

Self attention: the idea that changed everything

The bigram model from the previous chapter learned which token tends to follow the current token. Its limitation was not that its probabilities were fake. They were real. Its limitation was what information those probabilities could depend on: the current token, rather than the whole available prefix. Today we build the operation that lets a token position collect information from other positions. We will not train a language model yet. We will make the information flow visible, calculate it by hand, and test that future tokens cannot sneak through.

Why this matters

Imagine a code assistant completing a return statement. The current token alone may not tell it which variable is relevant. The definition might be earlier in the function. A useful internal representation should combine the current position with selected information from the preceding code. Looking up an embedding gives a token a starting vector. Attention gives that vector a way to depend on its context.

A fixed average of everything seen so far would be a start, but it treats a variable definition and an unrelated comment alike. We want the mixture to depend on the content. Which earlier positions contribute, and by how much, should be computed from the vectors themselves. That is the problem solved by the operation in this chapter. The original Transformer paper describes attention as a weighted sum of values, with weights determined by the compatibility of a query and corresponding keys [1].

This matters when you implement a model, not just when you read architecture diagrams. A transposed axis can normalize the wrong dimension. A mask applied in the wrong place can leak the training answer. A plausible-looking matrix and a decreasing loss are not enough to prove correctness. You need tests whose failure would expose the mistake. Our lab includes an intervention: change only a future input and check whether earlier outputs change.

For agent engineering, the useful connection is modest and concrete. Information present in a model's input can influence its internal computation when the architecture permits that flow. An attention mask describes permitted positions, not which text is trustworthy. It does not authenticate a document or grant a tool permission. Those are separate application responsibilities. Do not confuse a model's information-mixing rule with an authorization boundary.

The title refers to the importance of attention to the Transformer, not to a claim that this mechanism alone explains every model capability. Training data, objectives, optimization, feed-forward networks, positional information, and deployment choices still matter. Our evidence today proves an operation and its invariants, not language understanding.

The idea in plain words

At each position, make a shopping list, a label, and a package. The shopping list describes what this position is looking for. The label describes what another position can match. The package is the information that will actually be delivered. Compare the shopping list with each allowed label, turn the matches into shares, and combine the packages using those shares.

The standard names are query, key, and value. These are vectors, not English descriptions. The analogy separates their jobs: queries and keys determine the routing; values are the content being routed. All are calculated from the same input sequence in self-attention. The word “self” does not mean a position can only look at itself.

Here is our deliberately constructed example. There are three positions, counted from zero in the code. Their value vectors are [2, 0], [0, 4], and [6, 2]. These are invented input features chosen to make arithmetic inspectable, not embeddings extracted from a trained model. At the final position, the scores produce shares of one seventh, two sevenths, and four sevenths. The resulting vector is [3.714286, 2.285714], rounded as printed by the lab. It is a blend, not the value from just the winning position.

If every allowed position had the same score, the final position would instead receive the ordinary average, [2.666667, 2.000000]. Earlier positions have fewer allowed sources. The first position can use only itself, so its output is [2, 0]. Attention is therefore a content-dependent replacement for a fixed averaging rule, with a visibility rule placed around it.

Our exact example uses manually assigned projections. A separate part of the lab creates randomly initialized, trainable projections and checks that gradients reach them. Random initialization is not semantic training. Neither part demonstrates that the head has learned what a variable, noun, or pronoun means.

How it actually works

Start with rows of features

Call the input matrix X . A row corresponds to a token position; a column corresponds to a feature of that position's current representation. T means the number of positions, C means the input feature width, d_k means query and key width, and d_v means value width. In our constructed example, X has shape (3, 6), while Q , K , and V each have shape (3, 2). The lab prints these dimensions before doing any attention.

