

Multi head attention: many perspectives at once

Last chapter, a token position learned how to collect information from other positions: compare queries with keys, normalize the scores, then blend values. Keep that operation intact. Today we let several independently parameterized copies produce different blends, preserve those blends side by side, and learn how to combine their features. That is multi-head attention. The mathematics is familiar; the important new ideas are independent routing, tensor layout, and the distinction between collecting information across positions and mixing features within a position.

Why this matters

A code completion may benefit from information about a variable's definition and the structure of the current expression. One attention head produces a single distribution over allowed source positions for each receiving position. Every coordinate of that head's value vector is mixed using that same distribution. If different useful features should use different source mixtures, forcing them through one distribution can be restrictive. Several heads provide several independently computed distributions instead [1].

Do not turn that motivation into a story about little people inside the model. We do not program a grammar head, a coding head, or a truth head. We create separate projection parameters and train the larger model on its objective. Those parameters may develop useful patterns, overlapping patterns, or patterns that are hard to summarize in English. The architecture supplies an opportunity for different routing, not a guarantee of named specialists.

For AI engineering, this chapter closes the gap between understanding an equation and reading a working Transformer. Real implementations often combine projections into one matrix multiplication and reshape the result. The equations are still there, but the head dimension is easy to confuse with the token dimension. An incorrect reshape may preserve every visible shape while quietly mixing values from different positions. Shape checks are necessary, but they are not sufficient.

For agent work, the practical benefit is architectural literacy. A model's internal attention heads are not separate agent workers or separately authorized tool users. Increasing their number does not create additional permission boundaries. Here we test numerical information flow inside a model layer. Application authorization remains outside that calculation. No security claim follows from our attention heatmaps or from the fact that a mask is causal.

By the end, you will have calculated a small example, built a real registered PyTorch module, and compared both its results and its gradients with an independent library implementation. We will not train semantic head specializations or benchmark production inference. Those are different experiments that need their own evidence.

The idea in plain words

Imagine asking several readers to summarize the same page, but letting each reader use a different rule for deciding which passages matter. Keep their summaries separate initially. Then combine the useful

parts. The readers are only an analogy: our heads are matrix operations with separate parameters, all receiving the same input vectors.

The lab makes this concrete without any learned language behavior. Its source value vectors are [2, 0], [0, 4], and [6, 2]. At the last position, head 0 assigns the sources shares of one seventh, two sevenths, and four sevenths. Its result is [3.714286, 2.285714]. Head 1 reverses the source preference: four sevenths, two sevenths, and one seventh. Its result is [2.000000, 1.428571]. These are measured lab outputs, rounded for display.

Put the results next to each other and you get [3.714286, 2.285714, 2.000000, 1.428571]. That operation is concatenation. It has not averaged the heads or chosen a winner. The two routes still occupy distinguishable blocks of coordinates. An output projection then turns those coordinates into [5.714286, 0.857143, 1.714286, 3.714286] using an explicitly chosen matrix that we will inspect.

One misconception needs fixing immediately: a head does not ordinarily receive only a disjoint slice of the raw input embedding. Each head's query, key, and value projections can read the full input width. What gets split into head-sized pieces in a packed implementation is the projected representation. Our neat hand example uses selector weights for easy arithmetic, but the general module uses dense trainable projections.

How it actually works

Begin with the single-head operation

Let X be the input tensor. B is the number of independent examples in a batch, T is the number of token positions in each example, and C is the feature width at each position. Thus X has shape (B, T, C) . Let H be the number of heads and d be the width of each head. In this chapter, all heads have equal width and H times d equals C , so d equals C divided by H .

The hand example uses $B=1$, $T=3$, $C=4$, $H=2$, and $d=2$. The random test uses input shape $(2, 4, 8)$ and per-head weight shape $(2, 2, 4, 4)$. These are intentionally tiny configurations printed by the lab, not specifications of a commercial model.

For head h , calculate:

```
Q_h = X W_Qh
K_h = X W_Kh
V_h = X W_Vh
A_h = softmax(Q_h K_h^T / sqrt(d) + M)
O_h = A_h V_h
```

The subscript h identifies which head's parameters we use. W_{Qh} , W_{Kh} , and W_{Vh} each have mathematical shape (C, d) . Q_h , K_h , and V_h have shape (B, T, d) . Transposing K_h swaps its final two axes, not the batch axis. The scores and attention weights A_h have shape (B, T, T) . Each head output O_h has shape (B, T, d) .

