

# Compute and Perturb Attention

Gavin · AI-assisted course design

## Contents

|   |                                              |   |
|---|----------------------------------------------|---|
| 1 | Goal .....                                   | 3 |
| 2 | The fixed head .....                         | 3 |
| 3 | Part 1 — Compute before coding .....         | 3 |
| 4 | Part 2 — Predict controlled changes .....    | 4 |
| 5 | Part 3 — Implement small arrays .....        | 4 |
|   | 5.1 Learner scaffold .....                   | 4 |
|   | 5.2 Run and compare .....                    | 5 |
|   | 5.3 Hand-worked alternative .....            | 6 |
| 6 | Part 4 — Explain what changed .....          | 6 |
| 7 | Completed reference implementation .....     | 7 |
| 8 | Optional checks with explanatory value ..... | 9 |
| 9 | Completion criteria .....                    | 9 |

# 1 Goal

Trace one attention head from projected vectors to output. Then change one part of the computation at a time to distinguish attention coefficients, value contributions, and mask effects.

**Prerequisite:** Lesson 2.2 — Attention, through the three-position example. Read its interpretation section before writing your conclusions.

**Time:** about 60–90 minutes. Allow another 20 minutes for the optional checks.

**Requirements:** paper or a text editor. The coding route uses an existing Python 3 installation, ordinary CPU execution, and the standard library only. No account, API, GPU, internet access, model download, package installation, or paid compute is needed. A hand-worked route is equally valid.

**Expected artifact:** a short `submission.md` containing predictions, calculations, comparisons, and interpretation. For the coding route, also retain `tiny_attention.py` and its actual output. Label hand-derived answers and observed program output separately.

! Predict before executing

Complete Parts 1 and 2 before running code or reading the reference solution. If you use an LLM, ask it to critique your attempt before showing you its own. The supplied scaffold is unfinished; the completed reference is a separate comparison aid.

## 2 The fixed head

All position vectors are rows. Use

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}, \quad W_Q = I_2, \quad W_K = \sqrt{2} \ln(2) I_2, \quad W_V = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix}.$$

Compute  $Q = XW_Q$ ,  $K = XW_K$ ,  $V = XW_V$ , then

$$S = QK^T / \sqrt{2}, \quad A = \text{softmax}_{\text{sources}}(S + M), \quad Z = AV.$$

The causal mask has zero on and below the diagonal and negative infinity above it. There is one head, no bias, no dropout, no normalization, and no residual addition in this Lab's measured output. An identity output projection would leave  $Z$  unchanged. These choices isolate the mechanism; this is not a trained language model.

## 3 Part 1 — Compute before coding

1. Write the shapes of  $X$ , each projection matrix,  $Q$ ,  $K$ ,  $V$ ,  $S$ ,  $A$ , and  $Z$ .
2. Calculate the three projected matrices. Retain the exact factor  $\sqrt{2} \ln 2$  instead of rounding it early.
3. Expand  $q_2 \cdot k_1$  and  $q_3 \cdot k_3$ , then construct the complete scaled score matrix.
4. Write the masked scores. For each row, list the allowed sources and the denominator used by softmax.
5. Calculate  $A$  exactly as fractions. Verify the row sums and the entries above the diagonal.

- Expand  $z_3$  as three scaled value vectors. Check  $z_1$  and  $z_2$  too.
- Explain why applying softmax across destinations rather than sources computes a different operation.

Now deliberately make a masking error on paper: apply softmax to every score in each row, then zero the future-position coefficients without renormalizing. Compute its row sums and output. Does a lower-triangular final matrix guarantee that the computation respected causality?

Do not correct your original prediction by silently overwriting it. If you discover an error, identify the first intermediate quantity that went wrong.

## 4 Part 2 — Predict controlled changes

Restart from the original projected matrices for every case. **Each intervention takes place after projection**, and changes only what is named. Do not change  $X$  as a shortcut: that could change queries, keys, and values together.

For each case, predict which rows of  $A$  change, which rows of  $Z$  change, and the exact new  $z_3$ .

