Attention and Language Models

Lecture 5a

Elodie Chervin Daniel Barbosa

TT 2026

University of Oxford & Nuffield College

Today's plan

From text to numerical representations

Contextual embeddings: simplified self-attention

Multi-head attention

The transformer block

Last week, in one slide

- Bag-of-words and tf-idf: count-based, sparse, throws away order.
- Word embeddings: semantics as geometry; analogies via vector arithmetic.
- Cliffhanger: *static* embeddings give one vector per word ⇒ "central bank" and "river bank" get the same vector. We need contextual embeddings.

Today: the mechanism that makes contextual embeddings (and almost all of modern AI) possible.

From text to numerical representations

Embeddings

The first learned object in the model is an embedding matrix:

$$E \in \mathbb{R}^{V \times d}, \qquad x_t = E[w_t].$$

- $V := |\mathcal{V}|$ is the vocabulary size: the number of unique tokens.
- d is the embedding dimension, chosen by the model designer.
- w_t is the token at position t; x_t is the embedding representation of token t.
- In a language model, E is learned jointly with the model's other parameters.

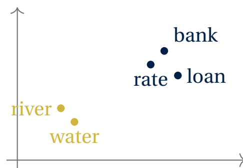
Embedding geometry

Set $d = 5$. Each token is a point, or vector, in a five-dimensional space:

token	1	2	3	4	5
$\text{bank}_{\text{finance}}$	0.72	0.10	0.61	−0.05	0.18
rate	0.69	0.14	0.58	−0.02	0.21
loan	0.66	0.08	0.63	−0.08	0.16
river	0.05	0.71	−0.04	0.64	0.20
water	0.02	0.76	−0.01	0.59	0.23

Geometry gives the model a notion of similarity:

$$\cos(x_i, x_j) = \frac{x_i^\top x_j}{\|x_i\| \, \|x_j\|}.$$



Nearby vectors tend to occur in similar contexts.

Static embeddings: useful but not enough

$$
\begin{aligned}
v_{\text{Berlin}} - v_{\text{Germany}} + v_{\text{France}} &\approx v_{\text{Paris}} \\
v_{\text{ECB}} - v_{\text{Eurozone}} + v_{\text{UK}} &\approx v_{\text{BoE}}
\end{aligned}
$$

"He sat on the river bank." vs. "The European bank raised rates."

word2vec / GloVe assign *one vector per word type*: "bank" gets the same embedding representation in both sentences.

To predict the next token well, a language model needs a representation of "bank" that already carries its context.

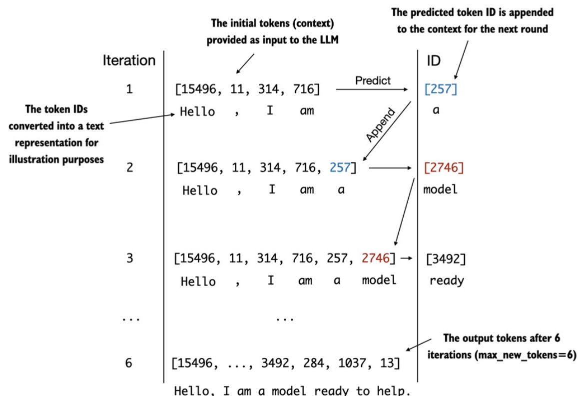
A language model is a next-token predictor

The chain rule opens the sequence probability into next-token terms:

$$
\begin{aligned}
P(w_1,\dots,w_T) = P(w_1) \\
\times P(w_2 \mid w_1) \\
\times P(w_3 \mid w_1, w_2) \\
\dots \\
\times P(w_T \mid w_{<T}) \\
= \prod_{t=1}^{T} P(w_t \mid w_{<t}).
\end{aligned}
$$

To predict after “the cat chased the mouse because it . . . ”, the representation of “it” must carry the relevant context.

Build a language model = build a next-token distribution.



Self-supervision: every token is a label

Cut the token-ID stream into fixed-length windows, and shift the target one step to the right:

row	input token IDs	target token IDs
1	[464, 3634, 6843, 284]	[3634, 6843, 284, 4485]
2	[3634, 6843, 284, 4485]	[6843, 284, 4485, 6217]

- Every adjacent token pair supplies a label for free — no human annotation.
- Here $L = 4$: each row has four input positions and four shifted targets, so the row gives four next-token predictions in parallel.

