

Inspect Activations

Gavin · AI-assisted course design

Contents

1 Goal	3
2 Part 1 — Reuse a verified bounded environment	3
3 Part 2 — Fix the prompts and predictions	4
4 Part 3 — Capture named boundaries only	5
5 Part 4 — Validate before interpreting	6
6 Part 5 — Summarize geometry and candidate readouts	7
7 Part 6 — Explain what the core established	7
8 Optional — A tiny held-out probe	8
8.1 Declare the complete dataset first	8
8.2 Fix the readout and controls	8
9 Checkpoints and solutions	9
10 Submit a reproducible record	10
11 More Learning	10
12 Optional technical runner	11

1 Goal

Build a small, trustworthy model-inspection record. Capture named boundaries without altering inference, reconstruct final logits, and compare fixed prompts at an aligned token. Then apply the model's final readout to earlier residual states while keeping its limitations visible.

Prerequisites: Lesson 5.1, the residual, attention, and MLP lessons, and the small-model environment from Labs 05/07 or Lab 12.

Time: approximately 90 minutes for the core implementation and explanation; allow another 30–45 minutes for the optional probe.

Requirements: an existing local CPU environment and verified Pythia specimen. No inference API, GPU, account, credentials, paid compute, or new model is needed. Acquisition, if necessary, is a separate explicit setup step.

Expected artifact: a small inspection program, exact environment record, prompt manifest, numerical-check report, compact activation/candidate records, and `submission.md`. Do not attach model weights.

The core is complete after Parts 1–6. The optional probe teaches an additional method and is not required. This Lab specifies an experiment; it supplies no measured activation values, successful model predictions, or guaranteed probe performance. Distinguish analytical expectations from results actually obtained.

2 Part 1 — Reuse a verified bounded environment

Use exactly EleutherAI/pythia-14m at revision

94f7c35d5e9f2e9bac8ca839329f505b4d007d5d.

Reuse the verified six-file download and environment from the earlier inspection Labs. The tested reference package set is:

- CPython 3.13.13
- PyTorch 2.9.0
- Transformers 4.57.1
- tokenizers 0.22.1
- safetensors 0.6.2
- huggingface_hub 0.35.3

Record the actual installed versions, operating system, CPU architecture, and effective thread settings. A previously tested package set does not mean this new inspection program has already passed its checks. If the existing environment differs, document the difference and revalidate interfaces before comparing observations; do not silently install a second stack.

The allowlist remains `config.json`, `tokenizer.json`, `tokenizer_config.json`, `special_tokens_map.json`, `generation_config.json`, and `model.safetensors`. These six files total **30,264,046 bytes**, approximately 30.3 MB decimal. The weights occupy 28,143,920 bytes and have SHA-256

116a02532db461f91386a5b20f942ff2c8d4de7341e21b55caafc3d7b25f49a1.

Verify local artifacts against their saved manifest. The immutable publisher commit identifies the weight file. If acquisition is necessary, use the earlier Lab’s explicit selected-file procedure and 35,000,000-byte artifact ceiling. Do not fetch alternate weights, optimizer states, training checkpoints, or custom Python code. Software dependencies occupy additional space.

Require these controls:

- Built-in GPTNeoXForCausalLM and fast tokenizer, safetensors only, `trust_remote_code=False`.
- Load the verified local directory with `local_files_only=True`; the experiment itself runs offline.
- Explicit CPU, float32 floating parameters, `model.eval()`, and inference mode without gradients.
- `attn_implementation="eager"`, `use_cache=False`, `output_attentions=False`, and `output_hidden_states=False`.
- No compilation, autocast, quantization, base-model training, generated continuations, or automatic device placement.
- One intra-op and one inter-op CPU thread, configured before computation; record effective values.

