

Trace and Perturb Expert Routing

Gavin · AI-assisted course design

Contents

1	Goal	3
2	The fixed two-layer toy	3
2.1	Layer 0	4
2.2	Layer 1	4
3	Part 1 — Compute a baseline trace	4
4	Part 2 — Specify and predict interventions	5
5	Part 3 — Implement and save the trace	5
5.1	Data and learner scaffold	5
5.2	Verification checks	7
6	Part 4 — Inspect one real configuration without loading a model	7
7	Reference functions	9
7.1	Running the supplied completed reference	11
8	Solutions and interpretation	11
8.1	Baseline	11
8.2	Interventions at position 0	12
8.3	Parameter accounting	12
9	Optional future extension — Trace a trained model	13
10	Submission	13
11	More Learning	13

1 Goal

Produce a routing log that connects each token position and layer to router logits, selected experts, combining coefficients, expert outputs, and the residual update. Then distinguish four interventions and inspect a small, pinned gpt-oss configuration.

Prerequisite: Lesson 2.5 — Mixture of Experts, through the worked example. Read its accounting and interpretation sections before drawing conclusions.

Time: about 90–120 minutes. Allow another 20 minutes for the optional extension.

Requirements: paper or a text editor. The coding route uses an existing Python 3 installation, a CPU, and the standard library. The toy requires no network, model weights, API, account, package installation, GPU, or paid compute. Configuration inspection needs only a public text file; an attributed excerpt is supplied for an offline route.

Expected artifact: `submission.md`, a routing log, and an exact run specification. If coding, save `tiny_moe.py`, its generated `routing-log.jsonl`, and `run-spec.json`. If working by hand, reproduce the same information in a legible table. Keep predictions separate from results and distinguish hand-derived values from actual program output.

! Predict before executing

Complete Parts 1 and 2 before running the code or reading the solution. The scaffold intentionally raises errors. Compare your implementation with the reference functions only after attempting it.

2 The fixed two-layer toy

This is an invented arithmetic system. It has no tokenizer, attention, normalization, trained weights, vocabulary output head, or language-generation ability. We call its three indexed inputs “token positions” to practise the logging convention used for a real model. Its token IDs are arbitrary labels, not entries in a published tokenizer.

Each layer has four experts and selects two. Vectors are columns mathematically; Python stores their coordinates in ordinary lists. The experts in this Lab are linear maps rather than full MLPs. Every layer computes

$$z = W_r x + b_r, \quad S = \text{TopKIndices}(z, 2), \quad \alpha_S = \text{softmax}(z_S),$$

$$f_e(x) = A_e x, \quad m = \sum_{e \in S} \alpha_e f_e(x), \quad x_{\text{next}} = x + m.$$

Sort by descending logit, breaking equal scores by ascending expert ID. Selected-only softmax is mandatory. There is no router noise, expert capacity limit, token dropping, or parameter sharing between layers. Some specified coefficients are equal or zero; those are still separate stored coefficient slots.

Use these inputs:

Position	Toy token ID	Initial vector
0	11	$[1, 1]^T$
1	11	$[1, 0]^T$
2	17	$[0, 1]^T$

The repeated ID with different vectors is an explicit fixture. We are not simulating the context that could have produced those vectors in a trained model.

2.1 Layer 0

Use the lesson's router:

$$W_r^{(0)} = \begin{bmatrix} \ln 4 & 0 \\ 0 & \ln 3 \\ \ln 2 & 0 \\ 0 & 0 \end{bmatrix}, \quad b_r^{(0)} = [0, 0, 0, 0]^T.$$

Its experts are

$$\begin{aligned} f_0(x) &= [2x_1, 0]^T, & f_1(x) &= [0, 3x_2]^T, \\ f_2(x) &= [-x_1, x_2]^T, & f_3(x) &= [x_2, x_1]^T. \end{aligned}$$

2.2 Layer 1

Feed each layer-0 residual output into this layer. Use a zero 4×2 router matrix and

$$b_r^{(1)} = [0, \ln 2, \ln 4, 0]^T.$$

This deliberately input-independent router makes the second hand calculation small. Its expert functions differ from layer 0:

$$\begin{aligned} f_0(x) &= [x_2, 0]^T, & f_1(x) &= [x_1, 0]^T, \\ f_2(x) &= [0, 2x_2]^T, & f_3(x) &= [-x_1, -x_2]^T. \end{aligned}$$

Layer 1's expert 1 is therefore not layer 0's expert 1. Keep the layer index in every record.