M is the causal mask: zero for an allowed source position, negative infinity for a future position. Softmax normalizes across source positions, the last axis. The receiving position may include itself because it will eventually predict the following token. Each head applies the same visibility rule independently. The

scaling divisor is the square root of d , the number of coordinates actually used in that head's dot product, not the square root of C [1].

Unlike the scores, the projection weights do not grow with sequence length. The same learned transformation is applied to every input position. A head may form a different attention distribution at every position because its query changes with the input. It is not limited to one fixed pattern across the whole sequence.

Construct two routes you can calculate by hand

The input rows have four features: a constant query feature, a log-key feature, and two value features. The first source has row $[1, 0, 2, 0]$. The second has $[1, \ln(2), 0, 4]$. The third has $[1, \ln(4), 6, 2]$. The function $\ln$ is the natural logarithm, the inverse of exponentiation with base e . These are constructed numeric inputs, not embeddings that came out of a trained model.

For both heads, the query projection produces $[1, 0]$. Head 0's key projection multiplies the log-key feature by $\sqrt{2}$ and places it in the first key coordinate. Its other key coordinate is zero. Dotting the query with that key and dividing by $\sqrt{2}$ cancels the scale, leaving scores $[0, \ln(2), \ln(4)]$. At the last receiving position, every source is allowed.

Exponentiating gives $[1, 2, 4]$. Their sum is 7, so softmax gives $[1/7, 2/7, 4/7]$. The first output coordinate is the weighted sum of the first value coordinates: $(2 + 0 + 24)/7$, or $26/7$. The second is $(0 + 8 + 8)/7$, or $16/7$. These fractions are the exact targets asserted by the lab, and their decimal forms are the head 0 result you already saw.

Head 1 uses the same query but negates the first head's key projection. Its scores become $[0, -\ln(2), -\ln(4)]$. Exponentiating gives $[1, 1/2, 1/4]$. Multiplying numerator and denominator by 4 makes the shares easy to see: $[4/7, 2/7, 1/7]$. Applying those shares to the same values gives first coordinate $(8 + 0 + 6)/7 = 14/7$ and second coordinate $(0 + 8 + 2)/7 = 10/7$.

The two outputs differ solely because their routing differs. Keeping values identical is a teaching choice that isolates this difference. A general multi-head layer also learns separate value projections, so heads may select different features as well as different source mixtures. None of this example establishes that head 0 understands recency or that head 1 understands subjects. We manually arranged the scores.

Preserve the routes before combining them

Concatenate head outputs along the feature axis:

```
Z = concat(o_0, o_1, ..., o_(H-1))
Y = Z W_O
```

Z has shape (B, T, Hd) . *Because we chose $Hd=C$* , its shape is already (B, T, C) . W_O has mathematical shape (C, C) , and Y also has shape (B, T, C) . Concatenation restores the total width in this configuration. The output projection mixes coordinates; it is not required merely to repair the width.

For the final position, name the concatenated coordinates a , b , c , and d_value to avoid confusing a value coordinate with the head-width symbol. Our explicit output projection computes $[a+c, b-d_value, a-c, b+d_value]$. Substituting $[26/7, 16/7, 14/7, 10/7]$ gives $[40/7, 6/7, 12/7, 26/7]$, exactly the projected result checked in the lab.

The subtraction matters. Each no-dropout head output is a nonnegative weighted mixture of that head's values. The output projection need not be another such average: it can add, subtract, amplify, or rotate feature combinations. It is an ordinary learned linear transformation. There is no softmax across heads in this implementation, and no requirement that its weights sum to one.

In PyTorch, `nn.Linear` stores the transpose of the mathematical weight matrix and computes input times stored-weight-transpose. The lab prints the stored rows of the output projection, each one defining an output coordinate. Keeping the storage convention explicit prevents an easy mistake when copying weights into another implementation [8].

Summing the two head outputs immediately would leave only [5.714286, 3.714286]. You would no longer have separate coordinates for their differences. Concatenation retains information that the subsequent projection can choose to use. A learned output projection can implement a sum if that is useful, but forcing the sum first is a stronger restriction.

The output projection acts independently at each token position. It cannot suddenly read a future token. Attention provides the cross-position mixing, and the final linear layer provides within-position feature mixing. This distinction explains why the whole module remains causal after projection.

Write the same computation with a head axis

The clearest teaching implementation loops over independent head modules. Efficient implementations often group their projections into large matrices instead. Stack all query projection weights together, all key weights together, and all value weights together. A single linear operation then produces a last dimension containing the projected queries, keys, and values for every head.

Split that last dimension into Q, K, and V, each with shape (B, T, C). Reshape each into (B, T, H, d), then transpose the middle dimensions to get (B, H, T, d). Matrix multiplication now treats batch and head as independent leading dimensions. It computes every head's scores with the same visible formula, producing (B, H, T, T).

