

Inspect Sparse Features

Gavin · AI-assisted course design

Contents

1	Goal	3
2	Part 1 — Predict before fitting	3
3	Part 2 — Implement the exact synthetic fixtures	4
3.1	Check the fixture before training	5
4	Part 3 — Fit six small autoencoders	5
4.1	Checkpoints and limits	7
5	Part 4 — Freeze and evaluate	7
5.1	Reconstruction and activity	7
5.2	Planted-direction matching	8
5.3	Explain the pattern	9
6	Part 5 — Inspect published Gemma Scope evidence	9
7	Runner contract and verification	9
8	Analytical checks	11
9	Completion checklist	11
10	More Learning	11
11	Optional technical runner	11

1 Goal

Train a small sparse autoencoder on vectors generated from a planted dictionary. Test reconstruction, activity, and recovery separately. Then inspect published Gemma Scope evidence and explain why an appealing feature label can be incomplete.

Prerequisite: Lesson 5.2, particularly the distinction between a latent and its interpretation.

Time: about 90–120 minutes for implementation and analysis. The computational experiment has a ten-minute wall-time ceiling; this is a stopping limit, not a promised runtime.

Requirements: a CPU, Python, and a compatible existing PyTorch installation. Python 3.13.13 with PyTorch 2.9.0 is the reference environment; record the environment actually used. The training core needs no other third-party package, account, GPU, API, network access, paid compute, or model download. Do not install packages automatically or start cloud resources. Reading the primary papers in Part 5 requires ordinary web access.

Expected artifact: your runner, its exact configuration and environment record, raw training history, checkpoints, held-out metrics, and a short interpretation. This Lab specifies an experiment; it supplies no measured training results. Analytical answers below are calculations, not evidence that a program ran.

⚠ Keep the claims within the experiment

The generator has planted numerical directions, not language concepts. Do not give its columns invented semantic names. Recovery may fail under these settings. A failed recovery with valid measurements completes the Lab; tuning against test-set answers until recovery looks good does not.

2 Part 1 — Predict before fitting

Use the lesson’s dictionary with columns $[1, 0]^T$, $[0, 1]^T$, and $[1, 1]^T / \sqrt{2}$.

1. For $x = [1, 1]^T$, verify both exact coefficient vectors from the lesson. Compute L0 and L1 for each.
2. For $x' = [1, 0.8]^T$, compare the exact two-coordinate reconstruction with the diagonal-only least-squares reconstruction. At $\lambda = 0.1$, calculate squared error plus λ times L1 for these two fixed candidate codes.
3. Explain why this comparison does not solve the full regularized optimization problem: the coefficient values themselves may change when regularization is included.
4. Starting with $d = [1, 0]^T$ and coefficient $z = 2$, replace them by $4d$ and $z/4$. What changes in reconstruction, L0, and L1?
5. Predict how increasing the sparsity penalty might change reconstruction error, mean activity, dead-latent counts, and planted-direction matching. Mark each prediction as an expectation rather than a guarantee.

Write one sentence explaining why two different random initializations are worth reporting. Do this before inspecting outputs.

3 Part 2 — Implement the exact synthetic fixtures

Use CPU float32 throughout the training experiment. Fix $d = 16$ observed dimensions, $K = 32$ planted directions, and $m = 64$ learned SAE latents. All indexes below are zero-based.

Create the planted matrix $D_\star \in \mathbb{R}^{16 \times 32}$ by drawing independent standard-normal entries with a dedicated PyTorch generator seeded 1801, then normalizing each column to unit Euclidean norm. Do not orthogonalize it, reject correlated columns, or search for a favorable seed.

For each row and planted index independently, sample

$$s_j \sim \text{Bernoulli}(1/16), \quad a_j \sim \text{Uniform}[0.5, 1.5), \quad z_j^\star = s_j a_j.$$

Generate $x = D_\star z^\star + \eta$, with independent coordinate noise $\eta_k \sim \mathcal{N}(0, 0.01^2)$. In row-major code, this is $X = Z^\star D_\star^T + \eta$.

Use separate generators for these independent splits:

- training: 4,096 rows, seed 1802;
- validation: 1,024 rows, seed 1803;
- test: 2,048 rows, seed 1804.

For each split, draw the entire support matrix first, then the amplitude matrix, then the noise matrix. This order is part of the fixture. Keep empty-support rows. Do not center, standardize, whiten, or normalize individual examples.