Use `logits_to_keep=1` in the inspected version so the head need only return final-position logits. Check the returned shape. Keep contexts at most **96 tokens**, batch size one, and at most **40 distinct prompts**, including the optional probe. Stop rather than truncate an overlong prompt or enlarge the budget. Limit automated execution to **15 minutes** after setup and saved inspection data to **10 MB**, excluding dependencies and weights. These are bounds, not performance promises.

3 Part 2 — Fix the prompts and predictions

Use the following eight exact strings. Each `\n` denotes one actual newline. Do not add a chat template, role markers, or trailing whitespace.

1. The boat is red.\nColor:
2. The boat is blue.\nColor:
3. A red flag hangs by the door.\nColor:
4. A blue flag hangs by the door.\nColor:
5. The label reads red.\nColor:
6. The label reads blue.\nColor:
7. Mira copied the word red.\nColor:
8. Mira copied the word blue.\nColor:

Pairs are 1–2, 3–4, 5–6, and 7–8. The declared condition is which color word appears in the prefix. This is a lexical-context comparison; it does not isolate an abstract color concept. The final two pairs also differ in what the surrounding sentence says about that word.

Tokenize with `add_special_tokens=False`, no padding, and no truncation. Save each exact string, token IDs, decoded pieces, zero-based positions, length, and attention mask. Select each sequence’s final position $i = n - 1$. Assert that its token ID is identical across all eight prompts. An identical suffix is a reason to expect alignment, not permission to skip the assertion.

If alignment or length checks fail, stop before inspecting activations, correct the manifest, and record why. Do not search for attractive examples after viewing model outputs. Prefix lengths and absolute final positions may differ; preserve those differences as possible confounds.

Before the first forward pass, save predictions for these questions:

1. Should the final-position raw input to block 0 match across all prompts?
2. Should block 0's MLP contribution match? Why does the parallel architecture matter?
3. Must the attention contribution or later raw states match?
4. Must residual norms increase at every block?
5. Must intermediate lens candidates become steadily more plausible?
6. Which exact numerical agreements would validate the measurement without establishing a semantic explanation?

Also save the numerical tolerance, proposed summaries, and interpretation limits. Use absolute tolerance 10^{-6} and relative tolerance 10^{-5} for the core float32 equivalence checks. Report maximum absolute errors. If a check fails, investigate before changing tolerances; preserve the failed result and rationale for any revision.

4 Part 3 — Capture named boundaries only

Read the versioned implementation. Block indices are 0–5. Give the raw states names $r_0, \dots, r_6$: r_0 enters block 0, and $r_{\ell+1}$ leaves block ℓ . Give the final-normalized state the separate name h_f .

Register observation-only callbacks at these boundaries:

- r_0 : forward pre-hook on `gpt_neox.layers.0`, using its incoming hidden-state tensor.
- $r_{\ell+1}$: forward hook on `gpt_neox.layers.{ell}`, selecting the tensor in the block's returned tuple.
- a_ℓ : output of `gpt_neox.layers.{ell}.post_attention_dropout`, the attention contribution at its addition boundary.
- m_ℓ : output of `gpt_neox.layers.{ell}.post_mlp_dropout`, the MLP contribution at its addition boundary.
- h_f : output of `gpt_neox.final_layer_norm`.

Capture all six blocks' contributions for the core. The dropout-boundary names locate the writes precisely even though evaluation disables dropout behavior. Do not hook an attention-weight matrix and label it a_ℓ .

Each callback must validate the actual input or output structure, as appropriate, and assert that its selected tensor has source shape $(1, n, 128)$. The pre-hook selects the incoming hidden-state tensor; the block forward hook selects the tensor from its returned tuple. Retain only a detached, cloned copy of `[0, n-1, :]`, with stored shape $(128,)$. Every callback must return `None`, never replace or modify model values, and reject duplicate capture keys. Track one invocation per expected boundary. Bind layer indices correctly when constructing callbacks so every callback does not accidentally record the final loop index.