This is parallel expression of independent routes, not communication between heads during scoring. Their parameters differ, but no head reads another head's attention weights. The packed and loop forms are algebraically equivalent when the parameter ordering matches. Our tests compare their outputs, weights, concatenations, and gradients, rather than assuming equivalence from the shapes.

After multiplying weights by values, the result has shape (B, H, T, d). Transpose back to (B, T, H, d), make the layout contiguous if using `view`, and merge the last two dimensions. Skipping the transpose can still produce a tensor with the desired outer shape, while assigning features from the wrong token or head to each row. That is why an independent numerical reference is valuable.

Count what changes when heads change

For a bias-free layer at fixed total width, each head has three projection matrices with Cd parameters each. Across H heads that is $3HCd = 3CC$ parameters. The output projection adds CC , giving $4C \cdot C$ altogether. The lab measures 256 parameters at $C=8$ for $H=1$, $H=2$, and $H=4$. More heads do not automatically mean more projection parameters when each head becomes correspondingly narrower.

The explicit attention matrices tell a different story. They contain $BHT \cdot T$ entries. For the lab's $B=2$ and $T=4$, the head sweep prints 32, 64, and 128 score entries. More heads at fixed width increase this teaching implementation's score storage even though projection parameter count stays fixed.

The lab also holds $B=2$ and $H=2$ while increasing T from 128 to 256. A single float32 score tensor grows from 262,144 bytes to 1,048,576 bytes. Doubling sequence length quadruples this tensor's storage. These are arithmetic storage counts, not measured peak memory. They exclude other activations, gradients, parameters, and allocator overhead.

The dense score and value products have leading arithmetic cost proportional to $BHTd$, which becomes $BT * C$ at fixed total width. Thus parameter count, arithmetic scaling, explicit intermediate storage, and measured wall-clock speed are different questions. The original paper motivates narrower per-head representations as a way to keep computation comparable to a full-width head [1]. Our lab does not measure a hardware speed advantage.

Build it

Run this from the project root:

```
env -u PYTHONPATH .venv/bin/python chapters/llm-fundamentals/05-multi-head-attention/lab.py
```

The lab finds `labs/` relative to its own path, so it also works when launched from another directory. It uses CPU tensors, seeded initialization, and no downloads or API credentials. The reusable module is `labs/tutorlab/multihead_demo.py`. It imports the previous chapter's `SingleHead` and `attention` without modifying them. The model intentionally omits dropout, bias, padding, and key-value caching so that each invariant has a clear scope.

Part 1: register genuinely independent modules

The essential construction is short:

```
self.heads = nn.ModuleList([SingleHead(width, self.head_width)
                             for _ in range(num_heads)])
self.proj = nn.Linear(width, width, bias=False)
```

Each comprehension iteration creates a new head with distinct parameters. `ModuleList` registers those submodules so `parameters()`, device moves, and state serialization can find them [3]. An ordinary Python list is not a substitute for that registration. Repeating the same head object would share parameters rather than creating independent routes.

The forward method asks each head for its output and weights, concatenates outputs on the last dimension, and stacks weights on a new head dimension. It returns the final output, per-head weights, and pre-projection concatenation. Returning these inspection tensors is useful for teaching but is not a requirement for production inference.

The worked fixture overwrites the initial weights with the hand-chosen projections. A separate random model uses normal trainable initialization. Keeping those cases separate prevents a misleading story in which manually engineered fractions are presented as learned linguistic behavior.

Part 2: inspect the actual mixture

These lines are copied from `build/lab-output.txt`:

```
head 0 final weights [0.142857, 0.285714, 0.571429]
head 0 final output [3.714286, 2.285714]
head 1 final scores [0.000000, -0.693147, -1.386294]
head 1 final weights [0.571429, 0.285714, 0.142857]
head 1 final output [2.000000, 1.428571]
```

```
final concatenation [3.714286, 2.285714, 2.000000, 1.428571]
final projected output [5.714286, 0.857143, 1.714286, 3.714286]
```

Read the final output against the printed projection rows. Do not skip directly to `CHECK PASSED`. The important exercise is identifying which output coordinate uses a sum and which uses a difference. The lab then zeroes head 1's concatenated block and measures the resulting change after projection. Its signed contribution is [2.000000, -1.428571, -2.000000, 1.428571]. Removing a head can increase some coordinates, because contributions are not restricted to positive additions.

