ChatGPT 5.2 Thinking

analyze_introspe...
Python

Read it through carefully, make sure you understand its intention, then evaluate the attached script for correctness and efficiency.

Thought for 2m 26s

I read the whole script. High-level intention is clear and sensible:

- Load **two direction sets** per layer: `d_self` (stated confidence) and `d_other` (other-confidence).
- Orthogonalize them per layer to form:
  - `d_introspection = d_self − proj(d_self on d_other)`
  - `d_surface = d_other − proj(d_other on d_self)`
- Then evaluate:
  1. **Predictive/associational strength** of each direction against each task's confidence signal (heatmap).
  2. **Causal steering**: push activations along each direction and see how the confidence signal changes.
  3. **Causal ablation**: remove the direction and see how correlation between confidence and an uncertainty metric (e.g. logit_gap) changes, compared to orthogonal random controls.

That intention matches the module docstring and the outputs you save.

Correctness issues (important)

1) "Predictive power" is *not* out-of-sample prediction (it's in-sample association)

In `evaluate_direction_predictive_power()` you compute the OLS mapping (slope/intercept) **from the same test set you evaluate on** (via `corr`, means, stds of `projections` and `targets`), then report R^2 on that same set.

- This makes R^2 effectively a dressed-up **correlation-on-test** (and optimistic if you interpret it as “prediction”).
- If what you want is “association strength on the held-out set”, then it’s fine, but you should rename it (and you don’t need the OLS step; `R² = corr²` for simple linear regression with intercept).
- If you want genuine prediction, fit slope/intercept on `train_idx` , evaluate on `test_idx` .

Where: `evaluate_direction_predictive_power` around lines **403–452**, and it’s called with only `test_idx` in `compute_predictive_power_matrix` around **496–507**.

Fix (minimal):

- Pass `train_idx` too, compute slope/intercept on train projections/targets, apply to test projections, compute test R^2.

---

2) Reproducibility bug: `hash(dt)` is not stable across runs

You seed controls with:

python Copy code

```python
seed = SEED + layer * 1000 + hash(dt) % 1000
```

Python’s `hash()` is randomized per process unless `PYTHONHASHSEED` is fixed, so your controls can silently change run-to-run.

Where: steering controls seed around **616**, ablation controls seed around **908**.

Fix: use a fixed mapping, e.g.

python Copy code

```python
DT_SEED_OFFSET = {"d_self": 11, "d_other": 22, "d_introspection": 33, "d_s
seed = SEED + layer * 1000 + DT_SEED_OFFSET[dt]
```

---

3) Steering function ignores `metric_values` entirely

`run_steering_matrix(..., metric_values, ...)` never uses `metric_values`. Right now steering measures **mean confidence vs multiplier** (global shift), not “effect on correlation with uncertainty metric”.

That might be intended! But then `metric_values` should be removed from the signature to avoid confusion.

Where: `run_steering_matrix` signature at **0564–0572** and no subsequent use.

---

4) Single-token assumption for options is unchecked

You do:

```python
option_token_ids = [tokenizer.encode(opt, add_special_tokens=False)[0] for
```

If any option encodes to multiple tokens, you silently take only the first token → wrong confidence signal.

Where: steering **622–623**, ablation **919–921**.

Fix: assert `len(ids)==1` for each option (or handle multi-token options properly).

---

5) Degenerate-layer handling: “random orthogonal direction” seeds are hard-coded

In `orthogonalize_directions`, degenerate fallback uses RNG seeds `42` and `43`, independent of global `SEED` and independent of layer. So every degenerate layer gets the **same fallback** direction (modulo numeric differences).

Where: **0327–0341**.

This is not necessarily wrong, but it’s surprising. If you want per-layer distinct fallbacks, seed with `(SEED, layer, role)`.

Efficiency issues (very significant)

1) Steering explodes both compute and RAM

With defaults:

- `NUM_CONTROLS=500` , 4 direction types ⇒ **2000 controls**
- multipliers exclude 0 ⇒ `k_mult=6`
- per question you run ~ `2000 * 6 = 12,000` steered variants (plus the 4 main directions)

Even with caching, this is an enormous number of forward passes.

Worse: you preallocate massive nested structures:

- `layer_ctrl_confs[layer][dt][c][m] = [None] * n_questions` (and similar)
- That's **4 * 500 * 7 * n_questions** Python objects *per layer* (plus dict overhead), easily hundreds of millions of `None` s across layers.

Where: steering preallocation **0629–0638** and baseline linking loop **0687–0692**.

This will dominate runtime and likely OOM in Python memory long before the model compute finishes, for any non-trivial `n_questions` and number of layers.

Fix (big win, no change in outputs):
Don't store per-question confidences at all. You only use *means* over questions per multiplier per direction/control.

Instead maintain streaming accumulators per layer/task:

- For main directions: `sum_conf[dt, mult]` , `count[dt, mult]`
- For controls: `sum_conf_ctrl[dt, c_idx, mult]` , `count_ctrl[...]`

That reduces storage to ~ `4*7 + 4*500*7` floats per layer (tiny), and removes millions of Python list ops.

You can compute the final `mean_confs` arrays from sums/counts and then run `polyfit` .

Also: baseline linking into every control at multiplier 0 (lines **0689–0692**) becomes unnecessary; just treat multiplier 0 as baseline in the means.

2) Ablation also stores far more than needed

Ablation stores per-question confidences for every layer, direction, and control:

- `layer_ctrl_confs[layer][dt][c_idx]` is length `n_questions`
- That's `4*500*n_questions` entries per layer, again huge.

Where: **0931–0934**.

But you only use these lists to compute Pearson correlations with `metric_values`. Pearson correlation can be computed **online** from sufficient statistics without storing all samples:

Maintain for each (layer, dt) and for each control:

- `sum_x`, `sum_x2`, `sum_xy`, `n` where `x = ablated_conf`, `y = metric` (and similarly for baseline once)

Then compute correlation at the end.

This reduces memory from O(layers * controls * questions) to O(layers * controls).

---

3) Running causal experiments on *all non-degenerate layers* is likely infeasible

`causal_layers = [l for l in layers if not degenerate]` then you run steering + ablation on all of them (often 30+ layers).

Given the control counts, that's extreme.

Where: **1363–1372** onward.

Practical approach:

- pick a shortlist (e.g., top-k layers by predictive association, or a few representative layers), **or**
- drastically reduce `NUM_CONTROLS` for exploratory runs (e.g., 50), **or**
- compute controls only for the *orthogonalized* directions if that's your main hypothesis test.

Summary of what I'd change first (highest leverage)

1. **Replace `hash(dt)` seeds** with a fixed mapping (correctness + reproducibility).
2. Decide whether "predictive power" means **association on held-out** or **true prediction**:
    - If prediction: fit on `train_idx`, evaluate on `test_idx`.
    - If association: simplify to Pearson and report `corr` / `corr²` with bootstrap on the test set.
3. **Rewrite steering and ablation to be streaming-statistics based** (no giant `None` arrays).
4. Add **assertions for single-token options** (or implement multi-token option handling).

If you want, I can draft the exact refactor for (3) in a way that minimally touches your logic and preserves the JSON outputs/plots—just without the massive intermediate storage.