Keep prompt ID, token ID and position, layer, boundary, dtype, and shape with the vector. Do not retain full-sequence tensors, attention matrices, or vocabulary-sized scores for every layer. Remove

every handle in a `finally` block, including when assertions fail. Clear per-prompt state before the next example.

The implementation’s returned `hidden_states` interface does not provide seven uniformly raw boundaries: its last state follows final normalization. Our named hooks avoid that ambiguity. Verify the source rather than guessing from tuple length.

5 Part 4 — Validate before interpreting

For each core prompt, run three passes under the same settings:

1. An uninstrumented baseline.
2. The instrumented pass with all required hooks.
3. A fresh uninstrumented pass after hook removal.

Keep each final vocabulary-logit vector transiently for comparison. Assert expected shape $(1, 1, 50304)$ and finite values. Compare passes 1–2 and 1–3 with the declared tolerances. Save errors and pass/fail status; discard these full logit arrays after the checks and compact summaries are made.

Then perform the following checks on captured final-position vectors, with hooks already removed:

1. **Parallel addition:** for every block, verify

$$r_{\ell+1} = m_{\ell} + a_{\ell} + r_{\ell}.$$

Use the implementation’s addition order for the reference calculation. This is a selected-position identity, not a test of every token row.

2. **Final normalization:** apply the actual final LayerNorm module to r_6 and compare with captured h_f .
3. **Reconstruction from normalized state:** apply `embed_out` directly to captured h_f and compare with the instrumented pass’s final logits.
4. **Reconstruction from raw state:** apply final LayerNorm, then `embed_out`, to r_6 , and compare with the same logits.

Items 3 and 4 check distinct capture routes into the same output computation. Keep reconstruction operations inside inference mode. Removing hooks first also prevents these diagnostic module calls from being mistaken for additional captures.

Do not normalize h_f a second time. Do not project through `embed_in.weight`: this checkpoint’s input and output weights are untied. Verify each named module against the source and actual model.

Finally, compare r_0 and m_0 across all eight prompts. Both should agree within tolerance because the selected token ID is identical, position handling is rotary inside attention, and block 0’s parallel MLP reads its normalized input embedding before that block’s attention contribution is added. This is a structural prediction, not a supplied measured result. Later MLPs read contextualized states and need not match.

A failed invariant pauses interpretation. Report the prompt and boundary, shapes, maximum discrepancy, and likely instrumentation or runtime issue. “The model is mysterious” is not an explanation for a broken addition identity.

6 Part 5 — Summarize geometry and candidate readouts

For each raw state $r_0, \dots, r_6$, report its norm. For successive raw states, calculate the difference norm and cosine similarity. Keep the change $h_f - r_6$ in a separate normalization record; it is not another Transformer block update.

For every pair at every raw boundary, report:

$$\| r_{\ell, \text{red}} - r_{\ell, \text{blue}} \|_2, \quad \cos(r_{\ell, \text{red}}, r_{\ell, \text{blue}}).$$

Check finiteness and nonzero norms before computing cosine. Store undefined cases as an explicit status or JSON null, not zero. Preserve the native float32 measurements; you may calculate summary metrics from float64 copies if that reporting choice is stated. Casting a copy does not convert the original model execution into float64 inference.

For every raw state, compute the diagnostic lens

$$\tilde{z}_\ell = \text{embed_out}(\text{final_layer_norm}(r_\ell)).$$

Retain only the top five token IDs, escaped decoded pieces, and logits. Use smaller token ID to resolve exact ties. Compute a full-vocabulary vector transiently, select candidates, then discard it; do not save a layers-by-vocabulary cache. If an output ID lacks a tokenizer entry, preserve the ID and mark the missing piece rather than dropping or replacing it.

The r_6 lens must pass the final-logit check. Earlier rows are hypothetical readouts using the same normalization and head. Do not label them the model’s actual decisions at earlier layers. Do not renormalize the top five scores and call them full-model probabilities. Reporting logits is sufficient here.