Contextual embeddings: simplified self-attention

The input as a (b, T, D) tensor

After tokenisation and embedding lookup, a batch of text is a three-axis tensor:

$$\underbrace{b}_{\text{documents}} \times \underbrace{T}_{\text{tokens per doc}} \times \underbrace{D}_{\text{embedding dim}} .$$

Fix one document. It is a matrix

$$X = \begin{pmatrix} x_1^\top \\ \vdots \\ x_T^\top \end{pmatrix} \in \mathbb{R}^{T\times D}, \qquad x_t = E[w_t] + P[t],$$

one row per token (token embedding + position embedding $P[t]$).

The challenge: every row x_t is still the *static* embedding. We want to replace it with a *contextual* row that mixes in the relevant information from the other tokens.

Why positions matter

Without $P[t]$, self-attention is permutation-equivariant: shuffle the rows of X, and the output rows shuffle the same way.

“rates will rise” vs. “will rates rise”

Same tokens, different order, different meaning — so position must enter the representation.

- GPT-2 / BERT: learned position embeddings.
- Modern LMs: rotary or relative position encodings.

Rotary and relative schemes still give the model order information, but they attach it to *pairwise attention* — how far token s is from token t — rather than only adding a fixed absolute row $P[t]$.

Simplified self-attention: the goal

Build the contextual row for each token as an *enriched* embedding — with no trainable weights yet, so the mechanism is fully transparent (Raschka §3.3).

For a given token with static embedding representation x_t, we want a context-vector representation of it:

$$z_t = \sum_{s=1}^{T} \alpha_{t,s} x_s,$$

a weighted average of *all* token embeddings $x_1, \ldots, x_T$. The weights $\alpha_{t,s}$ say how much token t should draw on token s.

Three steps, computed from X alone:

1. **scores** $\omega_{t,s}$ — how similar are tokens t and s?
2. **weights** $\alpha_{t,s}$ — normalise the scores into a distribution.
3. **context vector** z_t — weighted sum of the input rows.

Step 1 — attention scores

The score is the dot product of two token embeddings:

$$\omega_{t,s} = x_t^\top x_s.$$

The dot product measures *alignment*: the larger it is, the more the two tokens point the same way, and the more token t attends to token s.

Token list [the, central, bank, raised], with $D = 2$. Let X be the original embedding representation of those four tokens:

$$X = \begin{pmatrix} 1.0 & 0.0 \\ 0.7 & 0.7 \\ 0.5 & 0.5 \\ -0.3 & 0.9 \end{pmatrix}, \qquad \Omega = XX^\top = \begin{pmatrix} 1.00 & 0.70 & 0.50 & -0.30 \\ 0.70 & 0.98 & 0.70 & 0.42 \\ 0.50 & 0.70 & 0.50 & 0.30 \\ -0.30 & 0.42 & 0.30 & 0.90 \end{pmatrix}.$$

Row t holds the scores of token t against every token (including itself).

Step 2 — attention weights

Normalise each row of scores with a softmax, so the weights are positive and sum to one:

$$\alpha_{t,s} = \frac{\exp(\omega_{t,s})}{\sum_{r=1}^{T} \exp(\omega_{t,r})}.$$

The attention weights for “bank” are

$$\text{softmax}(0.50,\ 0.70,\ 0.50,\ 0.30) \approx (0.25,\ 0.30,\ 0.25,\ 0.20).$$

“bank” spreads its attention across the sentence, leaning most on “central”. The row is now a *distribution over positions*: how much of each token to mix in.

Step 3 — context vectors

The context vector is the weighted sum of the input rows:

$$z_t = \sum_{s=1}^{T} \alpha_{t,s}\, x_s.$$

For “bank”:

$$z_3 \approx 0.25\, x_1 + 0.30\, x_2 + 0.25\, x_3 + 0.20\, x_4 \approx (0.53,\ 0.52).$$

All tokens at once, in matrix form:

$$\Omega = X X^\top, \qquad A = \text{softmax}_{\text{row}}(\Omega), \qquad Z = A X.$$

Each static row x_t has been replaced by a contextual row z_t — still no trainable weights.