The exact generation functions are:

```
import torch

def planted_dictionary():
    g = torch.Generator(device="cpu").manual_seed(1801)
    d = torch.randn((16, 32), generator=g, dtype=torch.float32)
    return d / d.norm(dim=0, keepdim=True)

def make_split(dictionary, n, seed):
    g = torch.Generator(device="cpu").manual_seed(seed)
    support = torch.rand((n, 32), generator=g, dtype=torch.float32) < (1.0 / 16.0)
    amplitude = 0.5 + torch.rand((n, 32), generator=g, dtype=torch.float32)
    z_true = support.float() * amplitude
    noise = 0.01 * torch.randn((n, 16), generator=g, dtype=torch.float32)
    x = z_true @ dictionary.T + noise
    return x, z_true
```

Store separate tensor-only files; save model checkpoints as tensor state dictionaries. Use explicit CPU mapping and weights-only loading, such as `torch.load(path, map_location="cpu", weights_only=True)`, for these locally generated artifacts. Do not store or load arbitrary pickled Python objects. PyTorch loading reference

Store separate files:

- training and validation files contain only their X tensors;

- the test-input file contains only X_{test} ;
- an evaluation-truth file contains $D_\star$ and $Z_{\text{test}}^\star$.

Discard training and validation planted coefficients after constructing the inputs. Keep evaluation truth out of the trainer’s function arguments and file-loading path. The evaluator may first deserialize its tensors only after all declared runs are finalized. The generator necessarily creates these tensors, and integrity-only byte hashing of their saved file is allowed before that point; neither authorizes passing them to the trainer. This is an experimental separation, not a security system.

Record SHA-256 hashes of the produced files and preserve them. Matching seeds alone is not a guarantee of identical fixtures across PyTorch versions or platforms. Use the same generated fixture files for every run in this experiment. PyTorch reproducibility guidance

3.1 Check the fixture before training

Assert shapes, dtypes, finiteness, and planted unit norms within absolute tolerance 10^{-6} . The generator returns a test truth tensor solely so evaluation can compare it later; do not print matching scores yet.

The population expectation of planted L0 is $32/16 = 2$, not exactly two per row. The probability of no active planted coefficient is $(15/16)^{32}$. The expected squared noise norm is $16(0.01)^2 = 0.0016$. These are distributional facts, not acceptance tests requiring empirical equality.

Do not reject and regenerate data because measured frequencies depart from these expectations. Inspect substantial discrepancies as possible bugs while preserving the original files.

4 Part 3 — Fit six small autoencoders

Run the Cartesian product of initialization seeds $\{101, 202\}$ and penalties $\lambda \in \{0, 0.03, 0.1\}$. The $\lambda = 0$ runs are unregularized autoencoder controls. Their ReLU outputs may still contain zeros; “unregularized” does not mean every coefficient is positive.

Use this exact model, with row-major batches:

```
from torch import nn

class ToySAE(nn.Module):
    def __init__(self, seed, train_mean):
        super().__init__()
        g = torch.Generator(device="cpu").manual_seed(seed)
        d = torch.randn((16, 64), generator=g, dtype=torch.float32)
        d = d / d.norm(dim=0, keepdim=True)
        self.decoder = nn.Parameter(d.clone())
        self.encoder = nn.Parameter(d.T.clone())
        self.encoder_bias = nn.Parameter(torch.full((64,), -0.05,
            dtype=torch.float32))
        self.decoder_bias = nn.Parameter(train_mean.clone())

    def forward(self, x):
        z = torch.relu(
            (x - self.decoder_bias) @ self.encoder.T
```

```

        + self.encoder_bias
    )
    x_hat = z @ self.decoder.T + self.decoder_bias
    return x_hat, z

```

Initialize the decoder bias to the training-set mean, computed once from training inputs. The encoder starts as the decoder transpose but is an independent parameter thereafter. Do not initialize either matrix using the planted dictionary.

Before fitting, record an untrained checkpoint for each seed. This gives a random-direction/encoder baseline with the same shapes. The three penalties sharing a seed must begin with identical parameters and see identical batch order.

Use Adam with learning rate 0.003, betas (0.9, 0.999), epsilon 10^{-8} , and zero weight decay. Set foreach and fused optimizer paths to false when supported by the pinned version. Use 80 epochs, batch size 256, no learning-rate schedule, no sparsity warmup, no gradient clipping, no resampling, and no auxiliary loss. Each epoch contains exactly 16 updates.

