

CS 489: Introduction to Natural Language Processing

Lecture 7: Transformers for Text Classification

Instructor: Yixuan (Tom) Wang

October 1, 2026



UNIVERSITY OF
WATERLOO

Outline

1. Text classification: from token embeddings to a sentence representation.
2. Self-attention: queries, keys, values, scaling, and multiple heads.
3. Position and the Transformer encoder block.
4. Extract features, pool them, and train a classifier.

Recap: Where We Are

So far:

- **Lectures 1–2:** What is NLP? Tokenization (BPE, WordPiece, Unigram).
- **Lecture 3:** Word embeddings: distributional semantics, Word2Vec.
- **Lecture 4:** Supervised learning: logistic regression, neural classifiers.
- **Lecture 5:** Statistical language models: N -grams, smoothing, perplexity.
- **Lecture 6:** Neural network foundations: MLPs, backprop, RNNs, LSTMs, GRUs.

Today: use a **Transformer encoder** as a feature extractor for text classification.

- Turn token embeddings into context-aware token features.
- Pool them into one sentence feature, then predict a label.
- Understand why self-attention replaces recurrence in the encoder.

Motivation: Context Matters for Classification

Task: Given a review, predict its sentiment (positive or negative).

“The plot was **good**, but the ending was **not good**.”

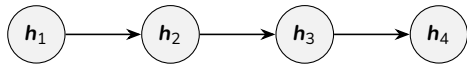
- A bag of word embeddings misses **order** and the context of “not.”
- An RNN builds context sequentially; distant words must pass through many steps.
- For one **sentence label**, we want features that capture interactions across *all* tokens before aggregating them.

Our pipeline: tokens → Transformer encoder → sentence feature → classifier.

Self-attention lets every token gather relevant context in parallel.

The Key Idea

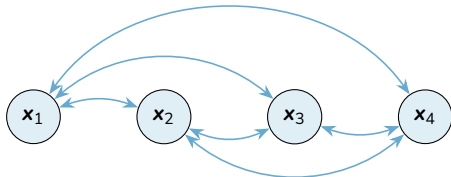
RNN: Sequential processing.



- Path length $h_1 \rightarrow h_4$: $O(n)$
- Computation: $O(n)$ sequential steps

Trade-off: Attention shortens the path between tokens and parallelizes positions, but its pairwise scores cost $O(n^2d)$ for sequence length n .

Self-attention: All-pairs interaction.



- Path length $x_1 \leftrightarrow x_4$: $O(1)$
- Computation: $O(1)$ sequential steps per layer

Attention: Intuition

Core idea: Given a **query**, find the most relevant **keys** in a set of key–value pairs, and return a weighted combination of the corresponding **values**.

Analogy: Looking up information in a database.

- **Query (q):** What am I looking for?
- **Key (k_i):** How relevant is entry i ?
- **Value (v_i):** What information does entry i contain?

Output: A weighted sum of values, where the weights are determined by query–key similarity:

$$\text{Attention}(\mathbf{q}, \{\mathbf{k}_i, \mathbf{v}_i\}) = \sum_i \alpha_i \mathbf{v}_i$$

where $\alpha_i = \frac{\exp(\text{score}(\mathbf{q}, \mathbf{k}_i))}{\sum_j \exp(\text{score}(\mathbf{q}, \mathbf{k}_j))}$ (softmax over scores).

Attention: A Worked Example

Consider the sentence: “The **cat** sat on the mat and it was **tired**.”

When processing the word “**tired**”, we want to know: what was tired?

- The **query** comes from “tired”: *Who is the subject?*
- Each word provides a **key**: *Am I a likely subject?*
- Each word also provides a **value**: its semantic content.

Attention weights (hypothetical):

Word	The	cat	sat	on	the	mat	and	it	was	tired
α_j	0.01	0.52	0.03	0.01	0.01	0.02	0.01	0.35	0.03	0.01

The model learns to attend to “cat” and “it”, the entities relevant to “tired.”

