL: Queries are processed independently!
Each query q_i gets its own output y_i with NO interaction between queries.

Analogy:

- **Correct:** Ask m questions $\rightarrow$ get m separate answers
- **Wrong:** Ask m questions $\rightarrow$ get 1 combined answer

Purpose: Computational efficiency only (parallel GPU processing).
Each query follows the same three steps: score $\rightarrow$ normalize $\rightarrow$ weighted sum.

Scaled Attention Formula

Scaled dot-product attention (standard):

$$Y = \text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

where:

- $Y \in \mathbb{R}^{m \times d}$: m answers (one output per query)
- $Q \in \mathbb{R}^{m \times d}$: m queries
- $K \in \mathbb{R}^{n \times d}$: n keys
- $V \in \mathbb{R}^{n \times d}$: n values
- $\sqrt{d}$: scaling factor

Dimensions:

- $QK^T \in \mathbb{R}^{m \times n}$: query-key scores

Part 4: Multiple Heads

Motivation for Multiple Heads

Problem: One attention operation may not capture all relationships.

Library analogy: Suppose you are researching "neural networks"
Single attention head: One search strategy

- Search by title keywords only
- Might miss books categorized differently

Multiple attention heads: Different search strategies in parallel

- Head 1: Search by title keywords
- Head 2: Search by author expertise
- Head 3: Search by publication date (recent work)
- Head 4: Search by citation patterns (influential work)

Each head uses different Keys (different ways to describe/index the same books).
Result: Combine insights from all search strategies for comprehensive results.

What is a "Head"?

Definition: A "head" is one complete attention operation with its own learned projections.

Single head attention:

- Learn: $W_Q \in \mathbb{R}^{d \times d_k}$, $W_K \in \mathbb{R}^{d \times d_k}$, $W_V \in \mathbb{R}^{d \times d_v}$
- Project: $Q = X_Q W_Q \in \mathbb{R}^{m \times d_k}$, $K = X_K W_K \in \mathbb{R}^{n \times d_k}$, $V = X_V W_V \in \mathbb{R}^{n \times d_v}$
- Compute: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \in \mathbb{R}^{m \times d_v}$

Multi-head attention: Run h attention operations in parallel, each head $i \in \{1, \ldots, h\}$:

- Head i projects with: $W_Q^{(i)} \in \mathbb{R}^{d \times d_k}$, $W_K^{(i)} \in \mathbb{R}^{d \times d_k}$, $W_V^{(i)} \in \mathbb{R}^{d \times d_v}$
- Head i output: $Y_i = \text{Attention}(X_Q W_Q^{(i)}, X_K W_K^{(i)}, X_V W_V^{(i)}) \in \mathbb{R}^{m \times d_v}$

Key idea: Different $W^{(i)}$ matrices $\rightarrow$ different "views" $\rightarrow$ capture different relationships.

Multi-Head: Concatenation and Output

After computing all heads: We have h outputs, each of dimension $m \times d_v$

$$Y_1 \in \mathbb{R}^{m \times d_v}, \quad Y_2 \in \mathbb{R}^{m \times d_v}, \quad \ldots, \quad Y_h \in \mathbb{R}^{m \times d_v}$$

Step 1: Concatenate all head outputs horizontally:

$$\text{Concat} = [Y_1 \mid Y_2 \mid \cdots \mid Y_h] \in \mathbb{R}^{m \times (h \cdot d_v)}$$

where $|$ denotes horizontal concatenation.

Step 2: Apply output projection to map back to dimension d:

$$\text{MultiHead}(X_Q, X_K, X_V) = \text{Concat} \cdot W_O \in \mathbb{R}^{m \times d}$$

where $W_O \in \mathbb{R}^{(h \cdot d_v) \times d}$ is a learned output projection matrix.

Note: The choices of d_k and d_v are design decisions. Common choice: $d_k = d_v = d/h$ for computational balance. Note that $h \cdot d_v = d$ keeps the concatenated dimension equal to the input dimension, but this is not required due to the output projection W_O.

