Lecture 6 — Attention & the Transformer Architecture

The Architecture Behind Modern NLP

Tamara Grechanaya
Multimodal Computational Methods for Political Science

Summer Semester 2026

Contents

Learning objectives

By the end of this lecture, you should be able to:

1. Recall the three structural limitations of RNNs and LSTMs that motivated the Transformer.
2. Explain the intuition behind self-attention as "looking at what matters" for every position in a sequence.
3. Describe the query/key/value framework and walk through the scaled dot-product attention formula step by step.
4. Compute a small attention example by hand, and explain why the $\sqrt{d_k}$ scaling factor is necessary.
5. Explain why self-attention alone is order-blind and how positional encodings solve the problem.
6. Describe the role of multi-head attention and the kinds of relationships that different heads learn.
7. Assemble a full Transformer encoder block from its components: multi-head attention, residual connections, layer normalisation, and a position-wise feed-forward network.
8. Distinguish the three Transformer families — encoder-only, decoder-only, and encoder-decoder — and identify which is appropriate for which task.
9. Recognise why Transformers won: parallelism enables training at unprecedented scale.

1 Part I — From RNNs to attention

1.1 Recap: the three limits of LSTMs

LSTMs were the workhorse of NLP from 2015 to 2018. They solved the vanishing gradient problem and enabled real progress on translation, named entity recognition, and question answering. But as we saw at the end of Lecture 5, they have three fundamental limits.

No parallelism. The hidden state at step t depends on the hidden state at step $t-1$. Steps cannot be computed in parallel, and modern GPUs — whose entire architectural advantage is parallel computation — are largely wasted on RNNs.

Long-range dependencies remain hard. Information from word 1 must pass through 499 cell-state updates to reach word 500. LSTMs are far better than vanilla RNNs at preserving this signal, but they are not perfect, and very long documents still lose information across the sequence.

Information bottleneck. The cell state is a fixed-size vector. The entire history of the sequence must be compressed into this single vector, and longer sequences lose proportionally more information through this bottleneck.

What we want, instead, is an architecture that processes all positions in parallel, lets any position directly access any other position without a bottleneck, and captures long-range dependencies without exponentially decaying signal. The answer, as we will see, is *self-attention*.

1.2 "Attention is All You Need" (2017)

In June 2017, eight researchers at Google published a paper with a deliberately provocative title: *Attention Is All You Need* (Vaswani et al.). The bold claim was that you could throw away all the recurrence (RNNs, LSTMs) and convolution that had defined NLP architecture up to that point, and build a model out of nothing but attention layers. The result, they argued, would work better, train faster, and scale further.

They were right. The Transformer architecture introduced in that paper is now the foundation of essentially every modern language model: BERT (2018) and its descendants such as RoBERTa, which form the basis of most political science NLP today; the GPT family (2019 onwards), from GPT-2 through GPT-4 and ChatGPT; T5 and BART for translation and summarisation; and every modern large language model including LLaMA, Mistral, Claude, and Gemini.

Understanding the Transformer is the conceptual key to everything that has happened in NLP since 2017. This lecture builds it from the ground up.

1.3 The intuition: "look at what matters"

When humans read a sentence, we don't process word by word in isolation. Our brain unconsciously links each word to other relevant words. Consider a typical political sentence:

> *"The Prime Minister, despite vocal opposition from her own party, announced new climate measures."*

When you read "announced," your attention drifts back to "Prime Minister" to ask *who* announced. When you read "her," your brain points back to "Prime Minister" again to identify whose party is being discussed. When you read "measures," your attention links forward to "climate" to ask *what kind* of measures.

Attention is this mechanism, made mathematical. For each word in the input, the model does three things. First, it asks: *which other words in this sentence are relevant to me?* Second, it computes a weight for each other word — high for relevant, low for irrelevant. Third, it builds its own new representation as a weighted combination of all the other words.

Crucially, every word can attend to every other word *directly*, in a single step. There is no sequential walk-through and no fixed-size bottleneck. To make this concrete, think about computing a new representation for the word "passed" in the sentence "The bill passed despite opposition." The output for "passed" is a weighted sum of all five input positions. Heavy weights would fall on *bill* and *opposition* (semantically related to the action of passing), lighter weights on *passed* itself, and very light weights on function words like *the* and *despite*. The model learns these weights from data, for each pair of positions.