Make one compact table or plot for geometry and another for candidate trajectories. Include uninteresting, repeated, or implausible candidates. Do not select only the pair or layer that looks most interpretable.

7 Part 6 — Explain what the core established

Write a short account with:

- One structural prediction and its actual check result.
- One measured paired difference, including prompt IDs and boundary.
- Two plausible explanations consistent with that difference.
- One additional experiment that could distinguish them, without running an unplanned search.
- One sentence separating a lens readout from actual model use.
- A statement of which evidence categories from Lesson 5.1 your work supports.

If all numerical checks pass but the model’s candidates are poor, the inspection still succeeds. If numerical checks fail, retain partial results and mark dependent interpretations as blocked. Do not fill the gap with invented or expected model outputs.

8 Optional — A tiny held-out probe

This extension asks a deliberately narrow question: can a fixed linear readout of one contextual state recover a synthetic color assignment across held-out wording families? It adds **32 prompts**, for a total of 40 distinct prompts with the core. It makes no positive performance guarantee.

8.1 Declare the complete dataset first

Use box names `cedar` and `maple`. Each prompt assigns red to one box and blue to the other. Generate the Cartesian product of:

- Four families below.
- Two assignments: cedar red/maple blue, or cedar blue/maple red.
- Two record orders: cedar first, or maple first.
- Two queried boxes: cedar, or maple.

This gives eight examples per family. The target label is the assigned color of the queried box: red is $+1$, blue is -1 . Both color words and both box names occur in every prompt. Append exactly `\nAnswer:` to each complete family text.

Keep a fixed row order: family F0–F3 outermost, then assignments in the order listed above, then record orders as listed, then queried boxes as listed innermost. Preserve that order within each split so saved permutation indices have an unambiguous meaning.

Write the family templates as follows. Construct two record clauses using the chosen order and assignment, then append the query sentence:

1. **F0:** clauses `The {box} box is {color}.` joined by one space; query `" What color is the {query} box?"`
2. **F1:** clauses `Box {box}: {color}.` joined by one space; query `" Give the color of box {query}."`
3. **F2:** clauses `A {color} tag marks {box}.` joined by one space; query `" Which tag color belongs to {query}?"`
4. **F3:** clauses `Record {box} as {color}.` joined by one space; query `" Retrieve the color recorded for {query}."`

For example, one F0 string is `The cedar box is red. The maple box is blue. What color is the cedar box?\nAnswer:`. Its declared label is $+1$. This is dataset construction, not a model answer.

Before observing activations, save all texts, labels, family IDs, assignment, record order, query, tokenization, final position, and split. Confirm each family has four examples of each label and that the final token ID matches across the 32 examples. Use **F0 and F1 for training**, **F2 and F3 for held-out testing**. Never split neighboring variants randomly across train and test.

8.2 Fix the readout and controls

Capture only r_0 and r_3 , the input embedding boundary and the raw output of zero-indexed block 2. Choose these boundaries before observing performance. Run the already validated observation routine; keep only final-position vectors.

Use one fixed ridge-regression probe with regularization $\lambda = 1$. From training examples only, calculate each coordinate's mean and population standard deviation: divide its sum of squared deviations by

16, then take the square root (correction=0 in PyTorch, or ddof=0 in NumPy). Replace an exactly zero standard deviation with one. Apply those same transformations to held-out examples. Fit an unpenalized intercept and penalized weights by minimizing

$$\sum_{j \in \text{train}} (y_j - w^T x_j - b)^2 + \|w\|_2^2.$$

Solve the linear system rather than explicitly inverting a matrix. Predict red when the fitted score is at least zero, otherwise blue. Use float64 for this small regression and record that choice separately from model dtype. No hyperparameter search, layer search, repeated dataset rewriting, or held-out calibration is allowed.