- Value ablation:** replace  $v_2 = [0, 4]$  with  $[0, 0]$ . Leave  $Q$ ,  $K$ , and the mask unchanged. Do not renormalize  $A$ .
- Query perturbation:** replace only  $q_3$  with  $[1, 0]$ . Keep  $K$ ,  $V$ , and the mask fixed, and recompute attention coefficients.
- Connection removal:** forbid only the connection from destination 3 to source 2. Leave the other mask entries, scores, and values unchanged. Recompute softmax over the remaining sources. This does not delete token 2 from the sequence.
- Future-key perturbation:** replace  $k_3$  with twice its original vector, leaving  $Q$  and  $V$  unchanged. Predict whether any correct output at positions 1 or 2 can change. Also predict the result for position 1 in the deliberately buggy masking procedure from Part 1.

Before checking the answers, explain why value ablation and connection removal should not generally produce the same result.

## 5 Part 3 — Implement small arrays

### 5.1 Learner scaffold

Copy this into `tiny_attention.py`. The unfinished functions deliberately raise errors. Implement them using lists, loops, sum, and functions from `math`.

```
import math

X = [[1.0, 0.0], [0.0, 1.0], [1.0, 1.0]]
C = math.sqrt(2.0) * math.log(2.0)
WQ = [[1.0, 0.0], [0.0, 1.0]]
WK = [[C, 0.0], [0.0, C]]
WV = [[2.0, 0.0], [0.0, 4.0]]
ALLOWED = [[True, False, False],
            [True, True, False],
```

```

[True, True, True]]

def matmul(left, right):
    # Reject empty, ragged, or incompatible matrices.
    # Compute every output entry as a row-column dot product.
    raise NotImplementedError("Implement checked matrix multiplication")

def masked_softmax(scores, allowed):
    # Check matching nonzero lengths and at least one allowed source.
    # Reject nonfinite input scores and non-boolean mask entries.
    # Subtract the largest ALLOWED score before exponentiation.
    # A blocked source contributes exactly zero.
    raise NotImplementedError("Implement stable masked softmax")

def attention(Q, K, V, allowed):
    # Q/K widths must agree; K/V must have the same source count.
    # allowed must have one row per query and one column per key.
    # Build scaled scores, masked scores, row-wise weights, and Z.
    # Return a dict with scores, masked_scores, weights, and output.
    raise NotImplementedError("Implement the attention trace")

# Q = matmul(X, WQ); K = matmul(X, WK); V = matmul(X, WV)
# Print attention(Q, K, V, ALLOWED) after completing the functions.

```

The allowed convention is explicit: True permits a source. Some framework APIs use the opposite convention for a padding mask. This Lab does not import a framework or inherit its mask semantics.

Use fresh nested-list copies for interventions, such as `[row[:] for row in V]`. A shallow outer-list copy alone still shares the inner rows. Make sure later edits cannot change a previously recorded baseline.

## 5.2 Run and compare

Run your completed file with an appropriate command for your existing installation, such as `python3 tiny_attention.py`. Record the command and Python version.

Print all four returned arrays. Compare the masked pattern exactly and finite numerical values within an absolute tolerance of  $10^{-9}$ . Floating-point approximations to  $\ln 2$  and exponentiation need not produce exact fractions. Do not compare an expression such as infinity minus infinity to measure error; compare blocked positions separately.

Then run the four interventions. Record the observed differences alongside your predictions. Finally rerun the unchanged baseline to detect accidental mutation.

Your implementation should reject:

- a ragged matrix;
- incompatible query and key widths;
- different key and value source counts;
- a mask with the wrong dimensions;
- a row with every source blocked;
- nonfinite supplied query, key, value, or score entries.

This deliberately narrow implementation rejects all-masked rows instead of inventing a fallback. Production code may have additional policies, data types, batching, or numerical constraints; those are outside this Lab.

### 5.3 Hand-worked alternative

Calculate every intervention independently, using fractions where possible. Check the answer once by expanding sums and once by matrix multiplication. For the future-key perturbation, you need not recompute forbidden terms to prove earlier outputs unchanged in the correct computation. Record that these are analytical results rather than program observations.