Simplified self-attention, end to end

```python
import torch, torch.nn.functional as F
X = torch.tensor([[1.0, 0.0], [0.7, 0.7], [0.5, 0.5], [-0.3, 0.9]])
scores  = X @ X.T                          # (4, 4): raw dot products
weights = F.softmax(scores, dim=-1)        # rows sum to 1
Z       = weights @ X                      # (4, 2): contextual vectors
```

This is the whole idea. What it *cannot* do: the weights are fixed by the raw embedding geometry, so the model has no way to *learn* what “relevant context” means.

From fixed geometry to learned attention

Simplified self-attention created context vectors using *raw* embedding similarity. Two limits for a language model:

- The weights are fixed by geometry — the model cannot *learn* which tokens matter for predicting the next one.
- "Similar direction" is not the same as "relevant context": what a token *looks for* and what it *offers* can differ.

Fix: give the model three learnable projections of each token, trained for next-token prediction.

The model learns what to attend to.

Same three steps — scores, weights, weighted sum — but now on *learned* views of each token.

Query, key, value

Let $x_t \in \mathbb{R}^D$ be the input embedding at position t. Choose an attention dimension d_k; in multi-head attention, usually $d_k = D/H$. For now, $d_k = D$ with a single head.

Project each token into three learned vectors:

$$M_Q, M_K, M_V \in \mathbb{R}^{d_k \times D}, \qquad q_t = M_Q x_t, \quad k_t = M_K x_t, \quad v_t = M_V x_t,$$

where $q_t, k_t, v_t \in \mathbb{R}^{d_k}$.

- **query** q_t: what token t is looking for.
- **key** k_s: what token s offers as context.
- **value** v_s: the content token s contributes.

The score $q_t^\top k_s$ is a scalar; the output for token t is a weighted average of value vectors $v_s \in \mathbb{R}^{d_k}$.

Scores, scaled weights, and context vectors

The same three steps as before, now on queries, keys, and values:

$$\alpha_{t,s} = \mathrm{softmax}_s\left(\frac{q_t^\top k_s}{\sqrt{d_k}}\right), \qquad c_t = \sum_{s=1}^{T} \alpha_{t,s}\, v_s.$$

Stacking rows ($Q = XW_Q$, etc., with $W_Q = M_Q^\top$) gives the standard matrix form:

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

$\sqrt{d_k}$ scaling: Without it, dot products grow with d_k and saturate the softmax, killing gradients. The scale keeps the variance of $q \cdot k$ at $\mathcal{O}(1)$.

What the learned weights buy us

The mechanism is identical to simplified self-attention; only the inputs are now *learned* views of each token.

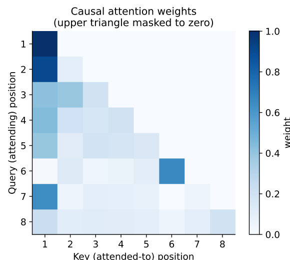
- Interpreting any single q_t or k_s is largely lost — but the *purpose* is unchanged.
- Trained for next-token prediction, M_Q, M_K, M_V learn to score a token pair as similar exactly when attending to s helps predict what follows t.
- So the model *learns which tokens to attend to*: in “the cat ran after the mouse as it escaped”, a well-trained model gives “it” a high weight on “mouse” (Gaillac §14.2.2).

Self-attention turns static input rows into contextual ones, with the contextualisation itself learned from data.

Tweak 1: causal attention and leakage

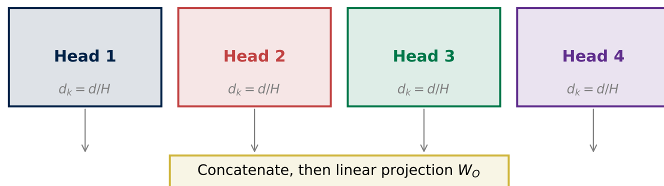
When predicting the token at position t, the model must not look at positions $t+1, \ldots, T$ — otherwise the answer leaks into the input.

Mask the future before the softmax: set the upper-triangular scores to $-\infty$, so those weights become exactly zero.



Multi-head attention

Multi-head attention



H parallel heads, each with its own M_Q, M_K, M_V in a $d_k = D/H$ subspace; concatenate and project back to $\mathbb{R}^D$ with W_O. Each head can specialise on a different relation (subject–verb, coreference, ...). BERT-base: $H = 12$ heads, $d_k = 64$, $D = 12 \times 64 = 768$.