Dot-Product Attention

The simplest scoring function: **dot product** between query and key.

$$\text{score}(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i$$

Why dot product?

- Measures **similarity** between query and key vectors.
- If $\mathbf{q}$ and $\mathbf{k}_i$ point in similar directions $\rightarrow$ high score $\rightarrow$ high attention weight.
- **No extra parameters**: the learned projections (next slide) do the work.

Other scoring functions (historical):

- **Additive** (Bahdanau et al., 2015): $\text{score}(\mathbf{q}, \mathbf{k}) = \mathbf{v}^\top \tanh(\mathbf{W}_1 \mathbf{q} + \mathbf{W}_2 \mathbf{k})$
- **Multiplicative** (Luong et al., 2015): $\text{score}(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{W} \mathbf{k}$

Dot-product is fastest and works best in practice (with scaling).

Queries, Keys, and Values from Input

In **self-attention**, queries, keys, and values all come from the *same* sequence.

Given input embeddings $\mathbf{x}_1, \dots, \mathbf{x}_n \in \mathbb{R}^d$, stack them into a matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$.

Compute Q, K, V via **learned linear projections**:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q$$

$$\mathbf{W}_Q \in \mathbb{R}^{d \times d_k}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_K$$

$$\mathbf{W}_K \in \mathbb{R}^{d \times d_k}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_V$$

$$\mathbf{W}_V \in \mathbb{R}^{d \times d_v}$$

- Each token $\mathbf{x}_i$ simultaneously produces a **query** $\mathbf{q}_i$, a **key** $\mathbf{k}_i$, and a **value** $\mathbf{v}_i$.
- $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$ are **learned parameters**: three different projections of the same input.
- Typically $d_k = d_v = d/H$ where H is the number of attention heads (more later).

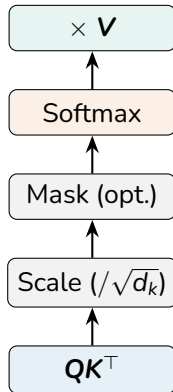
Scaled Dot-Product Attention

The full attention computation in matrix form:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\mathbf{Q}\mathbf{K}^\top / \sqrt{d_k}\right) \mathbf{V}$$

Step by step:

1. Compute scores: $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top \in \mathbb{R}^{n \times n}$
2. Scale: $\mathbf{S} \leftarrow \mathbf{S} / \sqrt{d_k}$
3. (Optional) Apply mask (for causal attention)
4. Softmax (row-wise): $\mathbf{A} = \text{softmax}(\mathbf{S})$
5. Weighted sum: output = $\mathbf{A}\mathbf{V} \in \mathbb{R}^{n \times d_v}$



Why Scale by $\sqrt{d_k}$?

Problem: When d_k is large, the dot products $\mathbf{q}^\top \mathbf{k}$ can grow large in magnitude.

Why? Assume entries of $\mathbf{q}$ and $\mathbf{k}$ are i.i.d. with mean 0 and variance 1. Then:

$$\mathbf{q}^\top \mathbf{k} = \sum_{j=1}^{d_k} q_j k_j$$

has mean 0 and variance d_k .

So $\text{std}(\mathbf{q}^\top \mathbf{k}) = \sqrt{d_k}$.

Effect on softmax: Large dot products $\rightarrow$ softmax saturates $\rightarrow$ gradients become tiny.

- With $d_k = 64$: values $\sim \mathcal{N}(0, 64)$, so scores can be ± 16 .
- Softmax of $[16, -16, 0, \dots] \approx [1, 0, 0, \dots]$, which is effectively hard attention.

Solution: Divide by $\sqrt{d_k}$ to restore unit variance: $\text{Var}(\mathbf{q}^\top \mathbf{k} / \sqrt{d_k}) = 1$.