Evaluate these prespecified controls:

1. The same ridge procedure on r_0 . Matching final token IDs make these features constant apart from numerical error.
2. A majority-label baseline, with red chosen on ties. Balanced held-out labels make its accuracy exactly 8/16 by construction.
3. Query-name and final-token-ID baselines. Each query name occurs equally with both labels; the final ID is constant. Neither alone identifies the answer in the balanced construction.
4. Twenty probes on r_3 with the training labels shuffled. Initialize one dedicated `torch.Generator(device="cpu").manual_seed(20261007)` as `rng`. For each trial, draw `torch.randperm(16, generator=rng, device="cpu")` and index the original ordered training-label vector by that permutation. Advance the same generator across all twenty trials without reseeding or other random draws. Keep features in their original row order and held-out labels unchanged; do not select a favorable shuffle. Save the PyTorch version and every permutation, then report all scores or their complete histogram, not merely the worst shuffled result.

Train preprocessing without held-out data. Reuse the same training-fitted preprocessing for the shuffled-label trials because it does not depend on labels. Save the feature boundary, preprocessing values, coefficients, training labels, permutation indices, and predictions. No new model passes are needed for shuffles.

Report held-out accuracy as a count out of 16, per-family errors, training accuracy, and control results. A small sample and twenty shuffles do not justify a sweeping statistical claim. Even above-control success could reflect positional or lexical relationships that transfer across these four templates. It supports limited decodability, not proof of abstract color understanding or causal use. Chance-level or inconsistent results are valid outcomes.

9 Checkpoints and solutions

i Structural expectations and interpretation checks

- Matching final token IDs predict matching r_0 and block-0 MLP contributions under these settings. Attention may introduce prefix dependence into r_1 ; later states and MLP contributions can differ.
- The core stores seven raw 128-coordinate vectors and one separately named final-normalized vector per prompt. Six pairs of branch contributions add twelve more 128-coordinate vectors. These 20 vectors require 10,240 raw float32 bytes per prompt before metadata and serialization overhead.
- Raw-state norms need not increase. Opposing updates can reduce the norm; final normalization must be treated separately.
- Intermediate lens candidates need not improve monotonically or resemble sensible text. The mandatory equality is at the final boundary, where the actual output computation is reconstructed.
- If hooks change logits beyond tolerance, a later plausible-looking interpretation does not repair the failed observation check.
- The optional training/test split is 16/16 examples. The embedding, majority, and token-ID controls cannot solve the balanced labels from their stated information alone. Their expected limitations do not determine the contextual probe's observed score.
- A successful contextual probe supplies a predictive association for this dataset. An intervention and appropriate controls would be additional evidence about whether the base model uses the relevant information.

10 Submit a reproducible record

Include predictions recorded before observation, the immutable checkpoint identity and verified manifest, actual runtime versions, exact commands, input manifest, named boundaries, cleanup/invariance checks, reconstruction errors, compact plots or tables, and your interpretation. Include optional-probe work only if performed, with all controls and failures. State **not run** when a stage was not executed.

Preserve all eight core prompts and all declared optional examples. Do not enlarge the dataset, model, capture size, or compute budget to obtain a more attractive story. A reproducible negative or limited result is a complete result.

11 More Learning

- Lesson 5.1 gives the geometry, readout definitions, and evidence taxonomy.
- Pinned GPT-NeoX source is the boundary map for this implementation.
- PyTorch hook reference explains callback outputs and removable handles.
- Tuned Lens motivates distinguishing an intermediate representation from the final readout's expected space.
- Control tasks for probes explains why raw probe accuracy is insufficient.

12 Optional technical runner

Run with the existing pinned environment: `python inspect_activations.py --artifacts PATH_TO_VERIFIED_PYTHIA --output NEW_DIRECTORY`. Add `--probe only` for the fixed optional extension. The runner uses local verified files and performs no acquisition. Download the following files into one directory: `inspect_activations.py`, `inspection_common.py`, `inspect_residual_stream.py`.