The projection equations are:

```
Q = X W_Q
K = X W_K
V = X W_V
```

W_Q , W_K , and W_V are parameter matrices. W_Q and W_K have mathematical shape (C, d_k); W_V has shape (C, d_v). Matrix multiplication applies the same transformation to every row. We do not create a new parameter matrix for every token position. In PyTorch, `nn.Linear(C, d_k, bias=False)` stores its weight transposed relative to this notation and evaluates X times weight-transpose [2]. That storage convention is why the lab fills rows of the layer's weight when selecting input features.

For the hand example, each query is [1, 0]. The keys are [0, 0], [$\sqrt{2} \ln(2)$, 0], and [$\sqrt{2} \ln(4)$, 0]. Here $\sqrt{2}$ means square root, and $\ln$ means natural logarithm, the inverse of the exponential function. The projection matrices simply select different pairs of columns from X . Real learned projections generally mix many input features instead of selecting such tidy blocks.

Keeping query and key separate is useful because seeking and offering are different operations. A token representation can produce a query that matches another token's key without the reverse match being equally large. A query-key score is not necessarily symmetric, even though both vectors originated in the

same sequence. Values need not have the same width as queries, because they are not involved in their dot product.

Match every receiving position with every source

For receiving position i and source position j , calculate:

```
s_ij = (q_i dot k_j) / sqrt(d_k)
S = Q K^T / sqrt(d_k)
```

The dot means multiply corresponding feature coordinates and add the products. K^T is the transpose of K , which turns its rows into columns. S therefore has shape (T, T) . Row i contains one score for every candidate source j . We use row-as-receiver consistently throughout the lab. Some diagrams choose another layout, so check the axis meaning rather than memorizing a triangle's appearance.

Our final query $[1, 0]$ matched with the middle key gives $\sqrt{2} \ln(2)$. Dividing by $\sqrt{2}$, printed as 1.414214, leaves $\ln(2)$, printed as 0.693147. The same calculation for all keys gives $[0.000000, 0.693147, 1.386294]$. These numbers are compatibility scores, not probabilities yet. Scores may be negative, positive, or zero.

Why divide at all? Under the simplifying assumption that query and key coordinates are independent, zero-mean, unit-variance random variables, the dot product's variance grows with d_k . Dividing by $\sqrt{d_k}$ controls that growth. Otherwise large magnitudes can push softmax toward very sharp distributions and small gradients. This is the original paper's motivation [1, section 3.2.1], not a promise that all trained scores have a particular variance. Scaling does not turn a dot product into cosine similarity; we never normalize the vector lengths here.

Hide future positions before normalization

For next-token prediction, row i may depend on positions up to and including i . Its target is the token after that position. Allowing the diagonal is therefore correct: seeing the current input token does not reveal the next-token target. Seeing position i plus one during training would reveal information unavailable when generating that target.

Define M_{ij} as zero when j is no greater than i , and negative infinity otherwise. Add M to S before softmax. Equivalently, use `masked_fill` to replace forbidden scores with negative infinity. For our row convention, allowed positions form the lower triangle including the diagonal. This mask is called causal because it prevents an output from depending on a future input within this sequence computation.

Why negative infinity rather than zero? The exponential of zero is one, so a zero score still receives a positive share. Negative infinity has exponential zero. Our deliberately broken zero-score mask gives the first position weights $[0.333333, 0.333333, 0.333333]$. It has accidentally opened access to the entire sequence. The row still sums to one, so a normalization-only test would miss the bug.

Do not mask every entry of a row in this elementary implementation. A row of negative infinities has no valid distribution; the naive softmax result is undefined and PyTorch returns NaNs for that direct operation. The causal diagonal ensures at least one allowed source. Padding and packed-sequence masks need their own careful policies and are outside this small implementation.

Turn scores into shares

Softmax exponentiates scores and divides each exponential by the sum across the row. Let a_{ij} be the resulting attention weight:

```
a_ij = exp(s_ij + M_ij) / sum_r exp(s_ir + M_ir)
A = softmax(S + M, along the source-position axis)
```

The index r runs across candidate sources for the same receiving position. `dim=-1` chooses that source axis in our tensor layout. The final row has exponentials $[1, 2, 4]$, with sum 7. Dividing by that sum gives $[1/7, 2/7, 4/7]$. At the middle position, the last source is forbidden, so the remaining exponentials $[1, 2]$ become $[1/3, 2/3, 0]$. At the first position, the sole legal weight is one.