At epoch $e \in \{0, \dots, 79\}$, form the row permutation with a fresh CPU generator seeded $100000 + s + e$, where s is the initialization seed. Use that permutation once, split into consecutive batches. Reset the model and optimizer independently for every run.

For batch size B , optimize

$$\frac{1}{B} \sum_{i=1}^B \|x_i - \hat{x}_i\|_2^2 + \lambda \frac{1}{B} \sum_{i=1}^B \sum_{j=1}^{64} z_{ij}.$$

Use sums over dimensions followed by means over rows. In particular, do not substitute a default elementwise-mean MSE without multiplying by 16.

The update order is:

```

optimizer.zero_grad(set_to_none=True)
x_hat, z = model(batch)
reconstruction = (batch - x_hat).square().sum(dim=1).mean()
mean_l1 = z.sum(dim=1).mean()
loss = reconstruction + penalty * mean_l1
if not torch.isfinite(loss):
    raise RuntimeError("non-finite objective")
loss.backward()

with torch.no_grad():
    d = model.decoder
    grad = d.grad
    # d has unit columns here; remove each parallel component.
    grad.sub_(d * (grad * d).sum(dim=0, keepdim=True))

optimizer.step()

```

```

with torch.no_grad():
    norms = model.decoder.norm(dim=0, keepdim=True)
    if not torch.isfinite(norms).all() or (norms <= 1e-12).any():
        raise RuntimeError("invalid decoder norm")
    model.decoder.div_(norms)

```

This is a teaching ReLU/L1 SAE. It is not the JumpReLU/L0 procedure used by Gemma Scope.

4.1 Checkpoints and limits

Set PyTorch intra-op and inter-op thread counts to one before computational work. Enable deterministic algorithms for this CPU exercise; stop and record any unsupported operation rather than silently changing it. Use no GPU/MPS, compilation, mixed precision, network downloads, or additional processes.

Save training and validation reconstruction, mean L1, and mean L0 at initialization and after epochs 20, 40, 60, and 80. Use evaluation mode and inference mode for these measurements, and return to training mode afterward. Validation is a diagnostic here: do not choose checkpoints, penalties, or stopping times from it. Every successful run’s designated checkpoint is epoch 80.

The maximum is six runs, 7,680 optimizer updates in total, and 600 seconds of wall time for training plus numerical evaluation. The wall-time ceiling overrides output-completeness requirements. Reaching the run or optimizer-update ceiling ends training; numerical evaluation may continue only within the same 600-second wall-time budget. Reaching the wall-time ceiling stops numerical work. Preserve completed checkpoints and existing logs. Track training/checkpoint status separately from evaluation status: completed training does not imply completed evaluation. Mark each unfinished evaluation as not started or interrupted, with its reason, and retain any valid partial results. Do not compute outside the limit to fill required tables. A runtime limit does not authorize extra compute.

Abort a run on nonfinite parameters, coefficients, gradients, or losses; record the stage and exception. Continue other independent runs only within the same overall budget. Never replace a failed run with a different seed and retain its original label.

5 Part 4 — Freeze and evaluate

First write a manifest listing the completed checkpoints, their hashes, final epochs, penalties, and seeds. If time remains within the overall budget, load held-out inputs and deserialize evaluation truth for the evaluator. All test metrics are post-training diagnostics. Do not feed them back into fitting or select a “best SAE” from their ground-truth matching scores.

Use activity threshold $\tau = 10^{-6}$ everywhere you report an empirical L0 or dead-latent count. Also report exact-positive mean L0 so numerical convention is visible.

5.1 Reconstruction and activity

For each trained run and each untrained-seed control, record:

- squared reconstruction error averaged over rows, with dimensions summed;
- FVU using the test-set mean in the denominator;
- mean L1, mean L0 at τ , and exact-positive mean L0;

- per-latent firing frequency and the number never above τ ;
- minimum and maximum decoder-column norms.

FVU is undefined if test variance is zero; return a JSON null with a reason, not NaN or an invented zero. Verify unit decoder norms within 10^{-5} after training.

Include two further baselines:

1. **Training-mean reconstruction:** reconstruct every test row as the training mean. Its FVU need not be exactly one because the metric denominator uses the test mean.
2. **Signed-coordinate identity:** set $D_{\text{id}} = [I, -I]$ and $z_{\text{id}} = [\text{ReLU}(x), \text{ReLU}(-x)]$. Use zero decoder bias. It reconstructs any input exactly up to arithmetic precision, with 32 available latents. Label it an analytical, hand-constructed control; do not claim it was learned.