Part 3: perturb inputs rather than trusting a triangle

The lab changes only the last input position. Earlier outputs remain unchanged under causal attention, while an unmasked negative control changes. It also checks that every future weight is zero and every no-dropout row sums to one. These checks protect different properties. A wrong-axis softmax can normalize something without constructing the intended distribution, and a missing mask can preserve width without preserving causality.

A second intervention sets the constant query feature to zero, making all query vectors zero in this fixture. Both heads then assign equal weights to allowed positions. At the final position, their concatenation becomes [2.666667, 2.000000, 2.666667, 2.000000], and the projected output becomes [5.333333, 0.000000, 0.000000, 4.000000]. The difference coordinates vanish because the two head results are now equal, not because the output projection stopped working.

Part 4: compare with PyTorch and inspect gradients

The random-model comparison copies our projections into `torch.nn.MultiheadAttention`. It sets `batch_first=True`, disables bias and dropout, and supplies a Boolean upper-triangle mask. In this API, True means blocked. We request `average_attn_weights=False` so the returned weights retain their head axis rather than averaging away the distinction under study [2].

The lab reports:

```
loop matches packed True
loop matches PyTorch True
per-head weights match PyTorch True
all projection tensors have finite nonzero gradients True
```

Those comparisons use floating-point tolerances. An exactly identical bit pattern is not required across different multiplication orders. The separate test file goes further: it compares gradients of the input and each projection against PyTorch, checks batch isolation, exercises single-token and noncontiguous inputs, and rejects incompatible shapes or head counts.

Calling `backward()` on a squared-output objective demonstrates differentiability and parameter registration. It does not demonstrate language learning. No training corpus, optimizer update, or semantic evaluation is part of this lab. Later chapters will connect these parameters to the full model and training loop.

Part 5: make a prediction before changing the code

Choose one `TRY THIS` comment. Predict the equal-query result, the effect of ablating a head before projection, or the parameter and score-storage consequences of changing head count at fixed width. Then run it and compare. If your prediction fails, identify whether the mistake concerned routing, feature mixing, or storage. Those are different mechanisms and deserve different corrections.

Common mistakes and misconceptions

"Split the raw embedding into pieces and give each head one piece." The general layer first projects the full input into head-specific representations. Our query weight has shape (2, 4), so each output coordinate can depend on any of the four input coordinates. Packed code hides this behind one large linear layer followed by reshaping.

"Concatenation mixes the heads." It only places their outputs next to each other. The output projection performs learned feature mixing. In the equal-width configuration, concatenation already restores width C. Saying that the projection is needed to shrink H full-width heads would describe a different configuration.

"More heads means a proportionally bigger model." Not at fixed total width with the bias-free setup used here. The lab's parameter count remains 256. Explicit score storage still grows with head count, and head width shrinks, so unchanged parameters do not mean every cost or modeling property stays unchanged.

"Each head learns a distinct human-readable job." Separate parameters permit different behavior but do not force useful diversity or unique semantics. Our opposite attention patterns were constructed. A name such as subject head needs evidence from a trained model and controlled interventions, not just an attractive heatmap.

"The output projection can violate the causal mask." It mixes features within the same receiving position, not across positions. If the head outputs are causal, a token-wise output projection preserves that dependency boundary. A reshape bug that crosses positions is another matter, which is why the intervention test checks the complete module.

"Attention weights tell the whole story." They describe routing, but value projections determine the payload and output projections determine how routes contribute downstream. Our ablation produces signed changes. A large attention weight alone does not establish that a source helped, harmed, or determined a final answer.

In the real world

As of 2026-10-10, the retrieved official PyTorch documentation describes `MultiheadAttention` as a reference implementation of the original architecture and notes that it can use optimized scaled-dot-product attention implementations when possible [2]. It documents splitting the total embedding width

across heads. That shape statement should be read together with the learned projection equations, not as a requirement to isolate raw input feature slices.

Karpathy's nanoGPT source, checked on the same date, constructs a combined query-key-value linear layer, splits its result, and reshapes into batch, head, token, and head-width axes [4]. That is the same transformation our packed path exposes. Combining linear operations changes their organization, not the underlying permission for each projected coordinate to read the full input width.

The local `labs/tutorlab/gpt.py` already implements this packed layout inside `CausalSelfAttention`. Its output projection restores a feature representation suitable for a residual addition in `Block`. That shared implementation also includes features intentionally omitted from our minimal demonstration, including biases and configurable dropout. We have not changed it or used our bias-free parameter formula as its parameter count.

Maintaining output shape makes it possible to stack blocks and add residual connections, but matching shape alone does not make a complete Transformer block. Normalization and the position-wise feed-forward network still have jobs to do [1]. Those are the next chapter's subjects. Multi-head attention is an important component, not an entire language model or an ensemble of separately trained models.

