Understanding Sequential Data with Attention and Transformers

Seth J. Chandler, with help from AI

September 7, 2025

1 Introduction: The Challenge of Sequential Data

The preceding chapter demonstrated how a neural network can learn to map a single, static input to a corresponding output. Through the elegant process of backpropagation, the network adjusts its internal weights and biases to minimize prediction error, effectively learning a function from isolated data points. However, many of the most complex problems, particularly in fields like law and finance, do not involve isolated data. Instead, they involve sequences, where order, context, and long-range relationships are paramount.

Consider the words in a legal contract, a series of financial transactions, or the chain of events in a case history. The meaning of any single element is profoundly shaped by its relationship to other elements in the sequence. A feed-forward network, as described previously, is fundamentally unsuited for this task. It possesses no memory of past inputs and treats every data point as an independent event. To understand sequences, we require an architecture that can not only process elements in order but can also grasp the intricate web of dependencies that connect them, regardless of the distance between them. This chapter introduces the mechanisms that were developed to meet this challenge, culminating in the Transformer architecture that powers modern artificial intelligence.

2 Early Approaches and Their Limitations: The Memory Bottleneck

The first major attempt to process sequential data involved a class of models called Recurrent Neural Networks (RNNs). Conceptually, an RNN processes a sequence one element at a time, maintaining an internal "state" or "memory" that acts as a running summary of the

information it has seen so far. At each step, the model combines the current input with its memory from the previous step to produce an output and update its memory for the next step.

While revolutionary, this sequential approach created two fundamental challenges that limited its effectiveness on long sequences:

1. **The Information Bottleneck.** The entire meaning of a preceding sequence, no matter how long, had to be compressed into a single, fixed-size state vector. This is analogous to trying to summarize the first 50 pages of a contract into a single sentence before reading page 51. Inevitably, crucial details are lost, and the model's memory of distant events becomes fuzzy.

2. **The Vanishing Gradient Problem.** As established in the previous chapter, learning is driven by the gradient of the loss function. In a long sequence, the error signal calculated at the end must be propagated backward through every single step. With each step back, the gradient can shrink exponentially, often becoming so small that it has no practical effect on the parameters at the beginning of the sequence. This makes it exceedingly difficult for the model to learn connections between elements that are far apart—the very problem of *long-range dependencies*.

3 The Attention Mechanism: A Paradigm Shift

The solution to the memory bottleneck was not to build a better memory, but to remove the need for one. The attention mechanism allows a model to break free from the rigid, one-at-a-time processing of RNNs. It provides a way for the model, when considering any single element in a sequence, to directly look at and weigh the importance of *every other element* in the entire sequence.

The core insight is to create a direct pathway between any two points in the sequence, allowing the gradient to flow without having to traverse all the intermediate steps. This is like a lawyer who, upon encountering the term "Primary Beneficiary" on page 50 of a contract, can instantly refer back to its precise definition on page 1, rather than relying on a fading memory of it.

4 The Mathematics of Self-Attention

Let $X \in \mathbb{R}^{n \times d_{\text{model}}}$ hold the token embeddings (one row per token). The model learns three projections:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V,$$

with $W^Q, W^K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ and $W^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$. *Scaled dot-product attention* is

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

The division by $\sqrt{d_k}$ keeps the variance of scores under control so softmax remains in a gradient-friendly regime.

Shapes at a glance.

$$\begin{aligned}
&X : \ n \times d_{\text{model}}, \quad W^Q, W^K : \ d_{\text{model}} \times d_k, \quad W^V : \ d_{\text{model}} \times d_v, \\
&Q, K : \ n \times d_k, \quad V : \ n \times d_v, \quad S \equiv QK^\top : \ n \times n, \\
&A = \text{softmax}(S/\sqrt{d_k}) : \ n \times n, \quad Z = AV : \ n \times d_v.
\end{aligned}$$

Masking and architecture families (clear and concrete).

- **Encoder-only** (e.g., BERT): bidirectional self-attention (no causal mask). Each row of A may place weight on any column.
- **Decoder-only** (e.g., GPT): *causal* self-attention. Row i of A may only place weight on columns $\leq i$ (no peeking at future tokens). Practically: entries with $j > i$ in the score matrix are set to $-\infty$ (or a large negative) *before* softmax, forcing the corresponding attention weights to 0.
- **Encoder–decoder** (e.g., T5): encoder uses bidirectional self-attention; decoder uses causal self-attention plus *cross-attention* over the encoder outputs.