Multi-Head Attention: Summary

Complete formula:

$$\text{MultiHead}(X_Q, X_K, X_V) = \text{Concat}(Y_1, \ldots, Y_h) W_O$$

where:

$$Y_i = \text{Attention}(X_Q W_Q^{(i)}, X_K W_K^{(i)}, X_V W_V^{(i)})$$

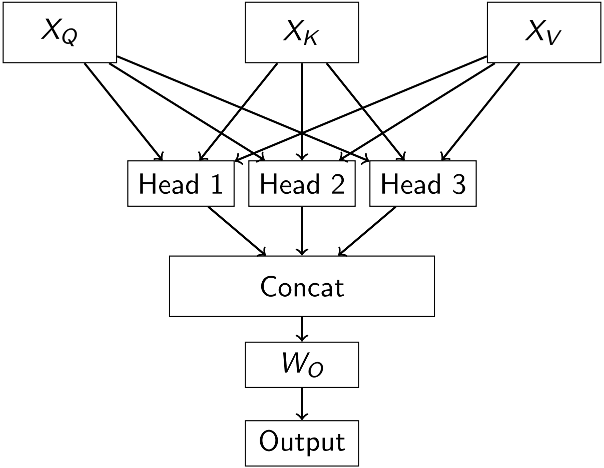
Learned parameters for h heads:

- h sets of projection matrices: $\{W_Q^{(i)} \in \mathbb{R}^{d \times d_k}, W_K^{(i)} \in \mathbb{R}^{d \times d_k}, W_V^{(i)} \in \mathbb{R}^{d \times d_v}\}_{i=1}^{h}$
- One output projection: $W_O \in \mathbb{R}^{(h \cdot d_v) \times d}$

Key properties:

- Each head operates independently (parallel computation)
- Each head output dimension is d_v (commonly $d_v = d/h$)
- Query/key dimensions are d_k (commonly $d_k = d/h$)
- More expressive: multiple different "views" of the relationships

Multi-Head Flow



Part 5: Self-Attention

Self-Attention: Everything from a Single Input

So far: We had separate inputs $X_Q \in \mathbb{R}^{m \times d}$, $X_K \in \mathbb{R}^{n \times d}$, $X_V \in \mathbb{R}^{n \times d}$.

Self-Attention: All three come from the same input $X \in \mathbb{R}^{n \times d}$:

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

where $W_Q, W_K \in \mathbb{R}^{d \times d_k}$ and $W_V \in \mathbb{R}^{d \times d_v}$ are learned projection matrices.

Note: Since $X_Q = X_K = X_V = X$, we have $m = n$ (number of queries equals number of keys/values).

Interpretation:

- Input X has n rows (elements), each of dimension d
- Each row serves as a query (via projection W_Q)
- All n rows serve as keys (via projection W_K) and values (via projection W_V)
- Output: Each row i attends to all rows $j = 1, \ldots, n$
- Result: Each row gets an updated representation based on all other rows

Key insight: All n rows communicate with each other simultaneously.

Self-Attention Example (1/2)

Setup: $n = 3$ rows (elements), dimension $d = 2$

Input: $X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \in \mathbb{R}^{3\times 2}$

Assume $W_Q = W_K = W_V = I_2$ (identity), so $Q = K = V = X$

Compute scores: $QK^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix}$

Scaled: $\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix} \approx \begin{bmatrix} 0.707 & 0 & 0.707 \\ 0 & 0.707 & 0.707 \\ 0.707 & 0.707 & 1.414 \end{bmatrix}$

Observation:

- Entry (i, j) captures the similarity between query i and key j
- Row 3 (which is $[1, 1]$) has highest score with itself (1.414)
- Row 3 also attends strongly to rows 1 and 2 (scores 0.707 each)
- Rows 1 and 2 are orthogonal (score 0)

Self-Attention Example (2/2)

Apply softmax row-wise:

Row 1: $\exp([0.707, 0, 0.707]) \to \text{normalize} \to w_1 \approx [0.401, 0.198, 0.401]$
Row 2: $\exp([0, 0.707, 0.707]) \to \text{normalize} \to w_2 \approx [0.198, 0.401, 0.401]$
Row 3: $\exp([0.707, 0.707, 1.414]) \to \text{normalize} \to w_3 \approx [0.248, 0.248, 0.504]$

$$\text{Attention weights: } W_{\text{attn}} \approx \begin{bmatrix} 0.401 & 0.198 & 0.401 \\ 0.198 & 0.401 & 0.401 \\ 0.248 & 0.248 & 0.504 \end{bmatrix}$$

$$\textbf{Output: } Y = W_{\text{attn}} V = \begin{bmatrix} 0.401 & 0.198 & 0.401 \\ 0.198 & 0.401 & 0.401 \\ 0.248 & 0.248 & 0.504 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \approx \begin{bmatrix} 0.802 & 0.599 \\ 0.599 & 0.802 \\ 0.752 & 0.752 \end{bmatrix}$$

Note: W_{attn} is not symmetrical in general. Here, it is symmetrical since we have used the same projection I_2 for queries, keys, and values. In practice, we learn different projections for each.