Self-Attention: Worked Numerical Example

Tiny example: 3 tokens, $d_k = 2$.

$$Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \quad K = \begin{pmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Step 1: $QK^T = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 2 & 1 & 1 \end{pmatrix}$ **Step 2:** Scale: $QK^T / \sqrt{2} \approx \begin{pmatrix} 0.71 & 0 & 0.71 \\ 0.71 & 0.71 & 0 \\ 1.41 & 0.71 & 0.71 \end{pmatrix}$

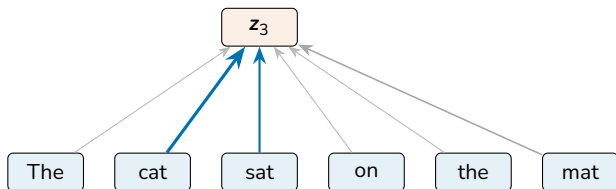
Step 3: Softmax (row-wise): $A \approx \begin{pmatrix} 0.40 & 0.20 & 0.40 \\ 0.40 & 0.40 & 0.20 \\ 0.50 & 0.25 & 0.25 \end{pmatrix}$

Step 4: Output = $AV \approx \begin{pmatrix} 0.80 & 0.60 \\ 0.60 & 0.60 \\ 0.75 & 0.50 \end{pmatrix}$ Each row is a weighted combination of all value vectors.

Self-Attention: Visualization

For an illustrative attention pattern, the output for a single query (token i) is:

$$\mathbf{z}_i = \sum_{j=1}^n \alpha_{ij} \mathbf{v}_j \quad \text{where} \quad \alpha_{ij} = \frac{\exp(\mathbf{q}_i^\top \mathbf{k}_j / \sqrt{d_k})}{\sum_{j'} \exp(\mathbf{q}_i^\top \mathbf{k}_{j'} / \sqrt{d_k})}$$



The output at “sat” combines all value vectors; thicker blue lines show larger **illustrative** weights. Actual attention weights depend on the input.

Why Multiple Heads?

A single attention head computes **one** set of attention weights.

But a word can be relevant to other words in **different ways**:

- **Syntactic:** in “The cat that **I saw** yesterday,” “saw” attends to “I” (subject).
- **Semantic:** “saw” attends to “cat” (object being seen).
- **Positional:** “saw” attends to nearby words for local context.

Solution: Run **multiple attention heads in parallel**, each with its own $W_Q^{(h)}$, $W_K^{(h)}$, $W_V^{(h)}$.

Each head can learn a **different attention pattern**.

This is called **multi-head attention** (Vaswani et al., 2017).

Multi-Head Attention: Formulation

Given H attention heads:

For each head $h = 1, \dots, H$:

$$\text{head}_h = \text{Attention}(\mathbf{X}\mathbf{W}_Q^{(h)}, \mathbf{X}\mathbf{W}_K^{(h)}, \mathbf{X}\mathbf{W}_V^{(h)})$$

where $\mathbf{W}_Q^{(h)}, \mathbf{W}_K^{(h)} \in \mathbb{R}^{d \times d_k}$ and $\mathbf{W}_V^{(h)} \in \mathbb{R}^{d \times d_v}$.

Concatenate and project:

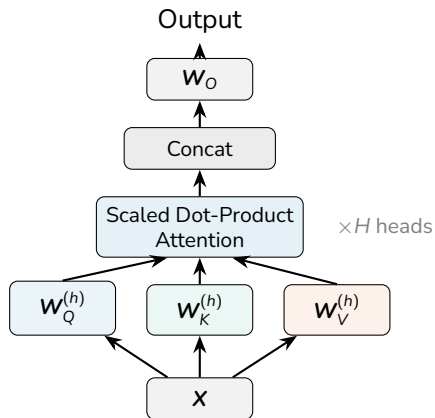
$$\text{MultiHead}(\mathbf{X}) = \text{Concat}(\text{head}_1, \dots, \text{head}_H) \mathbf{W}_O$$

where $\mathbf{W}_O \in \mathbb{R}^{Hd_v \times d}$.