2 Part II — Self-attention mechanics

2.1 The Q, K, V framework

For each input position, attention uses three derived vectors: a query, a key, and a value.

Query (Q): what am I looking for? The query represents this position's "question" to the rest of the sequence.

Key (K): what do I offer? The key represents what information this position makes available to others.

Value (V): what will I give? The value is the actual content this position contributes, if attended to.

The library analogy is helpful. You walk into a library (the sequence) with a question (your query). Each book has a label on the spine (its key) and content inside (its value). You compare your question to every label; the labels that match best get the most attention; and you read the matched books, weighted by how well their labels matched.

Each of Q, K, V is a vector. They are all derived from the same input by separate *learned* linear transformations:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \qquad \mathbf{K} = \mathbf{X}\mathbf{W}_K, \qquad \mathbf{V} = \mathbf{X}\mathbf{W}_V \tag{1}$$

The weight matrices $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ are what the model learns from data.

2.2 Computing attention: the steps

For one position i attending over the sequence, attention proceeds in four steps. First, *score* the query at position i against the keys at every position j using a dot product:

$$s_{ij} = \mathbf{q}_i \cdot \mathbf{k}_j \tag{2}$$

A big positive score means the key matches the query well; a big negative score means a poor match. Second, *scale* the scores by $\sqrt{d_k}$, where d_k is the dimensionality of the keys (we will explain why shortly):

$$\tilde{s}_{ij} = \frac{s_{ij}}{\sqrt{d_k}} \tag{3}$$

Third, apply *softmax* to turn the scores into a probability distribution:

$$\alpha_{ij} = \frac{e^{\tilde{s}_{ij}}}{\sum_{j'} e^{\tilde{s}_{ij'}}} \tag{4}$$

The α_{ij} are the **attention weights**: how much position i attends to position j. They sum to 1 across j. Fourth and finally, take a *weighted sum* of values:

$$\text{output}_i = \sum_j \alpha_{ij}\, \mathbf{v}_j \tag{5}$$

2.3 The whole thing in one formula

Putting it all together in matrix form, attention is:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V} \tag{6}$$

Breaking it down: the product $\mathbf{Q}\mathbf{K}^\top$ computes all pairwise dot products between queries and keys at once — for n positions, this is an $n \times n$ matrix of scores. Division by $\sqrt{d_k}$ is the scaling step. The softmax is applied row-wise so each row becomes a probability distribution. Multiplication by $\mathbf{V}$ produces a weighted sum of values for each position.

This single formula — "scaled dot-product attention" — is the heart of every modern language model. It is computed in parallel for all positions simultaneously, with no sequential walk-through.

2.4 Worked example: the setup

Let me walk through a concrete example with toy numbers. Consider the three-token sentence "The bill passed."

Suppose that after the linear projections, the three tokens have the following 2-dimensional query, key, and value vectors:

Token	Query $\mathbf{q}$		Key $\mathbf{k}$		Value $\mathbf{v}$	
The	1.0	0.0	0.2	0.0	0.1	0.2
bill	0.0	1.0	0.1	0.9	0.7	0.3
passed	0.5	0.5	0.7	0.5	0.4	0.5

The goal is to compute the attention output for "passed" (position 3), whose query is $\mathbf{q}_3 = [0.5, 0.5]$. We will score this query against each key, scale by $\sqrt{d_k} = \sqrt{2} \approx 1.414$, apply softmax to get attention weights, and take a weighted sum of values.

2.5 Worked example: the math

Step 1: Dot product scores.

$$
\begin{aligned}
s_{3,1} &= \mathbf{q}_3 \cdot \mathbf{k}_{\text{The}} = 0.5 \times 0.2 + 0.5 \times 0.0 = 0.10 \\
s_{3,2} &= \mathbf{q}_3 \cdot \mathbf{k}_{\text{bill}} = 0.5 \times 0.1 + 0.5 \times 0.9 = 0.50 \\
s_{3,3} &= \mathbf{q}_3 \cdot \mathbf{k}_{\text{passed}} = 0.5 \times 0.7 + 0.5 \times 0.5 = 0.60
\end{aligned}
$$

Notice that the scores already differ. "Passed" matches its own key most strongly (0.60), the key of "bill" moderately (0.50), and the key of "the" only weakly (0.10).