Go deeper

The lecture moments below were checked on 2026-10-10 with `tools/find_lecture_moment.py`. Transcripts were IP-blocked, so the verified anchors are creator-published chapter markers. They are not presented as quotations from spoken transcripts.

- Andrej Karpathy, "Let's build GPT", 81:59: <https://www.youtube.com/watch?v=kCc8FmEb1nY&t=4919>. The marker is "multi-headed self-attention". Watch how independent head modules are joined, then compare the module registration and concatenation with our loop implementation.
- Karpathy, the same lecture, 106:22: <https://www.youtube.com/watch?v=kCc8FmEb1nY&t=6382>. The marker is "super quick walkthrough of nanoGPT, batched multi-headed self-attention". Track the transition from separate modules to a head dimension in packed tensors.
- 3Blue1Brown, "Attention in transformers, step-by-step", 19:19: <https://www.youtube.com/watch?v=eMlx5fFNoYc&t=1159>. The marker is "Multiple heads". Watch how different attention patterns coexist, and keep the distinction between an explanatory picture and evidence of an actual trained specialization.

For the formal definition, read section 3.2.2 of "Attention Is All You Need" [1]. For the programming contract, read the PyTorch layer's mask and attention-weight return options [2]. For a real codebase, compare nanoGPT's projection and reshape sequence [4] with our independently tested packed path. You do not need to absorb the training loop yet to explain this operation accurately.

Explain it back

Explain to a colleague why several heads can retain different source mixtures while all seeing the same input, and distinguish concatenation from the output projection. Then apply your explanation to a new case: if the fixture's query feature becomes zero at every position, what happens to the two final mixtures and to the output coordinates that subtract one head from the other?

Your study partner checks the explanation and application separately, finds one gap, and asks one follow-up at a time. Running the lab or watching the walkthrough is not evidence of mastery by itself.

Check yourself

The full prompts, answers, and rubrics are in [quiz.json](#). Predict before opening the answers. The questions cover:

1. Whether a head can read the full input feature width.
2. What concatenation and the output projection each do.
3. Which feature width determines score scaling.
4. What changes when the number of heads increases at fixed total width.
5. Predicting equal-query mixtures and projected differences.
6. Predicting a head's signed contribution after projection.
7. Explaining a complete causality test with a negative control.
8. Explaining why distinct head numbers do not establish distinct semantic jobs.

The recall cards revisit full-input projections, tensor axes, parameter accounting, and head interpretation. Correct immediate answers and later independent recall are different kinds of evidence; neither should be inferred from delivery.

Sources

1. Vaswani et al., "Attention Is All You Need", sections 3.1, 3.2.1, and 3.2.2. Learned projections, scaling, concatenation, output projection, computational motivation, and residual-compatible width: <https://arxiv.org/html/1706.03762v7>
2. PyTorch, [MultiheadAttention](#), official documentation, retrieved 2026-10-10. Reference architecture, packed projections, head dimensions, mask semantics, and per-head attention returns: <https://docs.pytorch.org/docs/2.14/generated/torch.nn.MultiheadAttention.html>
3. PyTorch, [ModuleList](#), official documentation, retrieved 2026-10-10: <https://docs.pytorch.org/docs/2.14/generated/torch.nn.ModuleList.html>. The registration behavior is also exercised by our parameter and gradient tests.
4. Andrej Karpathy, nanoGPT [model.py](#), source checked 2026-10-10: <https://github.com/karpathy/nanoGPT/blob/master/model.py>
5. Andrej Karpathy, "Let's build GPT: from scratch, in code, spelled out." Verified creator markers: <https://www.youtube.com/watch?v=kCc8FmEb1nY>
6. 3Blue1Brown, "Attention in transformers, step-by-step". Verified creator marker: <https://www.youtube.com/watch?v=eMlx5fFNoYc>
7. Academy chapter consulted for coverage, not copied: <https://krypteiasec.com/academy/llm-fundamentals/ch5>
8. PyTorch, [Linear](#), official documentation, retrieved 2026-10-10. Transposed weight storage and within-position feature transformation: <https://docs.pytorch.org/docs/2.14/generated/torch.nn.Linear.html>

All experimental numbers are from [build/lab-output.txt](#) or the hand-calculation assertions in [lab.py](#). The front-matter minutes are a rough reading-and-lab allocation, not a measured learning time. This chapter's hand-assigned projections are an explicitly synthetic teaching fixture; the numerical library comparisons are real executions, not invented model responses.