## 6 Part 4 — Explain what changed

Write a short interpretation addressing all of the following:

1. Which intervention changed an output while preserving the entire attention map?
2. Which interventions changed only the third row of the map?
3. Why did connection removal redistribute weight while value ablation did not?
4. How can the buggy procedure leak a future key's information when its final coefficients above the diagonal are zero?
5. What did these experiments establish about this isolated head? What additional evidence would be needed to claim that a head explains a trained model's final answer?

A useful conclusion names the intervention, controlled quantities, and measured outcome. Avoid calling a coefficient a token's "importance" without specifying the meaning and test.

#### **i** Analytically derived answer key

These values follow from the equations. They are reference expectations, not a report of an executed learner program. Compare them only after recording your attempt.

**Shapes:**  $X, Q, K, V, Z$  are  $3 \times 2$ ; projection matrices are  $2 \times 2$ ;  $S, A, M$  are  $3 \times 3$ .

**Baseline:**  $Q = X$ ,  $K = \sqrt{2} \ln(2) X$ ,  $V = [[2, 0], [0, 4], [2, 4]]$ . The scaled scores are  $\ln(2)[[1, 0, 1], [0, 1, 1], [1, 1, 2]]$ . The causal softmax denominators are 2, 3, and 8 when using the unshifted exponentials.

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 1/3 & 2/3 & 0 \\ 1/4 & 1/4 & 1/2 \end{bmatrix}, \quad Z = \begin{bmatrix} 2 & 0 \\ 2/3 & 8/3 \\ 3/2 & 3 \end{bmatrix}.$$

**Buggy post-softmax masking:** coefficients are  $[[2/5, 0, 0], [1/5, 2/5, 0], [1/4, 1/4, 1/2]]$ . Row sums are  $2/5, 3/5$ , and  $1$ ; outputs are  $[[4/5, 0], [2/5, 8/5], [3/2, 3]]$ .

**Value ablation:**  $A$  is unchanged. The new outputs are  $[[2, 0], [2/3, 0], [3/2, 2]]$ . At position 3 the change is exactly  $-A_{32}v_2 = [0, -1]$ .

**Query perturbation:** only row 3 changes. Its scaled scores are  $[\ln 2, 0, \ln 2]$ , its weights are  $[2/5, 1/5, 2/5]$ , and its output is  $[8/5, 12/5]$ .

**Connection removal:** only row 3 changes. Its weights become  $[1/3, 0, 2/3]$ , giving  $z_3 = [2, 8/3]$ . This differs from value ablation because the remaining values receive larger coefficients.

**Future-key perturbation:** correct outputs at positions 1 and 2 are unchanged. Row 3 has exponentials  $[2, 2, 16]$ , weights  $[1/10, 1/10, 4/5]$ , and output  $[9/5, 18/5]$ . In the buggy procedure, row 1's full denominator changes from  $2 + 1 + 2 = 5$  to  $2 + 1 + 4 = 7$ , so its output changes from  $[4/5, 0]$  to  $[4/7, 0]$  despite having its future coefficients zeroed afterward.