Step 2: Scale by $\sqrt{d_k} = \sqrt{2} \approx 1.414$.

$$\tilde{s} = [0.10/1.414,\ 0.50/1.414,\ 0.60/1.414] \approx [0.071,\ 0.354,\ 0.424]$$

Step 3: Softmax. Compute the exponentials and normalize:

$$
\begin{aligned}
e^{0.071} &\approx 1.073 \\
e^{0.354} &\approx 1.424 \\
e^{0.424} &\approx 1.528
\end{aligned}
$$

The sum is $1.073 + 1.424 + 1.528 \approx 4.025$. Dividing each exponential by this sum gives the attention weights:

$$\boldsymbol{\alpha}_3 \approx [0.267,\ 0.354,\ 0.379]$$

(They sum to 1.0 as required for a probability distribution.)

Step 4: Weighted sum of values.

$$
\begin{aligned}
\text{output}_3 &= 0.267 \times [0.1, 0.2] + 0.354 \times [0.7, 0.3] + 0.379 \times [0.4, 0.5] \\
&= [0.0267, 0.0534] + [0.2478, 0.1062] + [0.1516, 0.1895] \\
&\approx [0.426,\ 0.349]
\end{aligned}
$$

This is the attention output for "passed." The key feature to notice is that the attention weights are **not uniform**: "passed" attends most strongly to itself (0.379) and to "bill" (0.354), with

much lower weight on "the" (0.267). The output vector $[0.426, 0.349]$ is therefore weighted toward the value vectors of "bill" and "passed."

In a real trained BERT model, the $\mathbf{W}_Q$, $\mathbf{W}_K$, $\mathbf{W}_V$ matrices would be learned from billions of words such that semantically important relationships (subject–verb, modifier–noun, coreference) produce much sharper attention patterns than this toy example. The mechanism is exactly what we just computed; only the scale of the numbers and the sharpness of the weights differ.

2.6 Why divide by $\sqrt{d_k}$?

The scaling factor $\sqrt{d_k}$ is not arbitrary. It exists for one reason: to keep softmax well-behaved. In high-dimensional spaces (say $d_k = 64$), random dot products tend to have large magnitudes. If scores are very large in absolute value, softmax becomes "peaky" — one position gets attention weight close to 1 and all others get close to 0.

The consequence is that gradients flowing through softmax become tiny: most weights are nearly 0 or 1, where the derivative of softmax is flat. Training stalls.

The fix is to divide by $\sqrt{d_k}$. The variance of the dot product of two random vectors of dimension d_k is proportional to d_k; dividing by $\sqrt{d_k}$ keeps the standard deviation roughly constant regardless of dimension. You do not need to remember the derivation. The practical takeaway is that the scaling is a numerical-stability trick, and without it, training is unstable in higher dimensions.

2.7 Attention is permutation-equivariant

An important property of self-attention: if you reorder the input tokens, the outputs reorder correspondingly — but the *content* of each token's representation doesn't change. In other words, self-attention doesn't know which word came first.

Consider the pair "*The opposition supported the bill*" vs. "*The bill supported the opposition.*" These have the same words. Self-attention, on its own, produces the same set of token representations for both sentences, just in a different order. The meaning is completely different, but attention cannot tell them apart.

This is a fatal problem if left unsolved. Almost everything in language depends on word order: subject vs. object, negation scope, syntactic structure. The fix is *positional encodings*, which we discuss in Part IV.

3 Part III — Multi-head attention

3.1 One head is not enough

The problem with single-head attention is that different kinds of relationships in language need different attention patterns. Return to the sentence "*The Prime Minister, despite vocal opposition, announced climate measures.*" For the word "announced," a useful model might want to attend to subject relations ("Prime Minister" — who announced?), object relations ("measures" — announced what?), contrast and concession ("despite" and "opposition"), and topical/semantic field ("climate"). A single set of attention weights cannot capture all four simultaneously.