Interpretation: Information is being mixed across all rows.

Why Do We Need Keys? Why Not Just Query-Value?

Valid alternative: Compute $s_i = q^T v_i$, then $w_i = \text{softmax}(s)_i$, then $y = \sum w_i v_i$.
So why use separate Keys?

Reason 1: Decoupling matching from content

- Keys: optimized for computing relevance/similarity
- Values: optimized for information content to propagate
- Same input can be "easy to match" (via W_K) yet "rich in content" (via W_V)

Reason 2: Dimensional flexibility

- Can use $d_k \neq d_v$ (e.g., smaller d_k for efficiency in score computation)
- Scores are $O(n^2 d_k)$, output is $O(n^2 d_v)$ — can optimize each separately

Reason 3: Empirical performance

- Two learned transformations (W_K, W_V) more expressive than one
- Model learns: "match on these features, retrieve different features"
- Significant performance gains in practice

Causal (Masked) Attention

Problem: In causal generation (auto-regressive models), position t shouldn't see future ($> t$).
Solution: Mask future positions before softmax:

$$\tilde{s}_{i,j} = \begin{cases} \frac{q_i^T k_j}{\sqrt{d}} & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases}$$

After softmax: $\exp(-\infty) = 0$, so future positions get zero weight.
Result: Position i only attends to $1, \ldots, i$ (causal).

Causal Masking Example

Unmasked scores:

$$\begin{bmatrix} 0.5 & 1.2 & 0.8 \\ 0.3 & 0.9 & 1.1 \\ 0.2 & 0.7 & 0.4 \end{bmatrix}$$

Apply causal mask: Row i (time $t = i$) can only attend to rows $j \leq i$

After causal mask:

$$\begin{bmatrix} 0.5 & -\infty & -\infty \\ 0.3 & 0.9 & -\infty \\ 0.2 & 0.7 & 0.4 \end{bmatrix}$$

After softmax:

$$\begin{bmatrix} 1.0 & 0 & 0 \\ 0.24 & 0.76 & 0 \\ 0.15 & 0.49 & 0.36 \end{bmatrix}$$

- Row 1 (t=1): attends only to itself
- Row 2 (t=2): attends to rows 1-2
- Row 3 (t=3): attends to all rows 1-3

Part 6: Why Attention Works

Computational Advantage: Adaptivity

Fixed architectures with hard-coded inductive biases:

RNN inductive bias: Sequential processing, recent context matters most

- Always combines h_{t-1} (previous state) and x_t (current input) the same way
- Information from distant past must flow through all intermediate steps
- Assumes temporal locality: nearby elements are most relevant

CNN inductive bias: Spatial locality and translation invariance

- Always looks at fixed local neighborhood (e.g., 3×3 receptive field)
- Same filter applied everywhere (translation invariance)
- Global context requires stacking many layers

Attention: Data-dependent, learned relevance

- Each element looks at *whatever is relevant*, not just neighbors
- Relevant context learned from data, not assumed by architecture
- Direct connections between any pair of elements

Trade-off: More flexible, but requires more data to learn patterns that RNN/CNN get "for free."

Expressiveness

1. **Aggregation (weighted sum):** This is what attention does directly
   - Attention weights w_i determine contribution of each element
2. **Filtering (suppress irrelevant):** Learn to assign near-zero weights
   - If $q^T k_i$ is very negative, then $w_i \approx 0$ after softmax
3. **Routing (select subset):** Learn sharp attention on a few elements
   - Make one or a few w_i large (e.g., $w_1 = 0.69, w_2 = 0.29$, rest ≈ 0)
   - Since $\sum w_i = 1$, this concentrates attention on selected elements
4. **Sorting (attend in order):** Most complex—requires multiple layers
   - Example: Input sequence $[5, 2, 8, 1]$, want to process in sorted order
   - Layer 1: Learn to encode "which element is smallest/largest"
   - Layer 2: Position 1 learns to attend most to element with value 1
   - Layer 3: Position 2 learns to attend most to element with value 2, etc.
   - Result: Each output position aggregates information in sorted order
   - Not physically reordering, but retrieving information as if sorted

Key insight: By learning appropriate Q, K, V projections, attention implements diverse computational patterns.