Typical setting (Vaswani et al., 2017): $d = 512, H = 8, d_k = d_v = d/H = 64$.

Total computation is similar to single-head attention with full dimensionality, because $H \times d_k = d$.

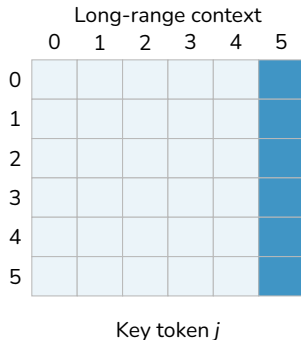
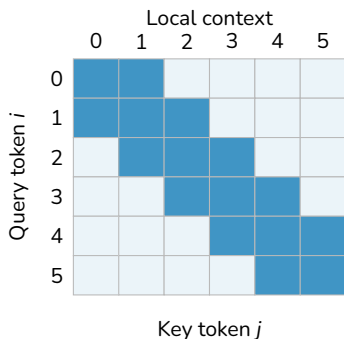
Multi-Head Attention: Diagram



Each head independently computes attention with different learned projections. Results are concatenated and linearly projected back to d dimensions.

Different Heads Can Gather Different Context

Each head has its own projections, so it *can* emphasize different token pairs.



Schematic patterns, **not** weights measured from a trained model. Actual heads can attend locally, track linguistic relations, or focus on special tokens (Clark et al., 2019).

The Position Problem

Self-attention is **permutation equivariant**: if you shuffle the input tokens, the outputs get shuffled the same way.

Proof sketch: attention weights depend only on $\mathbf{q}_i^\top \mathbf{k}_j$ (pairwise similarity), not on the *positions* i and j .

Problem: Without position features, pooling the outputs would give “dog bites man” and “man bites dog” the *same* sentence feature!

We need to **inject positional information** into the model.

Solution: Add a **positional encoding** $\mathbf{p}_i$ to each input embedding:

$$\mathbf{x}'_i = \mathbf{x}_i + \mathbf{p}_i$$

where $\mathbf{p}_i \in \mathbb{R}^d$ encodes the position i .

Sinusoidal Positional Encoding

Vaswani et al. (2017) proposed sinusoidal functions whose wavelengths increase with the dimension index:

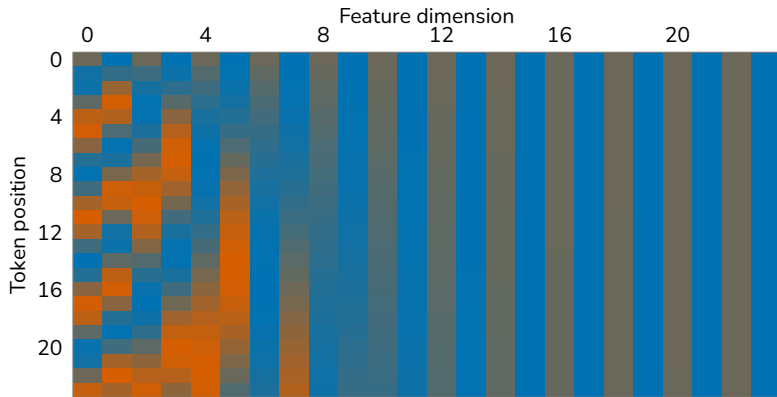
$$\text{PE}(i, 2j) = \sin\left(\frac{i}{10000^{2j/d}}\right), \quad \text{PE}(i, 2j+1) = \cos\left(\frac{i}{10000^{2j/d}}\right)$$

where i is the position and j is the dimension index.

Properties:

- Wavelengths: 2π to $10000 \cdot 2\pi$ (low to high dimensions).
- For any fixed offset k , $\text{PE}(i+k)$ can be expressed as a **linear function** of $\text{PE}(i)$.
 - This lets the model learn to attend to *relative* positions.
- **No learned parameters**, so this works for sequences longer than those seen in training.
- Different positions are guaranteed to have **distinct** encodings.