## 7 Completed reference implementation

This is a separate, complete reference listing for comparison after your own attempt. It expresses the intended algorithm; its presence does not certify a particular Python environment or substitute for recording actual execution. It omits automatic differentiation, batching, and optimized kernels.

```
import math
from numbers import Real

def matrix_shape(matrix, name):
    if not matrix or not matrix[0]:
        raise ValueError(f"{name} must be nonempty")
    width = len(matrix[0])
    for row in matrix:
        if len(row) != width:
            raise ValueError(f"{name} is ragged")
        if any(not isinstance(x, Real) or not math.isfinite(x)
               for x in row):
            raise ValueError(f"{name} needs finite real entries")
    return len(matrix), width

def matmul(left, right):
    rows, inner = matrix_shape(left, "left")
    right_rows, columns = matrix_shape(right, "right")
    if inner != right_rows:
        raise ValueError("matrix dimensions do not align")
    return [[sum(left[i][k] * right[k][j]
                  for k in range(inner))
             for j in range(columns)]
            for i in range(rows)]
```

```

        for i in range(rows)]

def masked_softmax(scores, allowed):
    if not scores or len(scores) != len(allowed):
        raise ValueError("scores and mask must have matching lengths")
    if any(type(a) is not bool for a in allowed):
        raise ValueError("allowed entries must be boolean")
    if not any(allowed):
        raise ValueError("at least one source must be allowed")
    if any(not isinstance(s, Real) or not math.isfinite(s)
           for s in scores):
        raise ValueError("input scores must be finite")
    shift = max(s for s, a in zip(scores, allowed) if a)
    terms = [math.exp(s - shift) if a else 0.0
              for s, a in zip(scores, allowed)]
    denominator = sum(terms)
    return [term / denominator for term in terms]

def attention(Q, K, V, allowed):
    nq, dk = matrix_shape(Q, "Q")
    nk, key_width = matrix_shape(K, "K")
    nv, _ = matrix_shape(V, "V")
    if dk != key_width:
        raise ValueError("Q and K widths must agree")
    if nk != nv:
        raise ValueError("K and V source counts must agree")
    if len(allowed) != nq or any(len(row) != nk for row in allowed):
        raise ValueError("mask dimensions must be queries by sources")
    scale = math.sqrt(dk)
    scores = [[sum(Q[i][k] * K[j][k] for k in range(dk)) / scale
                for j in range(nk)] for i in range(nq)]
    weights = [masked_softmax(row, mask)
                for row, mask in zip(scores, allowed)]
    masked_scores = [[s if a else float("-inf")
                      for s, a in zip(row, mask)]
                     for row, mask in zip(scores, allowed)]
    output = matmul(weights, V)
    return {"scores": scores, "masked_scores": masked_scores,
            "weights": weights, "output": output}

def baseline_inputs():
    X = [[1.0, 0.0], [0.0, 1.0], [1.0, 1.0]]
    c = math.sqrt(2.0) * math.log(2.0)
    WQ = [[1.0, 0.0], [0.0, 1.0]]
    WK = [[c, 0.0], [0.0, c]]

```

```

WV = [[2.0, 0.0], [0.0, 4.0]]
allowed = [[True, False, False],
           [True, True, False],
           [True, True, True]]
return matmul(X, WQ), matmul(X, WK), matmul(X, WV), allowed

if __name__ == "__main__":
    Q, K, V, allowed = baseline_inputs()
    baseline = attention(Q, K, V, allowed)
    for name, rows in baseline.items():
        print(name)
        for row in rows:
            print(row)

```

For example, the value-ablation case uses fresh inputs, sets  $V[1] = [0.0, 0.0]$ , and calls `attention` again. Position 2 has index 1 in Python. The query case sets  $Q[2] = [1.0, 0.0]$ ; connection removal sets  $allowed[2][1] = False$ ; the future-key case sets  $K[2] = [2.0 * x \text{ for } x \text{ in } K[2]]$ . Start with `baseline_inputs()` before each case.

## 8 Optional checks with explanatory value

**A shared value shift.** Add  $c = [1, -2]$  to every value vector, holding scores and masks fixed. Predict the change in each output. Since the weights sum to one, every correct output increases by  $c$ . The buggy implementation increases row  $i$  by its surviving coefficient sum times  $c$ . This tests normalization through output behavior.

**Different maps, equal mixtures.** Use separate value vectors  $[[2, 0], [2, 0], [4, 0]]$ . Compare row weights  $[1/4, 1/4, 1/2]$  and  $[3/8, 1/8, 1/2]$ . Both yield  $[3, 0]$  because the first two values are identical. This is an explicitly constructed counterexample, not a claim that arbitrary changes to every attention map leave output unchanged.

**Score offsets.** Add 1000 to every finite score in one row before masking and softmax. A numerically stable implementation should preserve that row's weights within tolerance. Explain why multiplying every score by 1000 would be a different operation.

## 9 Completion criteria

Your submission should let another learner check:

- the original hand trace and predictions;
- correct normalization and causal masking, including the deliberately buggy comparison;
- the four independent interventions and actual outputs or clearly labeled analytical checks;
- rejection of the listed invalid inputs for the coding route;
- preservation of the baseline after interventions;
- an explanation distinguishing mixture coefficients from transmitted values;
- a conclusion limited to the computation you actually investigated.

Keep your first attempt, corrections, and remaining questions. A useful result is a trace you can explain, including why a mistaken implementation produced its particular wrong answer.