Multi-head attention: intuition and parameters

One attention head gives one learned way to ask, "which previous tokens matter here?" Multi-head attention gives the model several such questions in parallel.

For head $h = 1, \ldots, H$:

$$q_t^{(h)} = M_Q^{(h)} x_t, \qquad k_t^{(h)} = M_K^{(h)} x_t, \qquad v_t^{(h)} = M_V^{(h)} x_t,$$

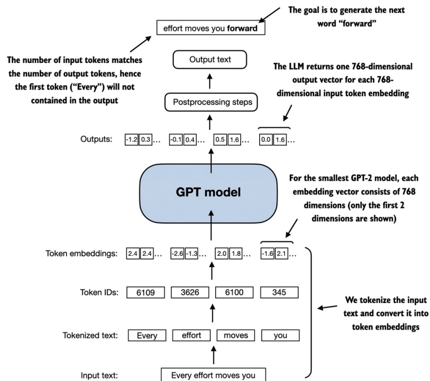
with $M_Q^{(h)}, M_K^{(h)}, M_V^{(h)} \in \mathbb{R}^{d_k \times D}$.

- Each head has its own parameters and its own attention map.
- If $d_k = D/H$, the Q, K, V projections across all heads have $3 H d_k D = 3D^2$ weights.
- After concatenation, $W_O \in \mathbb{R}^{D \times D}$ mixes the heads back into one D-dimensional representation per token.

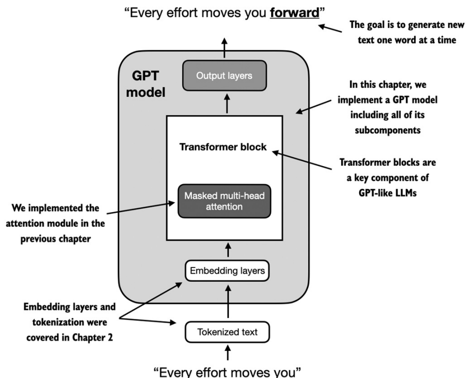
The transformer block

One transformer block

Raschka Figure 4.4: data flow through GPT



Raschka Figure 4.2: transformer blocks inside GPT



Why each piece is there

- **Self-attention:** routes information across positions.
- **Residual connections:** let gradients flow through deep stacks — a 96-layer model would be untrainable without them.
- **LayerNorm:** stabilises activation scales across layer depths and across batches.
- **Feed-forward (GELU):** per-position non-linear transformation; without it the block collapses to a linear sequence operation.

This is what GPT *is.* Stack L of these. Final linear → softmax over vocabulary. Done.

Counting parameters

Use model dimension d, feed-forward expansion $4d$, and ignore biases/LayerNorm for a first pass.

1. **Attention:** four $d \times d$ matrices:

$$W_Q, W_K, W_V, W_O \quad \Rightarrow \quad 4d^2.$$

2. **Feed-forward network:** one matrix $d \to 4d$ and one matrix $4d \to d$:

$$d(4d) + (4d)d = 8d^2.$$

3. **One block:**

$$4d^2 + 8d^2 \approx 12d^2.$$

4. L **blocks:** multiply by depth:

$$12Ld^2.$$

The architecture doesn't change. The dimensions, training data, and engineering do.

Model sizes: same block, different scale

The transformer block is the same idea across model families; scale mainly changes depth, width, vocabulary/output tables, and training data.

Model	Layers L	Dimension d	Reported parameters
Tiny GPT (today)	2	64	~0.05M
GPT-2 small	12	768	124M
GPT-3	96	12,288	175B

Recap and readings

Today we built the language-model core in layers:

- token IDs → embeddings → positional information;
- next-token prediction motivates contextual representations;
- self-attention creates context vectors by scores, weights, and weighted sums;
- learned Q, K, V projections and multiple heads let the model learn what to attend to;
- transformer blocks stack attention, feed-forward layers, residuals, and LayerNorm.

Suggested readings:

- Raschka, *Build a Large Language Model (From Scratch)*, ch. 3–4.
- Gaillac & L'Hour, *Modern Language Models*, ch. 14.
- Vaswani et al. (2017), *Attention Is All You Need*, sections 3.1–3.2.