3 Part 1 — Compute a baseline trace

For every position in layer 0:

1. Compute all four logits and the full softmax distribution p .
2. Rank the experts, naming any tie and its resolution.
3. Record the two selected IDs and their selected-only coefficients.
4. Evaluate only those two expert functions.
5. Compute the combined update and residual output.

Then compute layer 1 using the actual layer-0 outputs. There should be six routing records. Each record must contain:

- position, toy token ID, and layer;
- input vector and full router logits;
- full-router probabilities;
- selected expert IDs in rank order and selected-only coefficients;
- expert outputs before any output intervention;
- expert outputs actually combined;
- combined update and residual output;
- intervention name, or baseline.

Predict first: will the two occurrences of toy token 11 select the same experts in layer 0? Will a selected expert with a zero output disappear from the selection log? Can different expert outputs produce different updates while the route and coefficients remain fixed?

For each layer, count **assignments**, including both selected experts for every position. Explain why the counts sum to six, rather than three or one.

4 Part 2 — Specify and predict interventions

Restart from the original inputs for each scenario. Apply the intervention **only at layer 0, position 0**. All other positions and the layer-1 computation retain their baseline rules. Continue the altered residual vector through layer 1.

1. **Prohibit expert 0.** Remove ID 0 from the eligible set before selecting the best two remaining experts. Renormalize over those selected experts. Do not zero its output after ordinary selection.
2. **Force experts 2 and 3.** Override the selected set with exactly those IDs, retaining their original logits to calculate the selected-only weights. This is not equal weighting.
3. **Boost expert 2.** Add $\ln 4$ to only its router logit before selection. Recompute the ranking and selected-only weights. Leave expert matrices unchanged.
4. **Ablate expert 0's output.** Preserve baseline selection and coefficients. Replace only its selected output vector with zero. Do not renormalize or substitute another expert.

For each scenario, predict the selected IDs, coefficients, layer-0 update, and both residual outputs. Say which later router logits will change in **this particular toy**, and why. Do not generalize from layer 1's constant router to trained routers.

A control: add the same constant to every router logit. Predict what changes in raw scores, full probabilities, selection, and combined output.

5 Part 3 — Implement and save the trace

5.1 Data and learner scaffold

Copy the following into `tiny_moe.py`. Complete the three numerical functions; then follow the tracing recipe below.

```
import json
import math
import sys
```

```

from pathlib import Path

SPEC_ID = "tiny-moe-routing-v1"
K = 2
INPUTS = [
    {"position": 0, "token_id": 11, "x": [1.0, 1.0]},
    {"position": 1, "token_id": 11, "x": [1.0, 0.0]},
    {"position": 2, "token_id": 17, "x": [0.0, 1.0]},
]
LAYERS = [
    {
        "router_W": [
            [math.log(4), 0.0], [0.0, math.log(3)],
            [math.log(2), 0.0], [0.0, 0.0],
        ],
        "router_b": [0.0, 0.0, 0.0, 0.0],
        "experts": [
            [[2.0, 0.0], [0.0, 0.0]],
            [[0.0, 0.0], [0.0, 3.0]],
            [[-1.0, 0.0], [0.0, 1.0]],
            [[0.0, 1.0], [1.0, 0.0]],
        ],
    },
    {
        "router_W": [[0.0, 0.0] for _ in range(4)],
        "router_b": [0.0, math.log(2), math.log(4), 0.0],
        "experts": [
            [[0.0, 1.0], [0.0, 0.0]],
            [[1.0, 0.0], [0.0, 0.0]],
            [[0.0, 0.0], [0.0, 2.0]],
            [[-1.0, 0.0], [0.0, -1.0]],
        ],
    },
]

def matvec(matrix, vector):
    """Return matrix @ vector without mutating either input."""
    raise NotImplementedError("Compute a dot product for each row")

def softmax(values):
    """Subtract the largest input before exponentiating."""
    raise NotImplementedError("Return positive weights summing to one")

def choose(logits, eligible, k):