The solution is multiple attention heads running in parallel. Each head learns its own Q, K, V projections and attends differently; their outputs are concatenated and projected back into the model dimension.

3.2 The architecture

Multi-head attention takes an input matrix $\mathbf{X}$ and applies h separate attention computations in parallel. Each head i has its own learned weight matrices $\mathbf{W}_Q^{(i)}, \mathbf{W}_K^{(i)}, \mathbf{W}_V^{(i)}$, produces its own queries, keys, and values, and runs its own scaled dot-product attention. Each head operates in a smaller dimension $d_k = d_{\text{model}}/h$, so the total computation is comparable to a single attention at full dimension.

The outputs of the h heads are then concatenated along the feature axis and passed through one final linear projection $\mathbf{W}_O$ that mixes them back into the model dimension. BERT-base uses 12 heads of dimension 64 each ($12 \times 64 = 768$); BERT-large uses 16 heads of 64 ($16 \times 64 = 1024$).

3.3 What different heads learn

An empirical finding from work probing pre-trained Transformers is that different heads specialise in different relationships, often interpretable ones. Clark et al. (2019), in "What Does BERT Look At?", identified several recurring patterns. Some heads attend to the next or previous word, capturing simple positional patterns. Some attend to syntactic dependencies — for example, direct objects attending to their verbs, or possessive pronouns attending to their referents. Some heads attend predominantly to delimiters like `[SEP]` and `[CLS]`, behaving as a kind of "default" position when no other word is informative. Some attend to coreference, with pronouns pointing back to the entities they refer to. And some heads remain hard to interpret: they encode something useful, but not in a way that maps neatly onto human linguistic categories.

For political science research this matters because you can visualise attention in pre-trained models to study how they process political text. This is a growing research area — probing model biases, comparing how a model processes left-wing versus right-wing rhetoric, and so on.

4 Part IV — The Transformer block

4.1 Positional encodings: the missing piece

Because self-attention is order-blind, we must *inject* positional information into the inputs before attention is applied. The idea is to add a position-dependent vector to each word's embedding. So if $\mathbf{e}_{\text{bill}}$ is the word embedding for "bill" and $\mathbf{p}_2$ is the vector for position 2, the input to the first Transformer layer at that position is simply $\mathbf{e}_{\text{bill}} + \mathbf{p}_2$.

Two main flavours of positional encoding are used in practice.

Sinusoidal encodings (the original Transformer) are fixed functions using sines and cosines at different frequencies. They are mathematically elegant and generalise naturally to sequence lengths the model did not see during training.

Learned encodings (BERT and most modern models) are simply another learned vector per position. Simpler in implementation, but limited to the maximum length seen during training.

You do not need to memorise the sinusoidal formulas. The key insight is that position is encoded as a vector and added to the word embedding. After this step, the model knows where each token is in the sequence.

4.2 Residual connections and layer normalisation

Stacking many attention layers is hard. Two tricks make training deep Transformers possible.

Residual (skip) connections add the input of a sub-layer to its output:

$$\mathbf{y} = \mathbf{x} + \text{SubLayer}(\mathbf{x}) \tag{7}$$

Gradients can flow "around" the sub-layer through the addition. Even if the sub-layer makes things worse, the model can fall back to the identity. This idea, originally from ResNets (2015), is essential for training networks dozens or hundreds of layers deep.

Layer normalisation re-centers and rescales each token's vector so that it has mean 0 and standard deviation 1 (with learned scale and shift parameters). This keeps the magnitudes of activations stable across layers and prevents them from blowing up or vanishing during training.

The combination "add and norm" appears around every sub-layer in the Transformer block. These are engineering details, not deep conceptual ideas. But without them, the model would not train.

4.3 The feed-forward sub-layer

After attention (and the add-and-norm step), each Transformer block applies a small feed-forward network independently to each token position:

$$\text{FFN}(\mathbf{x}) = \text{ReLU}(\mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\,\mathbf{W}_2 + \mathbf{b}_2 \tag{8}$$

Two things are worth noting. First, the same FFN is applied to every position separately — same weights, different inputs. Second, the hidden dimension is wider than the model dimension, typically by a factor of four (e.g., $768 \to 3072 \to 768$ in BERT-base).