Each row is a distribution over input positions, not over vocabulary entries. It answers how to mix available information. A language model's later output layer creates vocabulary scores for next-token prediction. Confusing those distributions makes attention appear to emit a word directly, which it does not do.

For numerical stability, softmax can subtract the row's maximum score before exponentiating. This leaves the ratios unchanged, because all exponentials are multiplied by the same factor. The lab checks that adding 1000 to every score leaves the unmasked softmax unchanged within numerical tolerance. Use PyTorch's stable implementation rather than manually exponentiating arbitrary large scores.

Masking after softmax is also wrong unless you carefully renormalize. The forbidden positions already consumed part of the denominator. Our broken implementation produces row sums $[0.142857, 0.428571, 1.000000]$ after zeroing the future weights. The allowed contributions have been shrunk. More subtly, their scale can still depend on the forbidden scores through that original denominator.

Deliver the weighted values

The output O is A times V :

```
o_i = sum_j a_ij v_j
O = A V
```

o_i is the output vector at receiving position i , and v_j is the package at source j . The shapes multiply as (T, T) times (T, d_v) , giving (T, d_v) . Compute the final example coordinate by coordinate. Its first coordinate is $(1/7)$ times 2 plus $(2/7)$ times 0 plus $(4/7)$ times 6, which is $26/7$. Its second is $(1/7)$ times 0 plus $(2/7)$ times 4 plus $(4/7)$ times 2, which is $16/7$. The lab checks these fractions against the computed output, then prints their rounded decimal forms.

This makes a useful intervention possible. Keep Q and K unchanged, and increase the first value's first coordinate by 7. The final receiver's weight for that source is $1/7$, so its first output coordinate increases by exactly 1 within floating-point tolerance. The weights remain identical. Routing and payload are separate parts of the computation, even though learned models derive both from shared input representations.

Without dropout, the weights are nonnegative and sum to one, so each head output lies in the convex hull of its value vectors: a mixture of them, not an arbitrary point outside them. That restriction does not

apply unchanged to an entire Transformer block, which adds other transformations and residual connections. Later chapters will put this head into that larger structure.

What “all at once” really means

The matrices calculate every receiving position in parallel during a training forward pass. A causal mask still prevents each row from using future tokens. Parallel arithmetic and causal information flow are compatible. During ordinary autoregressive generation, however, an unknown next token still has to be produced before it can become an input for subsequent generation. Parallel training is not a claim that the model can generate an arbitrary continuation simultaneously.

Pure unmasked attention without positional information is equivariant to a joint permutation of its input positions: reorder the input rows and the output rows reorder correspondingly. The causal mask adds an ordering constraint, but it is not the same as giving the model a useful explicit representation of distance or position. The original Transformer supplies positional encodings [1, section 3.5]; our shared GPT uses learned position embeddings. This chapter intentionally isolates mixing and leaves those position mechanisms for the model assembly discussion.

Build it

Run from the project root:

```
env -u PYTHONPATH .venv/bin/python chapters/llm-fundamentals/04-self-attention/lab.py
```

The lab adds the shared `labs/` directory to its import path based on its own file location, so it is not dependent on the shell's current directory. It uses CPU tensors, a fixed random seed, and no network access. The source lives in `lab.py`; the reusable head lives in `labs/tutorlab/attention_demo.py`. Existing tokenizer and GPT behavior is untouched.

Part 1: a baseline you already understand

`worked_example()` returns a small feature matrix and projection layers whose weights were chosen explicitly. It does not load a checkpoint. Setting every query to zero makes every dot product zero, so the allowed sources get equal weights. The lab prints the weights and their outputs before introducing unequal matches. Start by checking that these outputs really are averages of the allowed values.