```

```
"""Descending score, then ascending ID; return exactly k IDs."""  
raise NotImplementedError("Validate the eligible count and sort")
```

The tracing function should perform these steps, in this order:

1. Calculate the raw affine router logits and preserve a copy.
2. For `boost_2`, add $\ln 4$ to logit 2. Preserve these effective logits separately.
3. Compute the diagnostic full softmax over all effective logits. In `prohibit_0`, this diagnostic still includes expert 0; the eligibility restriction affects selection, not this recorded diagnostic.
4. Set eligible IDs to `[1, 2, 3]` for `prohibit_0`, `[2, 3]` for `force_23`, and `[0, 1, 2, 3]` otherwise. Select two by the stated ranking rule.
5. Compute a new softmax on the selected effective logits only.
6. Calculate and save the selected expert outputs. For `ablate_0`, replace expert 0's output with zero in a separate copy used for combination.
7. Sum the weighted vectors, add the input residual, and return the complete record.

Run all three positions through both layers for each of five scenarios: baseline and the four interventions. Apply a scenario's intervention only to position 0 in layer 0. Keep the overall scenario label on the downstream records even when their local intervention is baseline.

Save one JSON object per line in `routing-log.jsonl`. The complete run should have 30 records. Save the input fixtures, layer coefficients, top- k , normalization convention, tie rule, scenario list, and Python version in `run-spec.json`. Save the source file too: floating-point values alone do not document every rule of the experiment.

5.2 Verification checks

Do not rely only on a final vector matching a reference. Check:

- four full probabilities sum to one for each record;
- two distinct, eligible expert IDs are selected;
- two combining coefficients sum to one;
- combining weights are paired with the correct expert outputs;
- the combined update reproduces the weighted sum;
- the output is the input plus that update;
- layer-1 inputs equal the corresponding layer-0 outputs in every scenario;
- all scenarios leave positions 1 and 2 unchanged;
- the zero-output selected expert at baseline layer 0, position 2 remains in the log;
- repeated runs give the same records, apart from deliberately recorded execution metadata.

Use a small floating-point tolerance, such as 10^{-10} , when comparing numerical results. Our exact fractions describe the mathematical answers; binary floating point may store nearby values.

6 Part 4 — Inspect one real configuration without loading a model

Open only the pinned `gpt-oss-20b config.json`. The repository is `openai/gpt-oss-20b`; the revision is `f81fef1ddd90d214968e951a76834f1ded130a18`. Reading the file in a browser is sufficient. Down-

loading this one small JSON file is also sufficient. Do not run model-loading examples or download weight shards for this Lab.

For an offline route, use this **selected-field transcription** from that pinned file. It is not the complete configuration or an independently observed local download:

```
{
  "model_type": "gpt_oss",
  "hidden_size": 2880,
  "intermediate_size": 2880,
  "num_hidden_layers": 24,
  "num_local_experts": 32,
  "num_experts_per_tok": 4,
  "experts_per_token": 4,
  "output_router_logits": false,
  "tie_word_embeddings": false,
  "vocab_size": 201088
}
```

Record whether you inspected the complete pinned file or used the excerpt. If you save the complete file, keep its source URL, revision, retrieval date, and SHA-256 digest. A digest documents the bytes you retained; an unverified digest does not independently prove publisher identity. There is no need to install a model library merely to read JSON.