Optimization

Gradient flow: Fully differentiable through:

1. Softmax to scores
2. Dot products to Q, K, V
3. Learned projections W_Q, W_K, W_V

Scaling benefit: $1/\sqrt{d}$ keeps gradients stable.
Result: End-to-end training via backprop works effectively.

Efficiency: Parallelization

Complexity: For sequence length n:

- All pairwise scores: $O(n^2 d)$
- Softmax and weighting: $O(n^2 d)$
- Total: $O(n^2 d)$ per layer

Parallelization:

- No recurrence: h_t doesn't depend on h_{t-1}
- Process entire sequence simultaneously
- Natural fit for GPU/TPU

Trade-off: Higher memory, but massive parallelism.

Part 7: Applications

Application 1: Language Modeling

Task: Predict next token given context.

Approach:

1. Embed tokens
2. Stack of multi-head self-attention with causal mask
3. Each layer: position attends to context
4. Output: logits over vocabulary

Why attention helps:

- Long-range dependencies (100+ tokens back)
- Dynamic context (learns what's relevant)
- Parallelization (train on thousands of tokens)

Examples: GPT, LLaMA, Claude, ...

Application 2: Vision Transformers

Apply attention to images:

1. Divide image into patches (e.g., 16×16)
2. Embed each patch (i.e. project to some other space)
3. Self-attention: each patch attends to all patches
4. Use output for classification/other tasks

Key insight: Patches are like tokens.
Advantage over CNNs: No hard-coded locality. Model learns spatial relationships.

Application 3: Graph Attention Networks (1/2)

Graph-structured data:

1. Each node has features
2. Each node attends to its neighbors
3. Compute attention weights for neighbors
4. Update: weighted combination

Setup: Graph with N nodes, node i has features $h_i \in \mathbb{R}^d$, neighborhood $\mathcal{N}(i)$

Key idea: Each node aggregates information from its neighbors using attention weights based on feature similarity.

Useful for: social networks, molecular graphs, and knowledge graphs.

Application 3: Graph Attention Networks (2/2)

Attention mechanism with Q, K, V projections:
Project features: $q_i = W_Q h_i$, $k_j = W_K h_j$, $v_j = W_V h_j$
Compute scores: $s_{i,j} = \frac{q_i^T k_j}{\sqrt{d_k}}$ for each neighbor $j \in \mathcal{N}(i)$
Normalize: $w_{i,j} = \frac{\exp(s_{i,j})}{\sum_{k \in \mathcal{N}(i)} \exp(s_{i,k})}$
Update: $h'_i = \sum_{j \in \mathcal{N}(i)} w_{i,j} v_j$ **Key differences from standard Transformers:**

- Node i only attends to neighbors $\mathcal{N}(i)$, not all nodes (sparse attention)
- Attention is constrained by graph structure
- $W_Q, W_K, W_V \in \mathbb{R}^{d_k \times d}$ are learned projection matrices

Application 4: Cross-Attention

Query from one source, Keys/Values from another.

Recall the library analogy: Keys and values come from the same source, whereas the queries are independent!

Example: Generate text describing an image (Image-to-Text)

- Image encoder $\rightarrow$ image features
- Language decoder (with causal attention for autoregressive generation) generates text
- Decoder queries: "What in image is relevant?"
- Image features serve as Keys/Values

General pattern: Align two modalities via attention.
Other examples:

- Machine translation: source language is K/V, target language is Q
- Visual question answering (QA): question is Q, image regions are K/V

Application 5: Control and Robotics

Transformers for decision-making:

- Policy network takes history of observations/actions
- Self-attention: each timestep attends to relevant past $\longrightarrow$ **causal attention!**
- Output: action to take

Why useful:

- Long-horizon reasoning (current action depends on distant past)
- Composable (combine heterogeneous information)
- Scalable (same architecture for different tasks)

Example: Navigation, manipulation from demonstrations, etc.

Why Attention Appears Everywhere

Attention solves a fundamental problem:

> *How do I adaptively combine multiple information sources based on relevance?*

Wherever you have:

1. Multiple information sources
2. Need to select/aggregate based on context
3. Benefit from learning what's relevant

$\longrightarrow$ Attention likely helps.