What does it do? Mathematically, it applies a non-linear transformation to each token's representation after the linear attention step. Intuitively, it is the "processing" each token undergoes after it has gathered context from others. A useful piece of trivia: in a typical Transformer, most of the parameters live in the FFNs, not in the attention. The attention layers get the conceptual headlines, but the FFN layers carry most of the storage.

4.4 The full Transformer encoder block

A single Transformer encoder block puts all of these pieces together: multi-head self-attention, then add-and-norm, then a position-wise feed-forward network, then another add-and-norm. The input flows into the attention sub-layer, the residual carries the original input around it, the result is normalised, the feed-forward layer processes it, another residual and norm step follows, and the output is passed to the next block.

A full encoder, such as BERT, stacks 12 or 24 such blocks on top of each other. Each block transforms representations: earlier blocks pick up local and surface-level patterns, later blocks pick up abstract and semantic structure.

4.5 Three Transformer families

The original Transformer (Vaswani et al. 2017) had an encoder and a decoder for machine translation. Today, three families dominate.

Encoder-only models, such as BERT and RoBERTa, are designed for *understanding* text. They use bidirectional attention: every token sees every other token. They are well suited to classification, named entity recognition, similarity, and sentence-pair tasks. This is the family we will fine-tune in Lecture 7.

Decoder-only models, such as the GPT family and LLaMA, are designed for *generating* text. They use causal attention: each token can only attend to itself and to past tokens, never to the future. They are well suited to text generation, dialogue, and code completion — the architecture behind ChatGPT, Claude, and similar systems.

Encoder-decoder models, such as T5 and BART, are designed for text-to-text mappings. The encoder reads the input; the decoder generates the output; cross-attention links them. They are well suited to translation, summarisation, and structured question answering.

For political science classification tasks, encoder-only models from the BERT family are usually the right choice. They are the most cost-effective, the easiest to fine-tune on limited compute, and the best-validated in the social science literature.

5 Part V — Why this matters

5.1 Transformers vs. LSTMs

It is worth laying out the comparison to LSTMs explicitly. LSTMs process the sequence sequentially, one step at a time; Transformers process all positions in parallel. LSTMs handle long-range dependencies through gated memory that decays over many steps; Transformers connect any two positions directly in a single layer. LSTMs compress the entire history into a fixed-size cell state; Transformers have no bottleneck and store full pairwise attention weights. LSTMs train slowly on GPUs; Transformers train fast because of parallelism. LSTMs have a built-in inductive bias for sequences via recurrence; Transformers must learn position from the data via positional encodings. And, decisively, LSTMs are mostly historical today, while Transformers are everywhere.

There is one notable downside: attention computes $n \times n$ pairwise scores, so its memory cost grows quadratically with sequence length. For a 10,000-word document, that is 100 million pairs — enough to exhaust the memory of a normal GPU. "Long-context" Transformers such as Longformer and BigBird use sparse attention patterns to scale to long inputs. This is an active research area.

5.2 Why Transformers won: scale

The Transformer's parallel structure means it can train on much more data than RNNs ever could. This is the single biggest reason it won. A brief scaling timeline illustrates the point: BERT-base in 2018 had 110 million parameters and was trained on roughly 3 billion words; GPT-2 in 2019 had 1.5 billion parameters and 40 GB of web text; GPT-3 in 2020 had 175 billion parameters and around 500 billion tokens; and large frontier models today have hundreds of billions to trillions of parameters trained on trillions of tokens.

The "scaling laws" lesson is that model performance keeps improving as you make the model, the data, and the compute bigger. So far, no plateau is in sight.

The implications for political science are practical. Cutting-edge models are out of reach to train, but they are not out of reach to *use*. Pre-trained models can be downloaded and applied to your data with minimal compute.

5.3 Political science applications

Several research directions in political science are being changed by transformer-based methods.