Answer:

1. How many layer-specific expert networks exist across all 24 MoE layers? How many expert evaluations lie on one token's full forward path?
2. What percentage of expert networks are selected at each layer? Why is this not the model's whole active-parameter percentage?
3. Using three $d \times m$ -equivalent matrices per gated expert, calculate total and selected expert-matrix coefficients. Explicitly exclude biases and non-expert parameters.
4. Why can the configuration not tell you which expert a particular real prompt selects?
5. What additional evidence would be required to claim that router logits were actually captured?

Compare your counts with Table 1 of the gpt-oss model card. Record its published 20b totals and its active-count convention. Do not equate "32 experts per layer" with "32 complete language models."

For the toy, count stored coefficient slots directly: each layer has a 4×2 router matrix, four router biases, and four 2×2 expert matrices. Count zeros as stored coefficients. How many slots are total? How many belong to the router and selected expert matrices along one token's two-layer path? The calculation assumes generic dense evaluation of each selected matrix, even when it contains zeros or produces zero output.

Finally, estimate a hypothetical packed float32 storage size for all toy coefficients. Explain why that number is not the actual memory consumption of Python lists, nor the memory consumption of a trained-model runtime.

7 Reference functions

Compare only after attempting the scaffold and tracing recipe. These functions implement the toy specification; their output is not trained-model evidence. Retain your own run output when you execute them.

```
def matvec(matrix, vector):
    return [sum(a * b for a, b in zip(row, vector))
            for row in matrix]

def softmax(values):
    peak = max(values)
    exp_values = [math.exp(value - peak) for value in values]
    total = sum(exp_values)
    return [value / total for value in exp_values]

def choose(logits, eligible, k):
    if len(set(eligible)) != len(eligible) or len(eligible) < k:
        raise ValueError("Need at least k distinct eligible experts")
    return sorted(eligible, key=lambda e: (-logits[e], e))[:k]

def trace_one(x, layer_id, position, token_id, scenario):
    local = scenario if layer_id == 0 and position == 0 else "baseline"
    layer = LAYERS[layer_id]
    projection = matvec(layer["router_W"], x)
    raw = [a + b for a, b in zip(projection, layer["router_b"])]
    logits = raw[:]
    if local == "boost_2":
        logits[2] += math.log(4)

    eligible = list(range(4))
    if local == "prohibit_0":
        eligible = [1, 2, 3]
    elif local == "force_23":
        eligible = [2, 3]
    selected = choose(logits, eligible, K)
    weights = softmax([logits[e] for e in selected])
    original = [matvec(layer["experts"][e], x) for e in selected]
    used = [vector[:] for vector in original]
    if local == "ablate_0":
        used[selected.index(0)] = [0.0, 0.0]
    update = [sum(w * vector[j] for w, vector in zip(weights, used))
              for j in range(2)]
    output = [a + b for a, b in zip(x, update)]
    return {
        "scenario": scenario, "local_intervention": local,
```

```

        "position": position, "token_id": token_id,
        "layer": layer_id, "input": x[:,],
        "raw_logits": raw, "effective_logits": logits,
        "full_probabilities": softmax(logits),
        "eligible": eligible, "selected": selected,
        "combine_weights": weights,
        "expert_outputs_original": original,
        "expert_outputs_used": used,
        "update": update, "output": output,
    }

SCENARIOS = ["baseline", "prohibit_0", "force_23",
             "boost_2", "ablate_0"]
records = []
for scenario in SCENARIOS:
    for item in INPUTS:
        x = item["x"][:,]
        for layer_id in range(len(LAYERS)):
            record = trace_one(x, layer_id, item["position"],
                              item["token_id"], scenario)
            records.append(record)
            x = record["output"][:,]

spec = {
    "spec_id": SPEC_ID, "python": sys.version,
    "top_k": K, "layers": LAYERS, "inputs": INPUTS,
    "tie_rule": "descending logit then ascending expert ID",
    "combine_rule": "softmax over selected effective logits",
    "normalization": "none", "attention": "none",
    "capacity_limit": None, "scenarios": SCENARIOS,
    "intervention_site": {"layer": 0, "position": 0},
}
Path("run-spec.json").write_text(
    json.dumps(spec, indent=2) + "\n", encoding="utf-8")
Path("routing-log.jsonl").write_text(
    "".join(json.dumps(row) + "\n" for row in records),
    encoding="utf-8")
print("Saved", len(records), "routing records")