The second baseline makes “low error proves feature discovery” untenable. Its width differs from the learned SAE, so it is not a controlled architecture comparison.

5.2 Planted-direction matching

Normalize a copy of each learned decoder column for evaluation. For planted index k and learned latent j , calculate signed cosine

$$C_{kj} = d_{*,k}^T \frac{d_j}{\|d_j\|_2}.$$

Choose $j^*(k) = \arg \max_j C_{kj}$, breaking ties with the smallest learned index. Do not take absolute cosine: coefficients are nonnegative, so flipping a direction’s sign is not automatically equivalent.

For each planted direction, record its best latent, cosine, and Pearson correlation between $Z_{\text{test},k}^*$ and the learned coefficient column $Z_{\text{test},j^*(k)}$. Compute correlation with centered columns; return null if either centered column has zero norm. This is an association metric, not a causal intervention.

Report:

- mean and minimum best cosine;
- fraction of planted directions with best cosine at least 0.90;
- fraction meeting both cosine at least 0.90 and coefficient correlation at least 0.80;
- the full 32-row matching record;
- collision count $32 - |\{j^*(k) : k = 0, \dots, 31\}|$.

Both candidate-match fractions always use all 32 planted directions as their denominator. An undefined correlation does not qualify for the joint threshold; retain its row with a null value and reason rather than dropping it.

Call the thresholded fractions **candidate matches**, not uniquely recovered concepts. The thresholds are declared diagnostic conventions, not statistical significance tests. Independent nearest-neighbor matching can assign several planted directions to one learned latent; the collision count exposes that limitation. We do not claim an optimal one-to-one assignment.

As a negative association control, permute all rows of Z_{test}^* with a CPU generator seeded 1805, leave learned activations unchanged, and recompute correlations for exactly the same matched pairs. Do

not repeat shuffles until the result looks convincing. Finite-sample shuffled correlations need not be exactly zero.

5.3 Explain the pattern

Compare the six declared runs without dropping failures. Did regularization change the fidelity–activity tradeoff? Did the best reconstruction also have the strongest matching? Are any directions aligned while their encoder coefficients track poorly? Did two seeds produce different dictionaries? Identify at least one specific mismatch, collision, inactive latent, or other limitation. If none of a particular kind occurred, say so rather than inventing it.

No recovery threshold is required for Lab completion. If all runs recover poorly, report that outcome and propose one follow-up using a fresh held-out dataset. Do not perform an unbounded search as part of this experiment.

6 Part 5 — Inspect published Gemma Scope evidence

Keep this section separate from your own numerical results. Use the original Gemma Scope report and the publisher’s residual-stream model card to record:

1. the base model family and what the residual-stream SAE observes;
2. the meaning of layer indexing, width, and average L0;
3. how JumpReLU selects active coefficients;
4. why its L0 training needs a gradient estimator;
5. one reason an arbitrary SAE should not be attached at a similarly sized but different model boundary.

Then read Section 4.2 and Figures 4–5 of A is for Absorption. This is a published inspection of Gemma Scope latents with fixed, citable evidence, so it does not depend on an interactive dashboard being available.

Record the authors’ specimen: Gemma 2 2B base model, post-MLP residual stream, zero-indexed layer 3, width 16k, average L0 59, latent 6510 and token-aligned latent 1085. Explain the reported contrast between the usual first-letter behavior and the exceptional token represented in the paper as “_short.” The underscore is the paper’s token-display convention; it is not an instruction to type a literal underscore into an unverified tokenizer.

Distinguish the authors’ activation observations from their ablation evidence. Explain why top examples of the first-letter latent alone would not reveal the missed case. Do not present their measurements as your own replication.

Optional: open the Gemma Scope demonstration linked by the release. If accessible without an account or download, inspect one feature page and record its exact URL, identifiers, date, displayed description, and limitations. If unavailable, omit it; the paper-based inspection completes this part. No account creation, model execution, gated access, or multi-gigabyte download belongs in this Lab.

7 Runner contract and verification

Implement a single local entry point, for example:

```
python lab18.py --output outputs/lab18
```

The only required command-line option is the output directory. All core settings above remain fixed. Refuse an existing nonempty output directory rather than silently overwriting a prior experiment. A deliberate new experiment uses a new directory.