Sinusoidal Positional Encoding: Visualization



Left-hand dimensions vary rapidly across positions; right-hand dimensions vary slowly. Learned positional embeddings are another option (used in BERT and GPT-2).

Residual Connections

Problem: Deep networks can be hard to optimize when gradients shrink across layers.

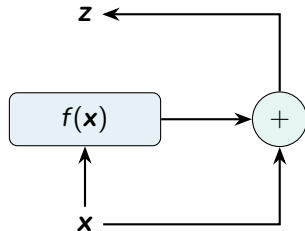
Solution (He et al., 2016): **Residual (skip) connections**.

$$\mathbf{z} = f(\mathbf{x}) + \mathbf{x}$$

Why this helps:

- Gradient flows through the **identity** path ($+\mathbf{x}$) even if f has small gradients.
- $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial f}{\partial \mathbf{x}} + \mathbf{I}$
- The $+\mathbf{I}$ term provides a direct gradient path.
- Same idea as the LSTM cell state “highway.”

Requirement: $f(\mathbf{x})$ must have the same dimension as $\mathbf{x}$.



Layer Normalization

Problem: Internal activations can have wildly different scales across layers and training steps, making optimization difficult.

Layer normalization (Ba et al., 2016) normalizes across the **feature dimension** for each token independently:

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

where $\mu = \frac{1}{d} \sum_{j=1}^d x_j$, $\sigma = \sqrt{\frac{1}{d} \sum_{j=1}^d (x_j - \mu)^2}$, $\gamma, \beta \in \mathbb{R}^d$ are learned.

Why Layer Norm?

- Normalizes each token independently, supporting variable lengths and any batch size.
- No dependence on other examples: consistent behavior at train and test time.

Position-Wise Feed-Forward Network

After attention, each token is processed by the **same** feed-forward network independently:

$$\text{FFN}(\mathbf{x}) = \mathbf{W}_2 \cdot g(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

- $\mathbf{W}_1 \in \mathbb{R}^{d_{\text{ff}} \times d}$, $\mathbf{W}_2 \in \mathbb{R}^{d \times d_{\text{ff}}}$ (typically $d_{\text{ff}} = 4d$).
- g : activation function (ReLU in original Transformer; GELU in later models).
- Applied to each position independently: same weights, different inputs.

Why is this needed for token features?

- Attention mixes information across tokens through weighted averages.
- The FFN adds **nonlinearity** and **per-token processing power**.
- Think of it as: attention **gathers** information; FFN **processes** it.

The FFN often contains **most** of the parameters (e.g., $d = 512$, $d_{\text{ff}} = 2048$: FFN has $2 \times 512 \times 2048 \approx 2\text{M}$ params per layer).

The Transformer Encoder Block

Putting it all together: one **Transformer encoder block** (or “layer”):

Post-norm (original Transformer):

$$\mathbf{Z} = \text{LN}(\mathbf{X} + \text{MHA}(\mathbf{X}))$$

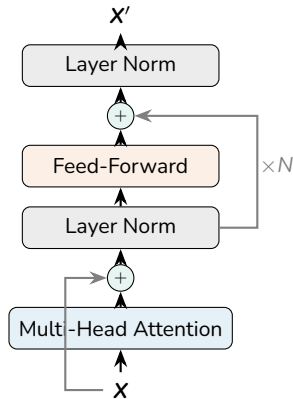
$$\mathbf{X}' = \text{LN}(\mathbf{Z} + \text{FFN}(\mathbf{Z}))$$

Pre-norm (used in GPT-2, modern LLMs):

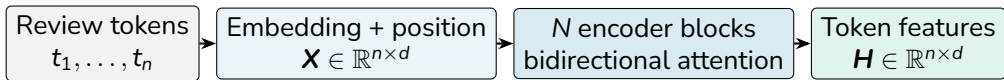
$$\mathbf{Z} = \mathbf{X} + \text{MHA}(\text{LN}(\mathbf{X}))$$

$$\mathbf{X}' = \mathbf{Z} + \text{FFN}(\text{LN}(\mathbf{Z}))$$

Pre-norm is more stable to train (gradient flows directly through the residual).