The fixture briefly isolates its random initialization so constructing it does not disturb the caller's random-number sequence. It then overwrites the projections with selector matrices. That distinction keeps the hand example reproducible without disguising manually chosen parameters as learned ones.

Part 2: the full head

The central operation is short enough to read without an abstraction hiding it:

```
scores = (q @ k.transpose(-2, -1)) / math.sqrt(q.shape[-1])
allowed = torch.ones(q.shape[-2], q.shape[-2],
                    dtype=torch.bool, device=q.device).tril()
scores = scores.masked_fill(~allowed, float('-inf'))
```

```
weights = torch.softmax(scores, dim=-1)
out = weights @ v
```

The last two dimensions always mean positions and features for Q, K, and V. `transpose(-2, -1)` swaps only those dimensions, preserving any batch axes. The mask is Boolean and created on the same device as Q. The tilde means logical NOT: fill forbidden locations, not allowed ones. The reusable function checks shape compatibility explicitly to prevent accidental broadcasting between unrelated batches.

This is square self-attention only. The function deliberately requires matching sequence lengths and matching leading batch dimensions. A key-value cache introduces rectangular query and key layouts and positional offsets. Do not reuse this helper for that case without designing and testing the new mask semantics.

Part 3: prove the mask has teeth

The lab modifies the final input's key feature and value features, then recomputes attention. With the mask, the earlier output rows do not change. Without the mask, those earlier rows do change. That unmasked run is a negative control: it confirms that the chosen modification would reveal leakage if the mask were missing. We also intentionally construct the zero-score mask and the mask-after-softmax mistakes and observe their distinct failures.

The value-only intervention checks a different property. By leaving Q and K alone, we should preserve the weights exactly while changing the output. This protects the conceptual distinction between the routing matrix and what gets routed. It also gives you a counterexample to treating a heatmap as a complete explanation of the output.

Part 4: connect to trainable code

`SingleHead` contains separate `nn.Linear` layers for query, key, and value. On the lab's random batch, input shape (2, 4, 6) becomes output shape (2, 4, 2), with weights of shape (2, 4, 4). The leading dimension holds independent examples. They must not attend to each other. The projection layers together contain 36 trainable parameters in this configuration.

The lab compares its manual result with `torch.nn.functional.scaled_dot_product_attention`, passing `is_causal=True` and `dropout_p=0.0`. Its measured maximum absolute difference is 1.110e-16. Tiny floating-point differences are normal, so the assertions use numerical tolerance, not exact bitwise equality. We do not infer that a particular GPU kernel ran from this CPU result.

Finally, a simple squared-output objective calls `backward()`. Every projection receives a finite, nonzero gradient. That proves the computation is connected to the parameters through differentiation. It does not prove the parameters improved, because no optimizer step or language-learning objective was used. The regression tests independently compare both outputs and gradients against PyTorch on random batched inputs.

Actual output excerpts

These lines are copied from `build/lab-output.txt`, not hand-written expected results:

```
causal weights
[1.000000, 0.000000, 0.000000]
```

```
[0.333333, 0.666667, 0.000000]
[0.142857, 0.285714, 0.571429]
outputs
[2.000000, 0.000000]
[0.666667, 2.666667]
[3.714286, 2.285714]
```

```
causal prefix max change 0.000000
unmasked prefix max change 9.921041
WRONG zero-score mask, first weight row
[0.333333, 0.333333, 0.333333]
WRONG mask after softmax, row sums
[0.142857, 0.428571, 1.000000]
```

Part 5: account for the matrix

The explicit score matrix has T squared entries. The lab reports 16,384 entries and 65,536 bytes at $T=128$, versus 65,536 entries and 262,144 bytes at $T=256$, for one float32 matrix. Doubling sequence length quadruples this matrix's storage. This is not whole-model memory: batches, heads, layers, gradients, projections, and other activations add further costs. Our teaching implementation materializes the matrix because inspection is the goal.