```

Keep the numerical definitions from the first block and replace its unfinished functions with these completed functions. The code writes files in the directory where you run it; use a new Lab directory so you do not replace an earlier result accidentally. Run `python3 tiny_moe.py`, then perform the verification checks on the saved records. Generating a file is not itself evidence that those checks passed.

7.1 Running the supplied completed reference

After attempting the scaffold, save `tiny_moe.py`. It uses the exact numerical functions and fixtures above with an output-directory wrapper. It was tested with CPython 3.13.13 and requires no packages or network.

```
python3 tiny_moe.py --output moe-run-01
```

Inspect all 30 actual records in `moe-run-01/routing-log.jsonl` and its `run-spec.json`. The wrapper refuses a nonempty directory. Choose a new directory for another retained run; use `--overwrite` only to replace the two named result files deliberately. Raw/effective logits and original/combined expert outputs are distinct fields. Preserve failed predictions in your submission.

The optional text-only source helper described in Lab 08 retrieves the pinned `gpt-oss` config together with four small architecture references; it never loads a model. Its verified config is 1,806 bytes with SHA-256 `3a2a26ded679375b7928ddeca59764df7cea83220c1961035f6d6e232659e9ce`. Record whether you used this actual retrieval, a browser view or the supplied excerpt.

8 Solutions and interpretation

8.1 Baseline

For layer 0, the exact results are:

Position	Full probabilities in ID order	Selected IDs	Combining weights
0	$[4, 3, 2, 1]/10$	$[0, 1]$	$[4/7, 3/7]$
1	$[4, 1, 2, 1]/8$	$[0, 2]$	$[2/3, 1/3]$
2	$[1, 3, 1, 1]/6$	$[1, 0]$	$[3/4, 1/4]$

The corresponding layer-0 vectors are:

Position	Update	Residual output
0	$[8/7, 9/7]$	$[15/7, 16/7]$
1	$[1, 0]$	$[2, 0]$
2	$[0, 9/4]$	$[0, 13/4]$

At position 2, expert 1 wins; experts 0, 2, and 3 tie for the second slot. The specified tie rule selects 0, whose output is zero for this input. It still receives a routing assignment. The layer-0 counts are $[3, 2, 1, 0]$.

Layer 1 has full probabilities $[1, 2, 4, 1]/8$ for every input and selects $[2, 1]$ with weights $[2/3, 1/3]$. For input v , its update is $[v_1/3, 4v_2/3]^T$, and its residual output is $[4v_1/3, 7v_2/3]^T$.

Its final outputs at positions 0, 1, and 2 are respectively

$$[20/7, 16/3]^T, \quad [8/3, 0]^T, \quad [0, 91/12]^T.$$

Its assignment counts are $[0, 3, 3, 0]$. The toy IDs do not determine routing independently of the input vectors.