5 Worked Example A: Two Tokens "Be Brief" (unmasked)

We compute attention step-by-step for a minimal two-token sequence, focusing on arithmetic transparency.

5.1 Setup

- Embeddings ($X \in \mathbb{R}^{2\times 2}$):

$$X = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \quad \begin{matrix} \leftarrow \text{Be} \\ \leftarrow \text{Brief} \end{matrix}$$

- Projections:

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

- Dimensions: $d_k = d_v = 2$ so $\sqrt{d_k} = \sqrt{2} \approx 1.414$.

5.2 Forward pass

1. $Q = XW^Q = X, \quad K = XW^K = X, \quad V = XW^V = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$.
2. Scores $S = QK^\top = XX^\top = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}$.
3. Scale: $S/\sqrt{2} \approx \begin{pmatrix} 1.414 & 0.707 \\ 0.707 & 0.707 \end{pmatrix}$.
4. Row-wise softmax:
$$A \approx \begin{pmatrix} 0.67 & 0.33 \\ 0.50 & 0.50 \end{pmatrix}.$$
5. Output $Z = AV = \begin{pmatrix} 0.67 & 1.00 \\ 0.50 & 1.00 \end{pmatrix}$. The contextualized vector for "Be" is the first row of Z.

How a causal mask would alter this (two tokens). For a decoder-only setup, token 1 may not attend to token 2. Replace the first row of A by $[1, 0]$. Then $Z_{1\cdot} = [1, 1]$ because $[1, 0] \cdot V = [1, 1]$. The formulas are unchanged; only the masked entries are forced to zero *before* softmax.

6 Worked Example B: Three Tokens "not guilty plea" (hand-computable)

This example is designed for calculator-only work. We pick tiny numbers so one row of scores is $[0, 1, 2]$, giving easy softmax weights with an intuitive interpretation.

6.1 Setup and goal

We examine the attention *from the first token* "not". To keep arithmetic simple, we directly specify the scaled scores for the first row (as if they resulted from $QK^\top/\sqrt{d_k}$):

$$\underbrace{\big[S_{1,1},\ S_{1,2},\ S_{1,3}\big]}_{\text{row 1 of } S/\sqrt{d_k}} = \begin{bmatrix} 0, & 1, & 2 \end{bmatrix} \quad \begin{matrix} \leftarrow \text{(to "not")} \\ \leftarrow \text{(to "guilty")} \\ \leftarrow \text{(to "plea")} \end{matrix}$$

Interpretation: for the first token as query, match to the three keys yields logits $0, 1, 2$.

6.2 Row-wise softmax for the first token

Let $e^0 = 1,\ e^1 \approx 2.72,\ e^2 \approx 7.39$. The sum is $1 + 2.72 + 7.39 \approx 11.11$. Hence the *attention weights* from "not" are

$$A_{1\cdot} \ = \ \big[\,0.09,\ 0.24,\ 0.67\,\big] \quad \text{(to two decimals)}.$$

So, in context, the model focuses mostly on the third token ("plea") when updating "not".

6.3 Producing the contextual vector for "not"

Choose a small value matrix $V \in \mathbb{R}^{3\times 2}$ to keep multiplication easy, e.g.

$$V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

Then the contextualized representation of "not" is the weighted average of V's rows with weights $A_{1\cdot}$:

$$Z_{1\cdot} = A_{1\cdot}V = 0.09\begin{pmatrix} 1 & 0 \end{pmatrix} + 0.24\begin{pmatrix} 0 & 1 \end{pmatrix} + 0.67\begin{pmatrix} 1 & 1 \end{pmatrix} = \begin{pmatrix} 0.76 & 0.91 \end{pmatrix} \ \text{(approx)}.$$

Students can compute this with a basic calculator.

Causal mask demonstration (three tokens). In a decoder-only LLM, the first token may attend to positions ≤ 1 only. For row 1, scores at columns 2 and 3 are masked (set to $-\infty$) before softmax. The resulting weights are $[1, 0, 0]$ and $Z_{1\cdot} = V_{1\cdot} = [1, 0]$. For row 2, only columns ≤ 2 are allowed; for row 3, all columns ≤ 3 are allowed. This is the entire substance of masking.

7 From Attention to Transformers: Building a Complete Model

7.1 Multi-Head Attention

Instead of one attention calculation, Transformers use h heads: split the model width into h subspaces, compute attention in each, then concatenate and project. Different heads capture different relations (scope, negation, co-reference, etc.).