Before changing the code, predict one of the [TRY THIS](#) experiments. Set queries to zero, change only a value, or remove the mask. State which property should change and which should remain invariant. Then run the experiment and compare. A failed prediction is useful evidence about your mental model, not something to hide by weakening the check.

Common mistakes and misconceptions

"Attention selects one token." Ordinary softmax attention mixes all permitted values according to their weights. A sharp distribution can resemble selecting one source, but this example intentionally blends several. [argmax](#) would produce a different operation and discard useful gradients.

"The biggest weight is the whole explanation." A weight does not tell you the size or direction of the corresponding value vector. Changing a value while preserving all weights changes the output in our lab. Subsequent layers and residual paths matter as well. Treat a heatmap as a view of routing, not a complete causal account of a model's answer.

"A zero score blocks a source." Exponentiating zero gives a positive contribution. Replace forbidden scores with negative infinity before softmax. Test both row normalization and the absence of future influence; either property alone can miss a broken implementation.

"The triangular mask is a universal Transformer rule." It is the appropriate visibility rule for the autoregressive head we are building. The original Transformer's encoder self-attention can use the whole input, while its decoder self-attention is causal [1]. Cross-attention takes queries from one stream and keys and values from another. The visibility rule follows the task.

"The queries and keys are a database lookup." The analogy helps name their roles, but the computation is a differentiable vector comparison and mixture. No exact string-key match is required. These matrices

are also not the durable memory of an agent application.

“Trainable means trained.” Our random layers have parameters and gradients. That is necessary plumbing for learning, not proof of useful learned behavior. Likewise, the neat ratios in the worked example came from our construction, not from a model discovering relationships in text.

In the real world

As of 2026-10-10, the official PyTorch attention documentation retrieved for this chapter describes `scaled_dot_product_attention` with causal masking, an optional mask, optional scaling, and explicit dropout control [3]. Its Boolean attention mask uses True to mean allowed. That is the opposite of the meaning of True in `MultiheadAttention`'s key-padding mask. Read the contract at the actual call site rather than reusing a mask because its shape looks right.

The same documentation warns that this functional operation applies dropout according to `dropout_p`; merely switching a surrounding module to evaluation mode does not change a literal nonzero argument. Our reference comparison passes zero explicitly. Dropout also means the realized post-dropout row weights need not sum to one, so our no-dropout invariant must not be transplanted without qualification.

Karpathy's nanoGPT source, checked on that date, packs query, key, and value projections into one linear layer, splits them, and reshapes them for multiple heads [4]. It calls the PyTorch operation when available and contains a spelled-out fallback with score scaling, causal masking, softmax, and multiplication by values. Separate projection modules here are a teaching choice, not a requirement of attention.

The shared project implementation in `labs/tutorlab/gpt.py` uses the same visible sequence of operations inside `CausalSelfAttention`, then recombines heads and applies an output projection. Reading that existing file after this lab should feel like recognizing a larger version of the same mechanism. We do not modify it in this chapter, because later lessons build on its established behavior.

FlashAttention addresses the memory traffic of attention through tiled computation rather than redefining attention as a cheap approximation [5]. Its paper describes an exact attention algorithm that reduces transfers between GPU memory levels. The important distinction is that the conceptual matrix need not be fully stored at once in an optimized implementation. That does not turn dense attention's arithmetic into a universally linear-time operation. Our small CPU parity test establishes numerical agreement with PyTorch, not a speed benchmark for FlashAttention or any frontier model.

Go deeper

These moments were verified on 2026-10-10 with `tools/find_lecture_moment.py`. YouTube transcripts were IP-blocked, so the evidence is the creators' published chapter markers, not claimed verbatim speech. The same method is recorded by the chapter gate.

- Andrej Karpathy, “Let's build GPT”, 47:11: <https://www.youtube.com/watch?v=kCc8FmEb1nY&t=2831>. The marker is “the trick in self-attention: matrix multiply as weighted aggregation”. Watch how multiplying by a matrix becomes mixing information across positions. Relate each row to our equal-score baseline.
- The same lecture, 62:00: <https://www.youtube.com/watch?v=kCc8FmEb1nY&t=3720>. The marker identifies “THE CRUX OF THE VIDEO: version 4: self-attention”. Compare the learned queries and keys with the hand-assigned selector projections in this lab.