Transformer Encoder as a Feature Extractor



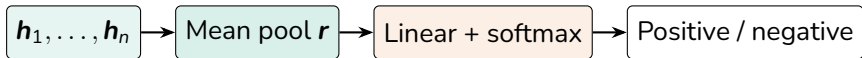
- Each row $\mathbf{h}_i$ summarizes token i in the context of the whole review.
- Unlike a language-model decoder, every token can attend in **both directions**.
- The encoder produces **features**, not class probabilities; a classification head comes next.

From Token Features to a Class Label

Pooling turns n contextual token vectors into one fixed-size review feature:

$$\mathbf{r} = \frac{1}{n} \sum_{i=1}^n \mathbf{h}_i \in \mathbb{R}^d \quad (\text{mean pooling over real tokens}).$$

Classification head: $\mathbf{p} = \text{softmax}(\mathbf{W}_c \mathbf{r} + \mathbf{b}_c)$, with $\mathbf{W}_c \in \mathbb{R}^{C \times d}$.



Alternatives include using a dedicated [CLS] token feature (as in BERT) or max pooling. For padded batches, average only the real token positions.

The Original Transformer: Encoder-Decoder

The original Transformer was designed for **sequence-to-sequence** tasks (e.g., machine translation):

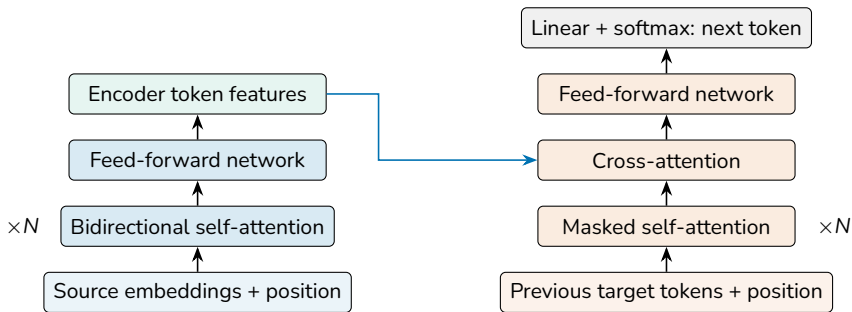
- **Encoder:** Processes the source sequence (e.g., English).
 - N stacked Transformer encoder blocks.
 - Bidirectional: each token attends to all tokens in the source.
- **Decoder:** Generates the target sequence (e.g., German), one token at a time.
 - N stacked Transformer decoder blocks.
 - **Masked** (causal) self-attention: token i can only attend to tokens $1, \dots, i$.
 - **Cross-attention:** each decoder token attends to all encoder outputs.

Configuration (the “base” model in Vaswani et al., 2017):

$N = 6, d = 512, H = 8, d_{ff} = 2048, d_k = d_v = 64$.

Total parameters: $\sim 65\text{M}$.

The Original Encoder-Decoder Architecture



Schematic adapted from [Vaswani et al. \(2017\)](#), Figure 1; residual connections and layer norms omitted for clarity. **For classification, we use only the encoder** and replace the decoder with pooling and a classifier.

Which Transformer for Which Task?

Architecture	Context per token	Example task
Encoder (BERT)	Both directions	Text classification, tagging
Decoder (GPT)	Past tokens only	Next-token generation
Encoder + decoder (T5)	Source + past target	Translation, summarization

Today: The bidirectional encoder makes a useful feature extractor for classification. Later lectures will cover pretrained encoders and language-model decoders.

Training a Transformer Text Classifier

For a review $\mathbf{x}$ with gold label y , the encoder and classification head produce $p(y \mid \mathbf{x})$.