This generality is rare. It's a primitive operation many tasks need.

Part 8: Limitations and Conclusions

Computational Limitations

Main constraint: Quadratic complexity $O(n^2)$ in sequence length.
Consequences:

- Long documents ($n > 10{,}000$): memory prohibitive
- Real-time processing: may not afford full attention
- Very large graphs or dense graphs: all-to-alls attention intractable

Active research:

- Sparse attention (only nearby positions)
- Linear attention (kernel approximations)
- Hierarchical attention (clusters)3

Each trades expressiveness for efficiency.

Interpretability Challenge

Question: Can we understand what attention is doing?
Positive: Attention weights are interpretable (tell you what was weighted).
Challenge:

- Many layers (once you stack multiple attention blocks sequentially as in transformers) and heads $\rightarrow$ complex interactions
- Weights might not directly explain behavior
- Multiple heads do different things
- Depth (increases as you stack multiple attention blocks): information mixes in complex ways

Status: More interpretable than other deep models, but full understanding remains open.

Data Requirements

Observation: Attention models need more data than constrained architectures.
Why:

- No built-in inductive bias (unlike CNNs)
- Must learn from data what relationships matter
- More parameters to train

Trade-off:

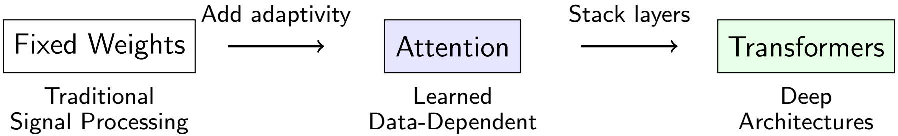
- High data regime: Transformers (the most popular attention based architecture) excel
- Low data regime: Constrained models are often better

Practical implication: For small datasets, consider simpler models or add inductive biases.

Key Takeaways

1. **Core Mechanism:**
   - Attention = adaptive weighted combination
   - Query-Key-Value separation enables flexibility
   - Softmax normalization ensures interpretable weights
2. **Why It Works:**
   - Adaptivity: learns what to attend to
   - Expressiveness: can implement many operations
   - Parallelizable: efficient on modern hardware
3. **Universality:**
   - Not task-specific; a general primitive
   - Appears in NLP, vision, graphs, control, ...
   - Same mechanism, different applications

From Signal Processing to Learning



Journey: Fixed filtering $\rightarrow$ Adaptive attention $\rightarrow$ Deep learning architectures

Mathematical Summary

The Attention Operation:

$$\boxed{\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V}$$

Components:

- QK^T: Compute all query-key similarities
- $1/\sqrt{d}$: Scale for stability
- softmax: Normalize to probability distribution
- Multiply by V: Weighted combination of values

Multi-Head: Parallel attention with different learned projections, then concatenate and project.
Self-Attention: Q, K, V all from same input via learned projections.

Practical Advice

When to use attention:

- Multiple information sources to combine
- Context-dependent relevance
- Sufficient data and compute available
- Need long-range dependencies

When to be cautious:

- Very limited data (consider simpler models)
- Extremely long sequences (need approximations)
- Strong domain knowledge available (consider inductive biases)
- Interpretability critical (attention helps but isn't sufficient)

Implementation: Modern frameworks (PyTorch, JAX) have efficient implementations. Start with standard architectures before customizing.

Resources for Further Study

Foundational papers:

- Vaswani et al. (2017): "Attention Is All You Need"
- Bahdanau et al. (2014): Original attention for NMT (Neural Machine Translation using attention in RNN-era)
- Dao et al. (2022): Flash Attention (efficient implementation)

Tutorials and courses:

- Stanford CS224N, MIT 6.S191
- 3Blue1Brown: Attention in transformers, step-by-step (Youtube Video)

Code implementations:

- PyTorch: `torch.nn.MultiheadAttention`
- Hugging Face Transformers library
- Annotated Transformer (Harvard NLP)

Conclusion

We've covered:

- Attention as adaptive weighted combination
- Mathematical formulation from first principles
- Scaling, multi-head extensions
- Why it works: adaptivity, expressiveness, efficiency
- Applications across domains
- Limitations

Attention is a powerful primitive for combining information.

Understanding it deeply enables you to apply it effectively and extend it creatively.

Questions?