- 3Blue1Brown, "Attention in transformers, step-by-step", 11:08: <https://www.youtube.com/watch?v=eMlx5fFNoYc&t=668>. The creator's marker is "Masking". Follow where forbidden connections disappear, then continue to the value discussion and distinguish the attention pattern from the vector it produces.

For the formal account, read the attention and positional-encoding sections of the original Transformer paper [1]. For implementation, compare the manual and optimized branches in nanoGPT [4]. Keep multi-head attention as the next step rather than trying to absorb the whole model in one sitting: parallel heads let different learned projections form different mixtures before those mixtures are combined.

Explain it back

Without looking at the equations, explain how the final position turns the value vectors [2, 0], [0, 4], and [6, 2] into a new vector. Include what determines the shares, why future positions are hidden before normalization, and why the resulting vector is not yet a next-token prediction. Then apply your explanation: if only the first value's first coordinate increases by 7, with queries and keys fixed, what changes and what stays the same?

Your study partner should check your explanation and application separately, identify one gap, and ask one follow-up at a time. Reading, watching, or running the lab is not itself evidence of mastery.

Check yourself

The prompts and answer rubrics are in [quiz.json](#). Answers stay there so you can predict before checking. The recall cards cover the mechanism, mask, value separation, and causality test.

1. What determines the attention weights, and what gets mixed?
2. Why include the diagonal in this next-token predictor's causal mask?
3. Which axis should softmax normalize in the lab's score tensor?
4. Why scale by the square root of the query/key width?
5. Predict the final weights and output if every query becomes zero.
6. Predict the effect of changing only a value coordinate.
7. Explain why row sums alone cannot prove causality, and propose an intervention test.
8. Explain the difference between attention weights and next-token probabilities.

Sources

1. Vaswani et al., "Attention Is All You Need". Sections 3.2, 3.5, and 4 support the attention mechanism, scaling motivation, positional information, masking, and parallelism discussion: <https://arxiv.org/html/1706.03762v7>
2. PyTorch, [torch.nn.Linear](#), official documentation, retrieved 2026-10-10. Parameter shape, transposed weight convention, and bias behavior: <https://docs.pytorch.org/docs/2.14/generated/torch.nn.Linear.html>
3. PyTorch, [scaled_dot_product_attention](#), official documentation, retrieved 2026-10-10. Mask semantics, dropout warning, shape contract, and backend-dependent numerics: https://docs.pytorch.org/docs/2.14/generated/torch.nn.functional.scaled_dot_product_attention.html
4. Karpathy, nanoGPT, [model.py](#), source checked 2026-10-10. Combined projections, causal attention, and model structure: <https://github.com/karpathy/nanoGPT/blob/master/model.py>

5. Dao et al., "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness". The tiling and memory-traffic claims are from the paper, not a benchmark performed here:
<https://arxiv.org/abs/2205.14135>
6. Karpathy, "Let's build GPT: from scratch, in code, spelled out." Creator-published timestamp markers verified 2026-10-10: <https://www.youtube.com/watch?v=kCc8FmEb1nY>
7. 3Blue1Brown, "Attention in transformers, step-by-step". Creator-published marker verified 2026-10-10: <https://www.youtube.com/watch?v=eMlx5fFNoYc>
8. Academy curriculum baseline, consulted for coverage rather than copied:
<https://krypteiasec.com/academy/llm-fundamentals/ch4>

All experimental numbers are from this chapter's `build/lab-output.txt` and the corresponding assertions in `lab.py`. The reading-and-lab allowance in the front matter is a rough study allocation, not a measured completion time. No hosted model, external corpus, paid inference, or trained semantic attention pattern is claimed by this lab.