The runner must produce the records below for work completed within the budget. Missing or partial numerical evaluations must have explicit status and reasons; output requirements never extend the hard time limit:

- a configuration manifest with fixture seeds, dimensions, distributions, run grid, optimizer settings, limits, activity/matching thresholds, and split counts;
- an environment record with Python, PyTorch, operating system, CPU, dtype, thread settings, deterministic setting, start/end timestamps, and measured elapsed time;
- generated fixture files and SHA-256 hashes;
- per-run initialization and final checkpoints plus their hashes;
- training/validation history at the declared checkpoints;
- test metrics and complete matching records for each completed evaluation of a trained run or untrained control, with partial records labeled;
- baseline and shuffled-association results;
- separate training/checkpoint and evaluation statuses for every scheduled run: completed, failed, interrupted, or not started; analogous evaluation statuses for controls and baselines;
- a validation report containing actual pass/fail results.

JSON and CSV from the standard library are sufficient. No plotting library is required. Serialize undefined values as null with a reason; reject nonstandard JSON NaN/Infinity. Use relative paths in manifests so artifacts can move together.

The validation report must check:

1. generator and model shapes, finiteness, and decoder normalization;
2. hand-calculated objective examples from Part 1;
3. loss reductions using a deliberately small batch;
4. equality of initialization and batch order across penalties sharing a seed;
5. that the trainer receives no planted dictionary or coefficient tensors;
6. that the evaluator first deserializes evaluation-truth tensors after the completed-run manifest is written; generator creation and earlier integrity-only byte hashing are allowed;
7. identity-baseline reconstruction within absolute tolerance 10^{-6} ;
8. that permuting both decoder columns and matching encoder rows/biases preserves reconstruction within 10^{-5} ;
9. output completeness and recorded interruption/failure states.

A source-code check plus a timestamped stage log supports checks 5–6. If the time budget prevents a runtime validation, mark that check not performed rather than claiming a pass. These are transparent leakage safeguards, not a claim of tamper-proof isolation. Preserve the original implementation and outputs if a bug is found; describe any corrected rerun separately.

8 Analytical checks

These are expected mathematical answers, not expected training outcomes:

- At $x = [1, 1]^T$, the two candidate codes have L0 values 2 and 1, L1 values 2 and $\sqrt{2}$, and zero reconstruction error.
- At $x' = [1, 0.8]^T$, the exact coordinate code has objective 0.18 for $\lambda = 0.1$. The fixed diagonal least-squares code has objective $0.02 + 0.1(1.8/\sqrt{2}) \approx 0.14728$.
- Those fixed codes need not minimize the regularized objective. For the diagonal-only constrained problem, the coefficient becomes $\max(0, 1.8/\sqrt{2} - 0.05)$ at $\lambda = 0.1$.
- Scaling d by 4 and its coefficient by $1/4$ leaves reconstruction and L0 unchanged while dividing L1 by 4.
- An identity reconstruction has zero theoretical error; its success says nothing about recovery of the planted dictionary.
- A zero-variance coefficient column has undefined Pearson correlation.

9 Completion checklist

- Predictions were recorded before viewing training results.
- The actual SAE trained on inputs alone, within the CPU/time/run limits.
- Held-out ground truth was used only for post-training evaluation.
- Every declared run, control, failure, and interruption is accounted for, with training and evaluation statuses separated.
- Reconstruction, activity, direction alignment, and coefficient association are separate measurements.
- Published Gemma Scope evidence is labeled as source inspection.
- The conclusion names the dataset, settings, observations, and limits without asserting that one latent always equals one concept.

10 More Learning

- Gemma Scope, Sections 2–4: compare its actual architecture and evaluation with this deliberately small teaching model.
- A is for Absorption: study missing positive cases and the additional evidence from interventions.
- SAE Bench: examine why several evaluation questions are needed.
- PyTorch reproducibility guidance: distinguish repeatability in one environment from cross-platform identity.

11 Optional technical runner

Run with the existing PyTorch environment: `python inspect_sparse_features.py --output NEW_DIRECTORY`. The six runs and shared 600-second limit are fixed. This runner performs synthetic training and numerical evaluation; published-paper inspection remains a separate reading task. Download the following files into one directory: `inspect_sparse_features.py`, `sparse_feature_evaluation.py`, `inspection_common.py`, `inspect_residual_stream.py`.