8.2 Interventions at position 0

The following are hand-derived predictions to check against your run:

Scenario	Selected IDs	Weights	Layer-0 update
Prohibit 0	$[1, 2]$	$[3/5, 2/5]$	$[-2/5, 11/5]$
Force 2 and 3	$[2, 3]$	$[2/3, 1/3]$	$[-1/3, 1]$
Boost 2	$[2, 0]$	$[2/3, 1/3]$	$[0, 2/3]$
Ablate output 0	$[0, 1]$	$[4/7, 3/7]$	$[0, 9/7]$

Residual outputs include the bypass additions:

Scenario	After layer 0	After layer 1
Prohibit 0	$[3/5, 16/5]$	$[4/5, 112/15]$
Force 2 and 3	$[2/3, 2]$	$[8/9, 14/3]$
Boost 2	$[1, 5/3]$	$[4/3, 35/9]$
Ablate output 0	$[1, 16/7]$	$[4/3, 16/3]$

Layer 1’s logits, routes, and combining weights remain fixed because its router matrix is zero. Its expert outputs and residual outputs change. In a trained network with input-dependent routers, an earlier intervention can change later routes too. Our system contains no cross-position mixing, so it also cannot demonstrate how attention would spread an intervention across positions.

Adding a common constant changes the raw logits but none of the probabilities, rankings, or vector outputs in exact arithmetic. Subtracting the largest logit before exponentiation exploits this identity for numerical stability.

8.3 Parameter accounting

The 20b architecture has $24 \times 32 = 768$ layer-specific experts and $24 \times 4 = 96$ expert evaluations along one token’s forward path. Four of 32 is 12.5%. This fraction applies to equal-sized routed expert banks, not the whole model.

Ignoring biases and non-expert components, the gated expert-matrix totals are 19,110,297,600 stored coefficients and 2,388,787,200 selected coefficients per token path. The model card’s whole-model figures are 20.91 billion total and 3.61 billion active; it counts unembedding parameters but not embeddings as active. The differences are not errors in the multiplication: the quantities include different components.

The toy stores $8 + 4 + 16 = 28$ coefficient slots per layer, or 56 total. Each token uses the 12 router coefficients plus two four-coefficient expert matrices per layer: $2 \times (12 + 8) = 40$ active slots under

our stated convention. Hypothetical packed float32 storage is $56 \times 4 = 224$ bytes. Python objects, activations, and interpreter memory are excluded. Storing only the 40 active slots would not preserve all possible routes.

The gpt-oss config has no learned tensors or prompt activations. It cannot produce a trained-model routing trace. An actual trace requires a specific checkpoint, matching tokenizer and formatting, a specified implementation, instrumented intermediate tensors, and evidence of a real execution.

9 Optional future extension — Trace a trained model

This is a separate advanced activity, not a required download or setup step for this Lab. Before attempting it, choose a compatible checkpoint and runtime, estimate disk and memory requirements including trace storage, and establish whether the implementation exposes or permits instrumentation of router logits and selected indices.

A normal text-generation API may return no internal routing data. Do not infer expert selections from answer text, ask a model to invent its expert IDs, or relabel this toy's arrays as real measurements.

For a properly instrumented experiment, record checkpoint and code revisions, token IDs and positions, layer indices, prompt formatting, precision, routing rules, and whether tracing covers prompt processing or generation. Validate a small baseline before collecting a large dataset. Compare categories using matched lengths and controlled tokens, and distinguish selection frequency from output contributions. Forced routing can move representations outside an expert's usual training distribution; effects need controls and a defined outcome measure.

10 Submission

Include your original predictions, exact baseline calculations, saved or hand-worked routing records, configuration source and inspection method, parameter accounting, and an interpretation of no more than 300 words.

Explain one initial mistake or uncertainty and how you checked it. Conclude with one finding established by the toy and one claim about trained experts that it cannot establish. You have completed the core when another learner can reproduce your routes and outputs from the specification and understand the limits of your conclusion.

11 More Learning

1. **Pinned gpt-oss-20b configuration — OpenAI.** Identify the fields used in your accounting and the information missing for an actual routing trace.
2. **gpt-oss reference implementation — OpenAI, pinned commit.** Locate plausible measurement boundaries without loading or running the model.
3. **gpt-oss Model Card — OpenAI, version 1.** Compare your explicitly partial matrix count with the card's whole-model accounting.
4. **Mixtral of Experts — Jiang et al., Section 5.** Design one control that would strengthen an association between input categories and expert selection.