7.2 Essential Architectural Components

- **Positional information (brief).** Attention is content-based and order-agnostic. Models inject a small position signal (fixed sinusoids, learned embeddings, or rotary variants) so position i differs from j. We do not dwell on formulas here.
- **Feed-Forward Networks (FFN).** A two-layer MLP applied independently to each token: $\mathrm{FFN}(x) = \phi(xW_1 + b_1)W_2 + b_2$.
- **Residual connections and LayerNorm.** Each sublayer (MHA, FFN) uses residual add and normalization to stabilize deep training.

7.3 From Z to next-token probabilities

Stack several Transformer blocks. After the final block, a linear map $W_O \in \mathbb{R}^{d_{\text{model}} \times |V|}$ produces *logits* over the vocabulary at each position; softmax converts logits to a probability distribution. Training uses cross-entropy for the correct next token, exactly as previewed at the end of a prior chapter.

Logits, Softmax, and Cross-Entropy in Pretraining

At the final layer of a language model, each token position produces a vector of *logits*—unconstrained real numbers, one per vocabulary item. These "logits" carry no immediate probabilistic interpretation. They are just numbers. To convert them into a distribution over possible next tokens, the model applies the *softmax* function:

$$p(y = j \mid x) = \frac{e^{z_j}}{\sum_{k=1}^{|V|} e^{z_k}},$$

where z_j is the logit for token j and $|V|$ is the vocabulary size. Softmax ensures all probabilities are nonnegative and sum to 1.

Training then compares these predicted probabilities to the actual next token, represented as a *one-hot* vector. For instance, if the true token is "contract," the target vector has a 1 in the "contract" entry and 0 elsewhere. The standard loss is *cross-entropy*, which for one-hot targets reduces to the negative log probability assigned to the correct token:

$$\mathcal{L} = -\sum_{t=1}^{T} \log p(y_t \mid x_{<t}).$$

Minimizing cross-entropy therefore pushes the model to place high probability mass on the true next token at every step in the sequence.

This softmax–cross-entropy pipeline is not unique to Transformers: it appears in convolutional models, recurrent networks, and many classifiers. For large language models, however, it is the bridge between the dense numerical space of hidden representations and the discrete symbolic world of text.

Remark 1. Later refinements, such as reinforcement learning with human feedback, modify the objective by replacing or augmenting cross-entropy with reward-driven loss functions. At their core, though, these methods still begin from the same logits, softmax normalization, and probability-of-token framework introduced here.

Remark 2. In practice, one often introduces a *temperature parameter* $T > 0$ to control the sharpness of the probability distribution derived from logits. The modified softmax is

$$p(y = j \mid x) = \frac{\exp\left(\frac{z_j}{T}\right)}{\sum_{k=1}^{|V|} \exp\left(\frac{z_k}{T}\right)}.$$

When $T = 1$, this reduces to the usual softmax. Lower temperatures $T < 1$ make the distribution more peaked, amplifying differences among logits and pushing probability mass toward the largest logit. Higher temperatures $T > 1$ flatten the distribution, reducing confidence and spreading probability mass more evenly across the vocabulary.

8 Complexity, Context Windows, and Practice

Self-attention forms an $n \times n$ score matrix, giving time and memory $O(n^2)$. Doubling sequence length roughly quadruples the cost. This motivates *context windows* and, for long documents, chunking or retrieval-augmented strategies.

9 Micro Exercises (calculator-only; no spreadsheet)

1. **Shapes.** For n=4, d_{model}=6, d_k=3, d_v=3, state the shapes of $X, W^Q, W^K, W^V, Q, K, V, S, A, Z$.
2. **Masking.** For n=3 with a causal mask, write the 3×3 pattern of zeros in A (which entries must be 0 in each row?).
3. **Hand softmax.** Reproduce the weights for $[0, 1, 2]$ to two decimals. Which token dominates and why?
4. **Weighted sum.** Using the V above, recompute $Z_{1\cdot}$ with your softmax weights to check your arithmetic.

10 Conclusion: The Impact of the Attention Revolution

By providing a parallel, learnable mechanism to connect distant parts of a sequence, attention overcomes the bottlenecks of recurrent processing. With multi-head attention, residual pathways, normalization, and light positional information, the Transformer forms the backbone of modern LLMs. The training objective—predict the next token via softmax and cross-entropy—closes the loop with the calculus and optimization developed in Chapter 7.