High-quality classification with little data. Fine-tuning a pre-trained model on a few hundred labeled examples often beats logistic regression with thousands. This is especially useful

for nuanced tasks like stance, framing, and irony.

Multilingual analysis. Multilingual BERT and XLM-RoBERTa make cross-country political text comparable without manual translation, opening up genuinely comparative research designs.

Zero-shot and few-shot classification. Large language models can classify political text from natural-language prompts with little or no training data. The results are strong but unreliable, and always require validation against human-coded benchmarks.

Probing political biases in models. Models reproduce biases in their training data. Studying which biases they have, and how strongly, is itself a research question of growing interest.

Contextual embeddings. Unlike word2vec or GloVe, BERT gives a different vector for every occurrence of a word. "Sovereignty" in a UKIP speech has a different vector than "sovereignty" in a Labour speech — and the difference is measurable. This enables new ways of studying polarisation and discourse change.

Key takeaways

1. **Self-attention** lets every position in a sequence directly attend to every other position, fixing the three weaknesses of RNNs: lack of parallelism, the fixed-size bottleneck, and exponential decay of long-range signal.
2. **The Q, K, V framework** is the heart of attention. Each position projects to a query, a key, and a value; queries match keys to determine how much to attend to each value.
3. **Scaled dot-product attention** is the single most important equation in modern NLP: $\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\mathbf{Q}\mathbf{K}^\top / \sqrt{d_k})\,\mathbf{V}$.
4. **Multi-head attention** runs several attentions in parallel, letting the model capture different relationships (syntactic, semantic, positional, coreferential) at once.
5. **Positional encodings** are essential. Without them, attention is order-blind and cannot distinguish "the bill passed" from "passed the bill."
6. **A Transformer block** combines multi-head attention, add-and-norm, a position-wise feed-forward network, and another add-and-norm — stacked many times to form the full model.
7. **Three families of Transformers** cover most modern NLP: encoder-only (BERT, for understanding), decoder-only (GPT, for generation), and encoder-decoder (T5, for text-to-text).
8. **Why they won: parallelism enabled training at unprecedented scale.** Scale wins.

Required reading

- Jurafsky, D. & Martin, J.H. (2025). *Speech and Language Processing*, Chapter 8: Transformers. `https://web.stanford.edu/~jurafsky/slp3/8.pdf`

Recommended reading

- Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). "Attention Is All You Need." *NeurIPS 2017*. arXiv:1706.03762.

- Alammar, J. (2018). "The Illustrated Transformer." `http://jalammar.github.io/illustrated-transformer/` (An intuitive visual explanation of the Transformer.)

Other related reading

- Clark, K., Khandelwal, U., Levy, O., & Manning, C.D. (2019). "What Does BERT Look At? An Analysis of BERT's Attention." *BlackboxNLP Workshop.*
- Rogers, A., Kovaleva, O., & Rumshisky, A. (2020). "A Primer in BERTology: What We Know About How BERT Works." *Transactions of the Association for Computational Linguistics*, 8, 842–866.

Home assignment

The home assignment for this lecture is to explore attention patterns in a pre-trained model on political text. Using the practical code as a starting point, pick three sentences from UK parliamentary speeches (or your own political corpus); each should have an interesting feature such as negation, a pronoun, a coordinated structure, or a long-range dependency. For each sentence, identify at least one attention head whose pattern you can interpret, and save the heatmap. For each interpretable head, write two or three sentences explaining what relationship it seems to encode and how you tested this. Then compare attention patterns in an early layer (e.g., layer 2) with a late layer (e.g., layer 10) for the same sentence, and describe what changes.

Looking ahead

In **Lecture 7**, we put the Transformer to work. We move from understanding the architecture to *fine-tuning* a pre-trained BERT model on a political classification task. We will discuss transfer learning conceptually — why pre-training plus fine-tuning works so well — and then walk through the practical pipeline: tokenisation with subword units, the fine-tuning loop, hyperparameter choices that matter, and the evaluation discipline carried over from Lecture 2. By the end of Lecture 7, you will have fine-tuned a BERT classifier on UK Commons speeches and compared it head-to-head against every method we have built across the course. This is the practical culmination of the seminar.