Supervised objective: minimize cross-entropy across labeled examples:

$$\mathcal{L} = - \sum_{(\mathbf{x}, y) \in \mathcal{D}} \log p(y \mid \mathbf{x}).$$

- **Train from scratch:** update embeddings, encoder, and classification head using task labels.
- **Reuse a pretrained encoder:** freeze it as a feature extractor and train only the head, or fine-tune both together (Lecture 9).
- At inference time, predict $\hat{y} = \arg \max_c p(c \mid \mathbf{x})$.

Computational Complexity of Self-Attention

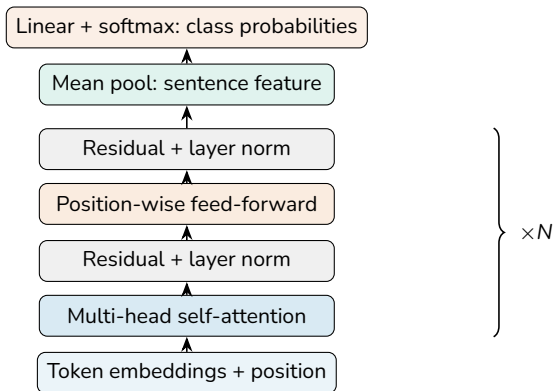
Cost per layer for sequence length n , model dimension d :

Component	Time Complexity	Space
Q, K, V projections	$O(n \cdot d^2)$	$O(n \cdot d)$
$QK^\top$ (attention scores)	$O(n^2 \cdot d)$	$O(n^2)$
Softmax + AV	$O(n^2 \cdot d)$	$O(n^2)$
FFN	$O(n \cdot d \cdot d_{ff})$	$O(n \cdot d_{ff})$
Total	$O(n^2 \cdot d + n \cdot d^2)$	$O(n^2 + n \cdot d)$

Key observation: The $O(n^2)$ term dominates for long sequences.

- $n = 512, d = 512$: $n^2 d \approx d^2 n$ (comparable costs).
- $n = 8,192, d = 512$: $n^2 d \gg d^2 n$ (attention dominates).
- Motivates efficient attention (Lecture 19): FlashAttention, linear attention, etc.

The Classification Pipeline: All Components



The **encoder** provides contextual token features; pooling and the head make one prediction per review.

Why Does the Transformer Work So Well?

1. **Parallelism:** All positions are processed simultaneously (unlike RNNs).
 - Parallelism substantially accelerates training relative to comparable RNNs.
 - Excellent GPU utilization: matrix multiplications are highly optimized.
2. **Direct long-range dependencies:** Any token can attend to any other in $O(1)$ layers.
 - No information bottleneck or vanishing gradients across positions.
3. **Flexible, data-dependent context:** Attention weights change with the input.
 - Unlike convolutions (fixed receptive field) or RNNs (decaying memory).
4. **Scalability:** Architecture scales well with data and compute.
 - Empirically: scaling up Transformers consistently improves performance (scaling laws, Lecture 10).

Impact: Transformers have become the **dominant architecture** in NLP (and increasingly in vision, biology, robotics, ...).

Readings

Required:

- Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). [Attention Is All You Need](#). *NeurIPS*.

Optional / further reading:

- Jurafsky & Martin, *SLP 3*, [Chapter 7: Transformers and Pretraining](#).
- Bahdanau, D., Cho, K., & Bengio, Y. (2015). [Neural Machine Translation by Jointly Learning to Align and Translate](#). *ICLR*. (Introduced attention for NMT.)
- Jay Alammur (2018). [The Illustrated Transformer](#). (Excellent visual walkthrough.)
- Clark, K., Khandelwal, U., Levy, O., & Manning, C.D. (2019). [What Does BERT Look At? An Analysis of BERT's Attention](#). *ACL Workshop BlackboxNLP*.

Next Lecture

Neural Language Models

- RNN language models.
- Transformer decoder (GPT-style).
- Training: teacher forcing, cross-entropy loss.
- Building a character-level LM from scratch.

[Reading: Holtzman et al., 2020]