H TOKEN DISTRIBUTION ENTROPY VS. AUTO-INTERPRETABILITY

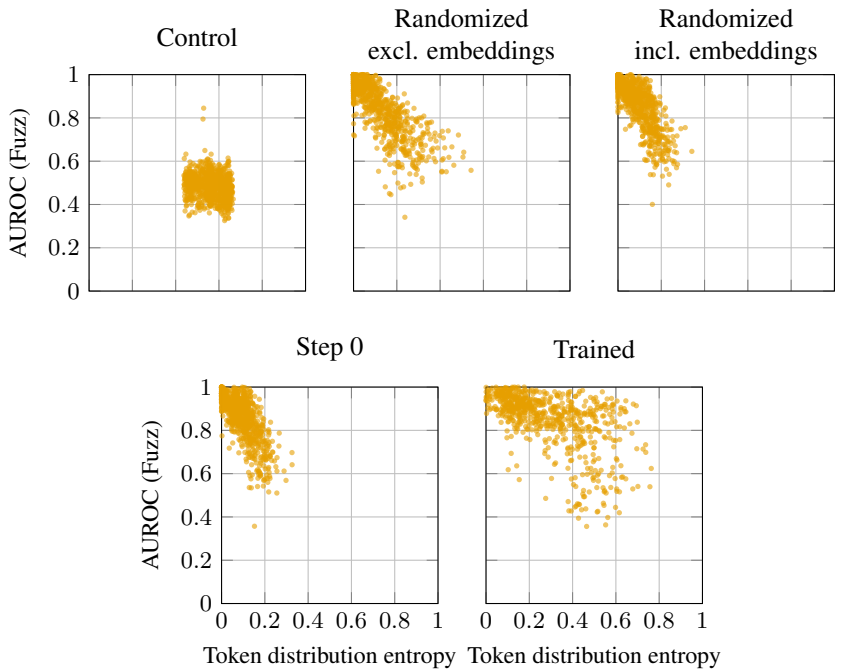

Figure 20: Scatter plots of the per-latent token distribution entropy against 'fuzzing' AUROC (auto-interpretability score) for SAEs trained on multiple layers of the Pythia-6.9b model. Each point corresponds to a single latent, taken from the sample of latents used to compute the aggregate metrics displayed in Figures 1 and 2.

Figure 20 clearly distinguishes the negative control, randomized variants, and the trained variant:

- **Control:** Latents have a consistently high entropy (i.e., max activating examples with activation patterns spread across many tokens) and low auto-interpretability score (i.e., generated explanations that fail to adequately explain these activation patterns). No correlation between the two variables is evident.
- **Randomized:** For each of the randomization schemes described in Section 3, we see a negative correlation between entropy and auto-interpretability: in general, the wider variety of tokens for which a latent is activated, the less well the latent's activation patterns are explained by its generated explanation.
- **Trained:** There is a weaker correlation between the two variables. Crucially, in addition to the broad trend observed for the randomized variants, we also see latents with high entropy *and* auto-interpretability. Some latents have activation patterns that are spread across multiple tokens, which are nevertheless consistent with the latent's generated explanation.

These results are consistent with the view that aggregate auto-interpretability scores obscure the differences between SAEs based on trained and randomized models. While randomized models with consistent token embeddings can produce 'single-token' features, whose activation patterns are easy to explain, only Transformers trained on natural language produce more complex semantic features.

I A TOY MODEL OF SUPERPOSITION

In Section 4, we trained SAEs on toy data designed to exhibit superposition (Sharkey et al., 2022) and GloVe word vectors (Pennington et al., 2014). In this section, we detail the data-generation procedure and training setup.

I.1 DATA GENERATION

First, we construct ground-truth features by sampling n_s points on an n_d-dimensional hypersphere.

For each sample, we determine the feature coefficients by generating $A \in \mathbb{R}^{n_s \times n_s}$ where $A_{ij} \sim \mathcal{N}(0, 1)$, defining a covariance matrix $\Sigma = AA^\mathsf{T}$, sampling $\vec{\alpha} \in \mathbb{R}^{n_s}$ where $\alpha_i \sim \mathcal{N}(\vec{0}, \Sigma)$, projecting α_i onto the c.d.f. of $\mathcal{N}(0, 1)$, decaying $\alpha_i \to \alpha_i^{\lambda i}$ where $\lambda \in \mathbb{R}$, normalizing $\alpha_i \to m\alpha_i / n_s \sum_j \alpha_j$ where $m \in \mathbb{R}$, and performing n_s independent Bernoulli trials with $p = \alpha_i$. Finally, we multiply the trial outcomes by n_s independent samples from a continuous uniform distribution $\mathcal{U}_{[0,1)}$.

The parameter λ determines how sharply the frequency of nonzero ground-truth feature coefficients decays with the feature index i. The parameter m is the expected value of the number of nonzero feature coefficients for each sample.

Like Sharkey et al. (2022), we choose $n_s = 512$, $n_d = 256$, $\lambda = 0.99$, and $m = 5$. We include a Python implementation of this procedure in Figure 21.

I.2 TRAINING

The SAEs described in Section 4 comprise a linear encoder with a bias term, a ReLU activation function, and a linear decoder without a bias term. We use orthogonal initialization for the decoder weights and normalize the decoder weight vectors before each training step.

The training loss is the mean squared error (MSE) between the input and decoded vectors, plus the mean L^1 norm of the encoded vectors multiplied by a coefficient, which we vary between 1×10^{-3} and 100.

For the toy data, we train for 100 epochs on 10K data points with 10 random seeds. For the word vectors, we train for 100 epochs on 400K data points with 1 random seed. In both experiments, we reserve 10% of the data points as a validation set, which we use to compute evaluation metrics.

The MLPs described in Section 4 comprise two layers (i.e., one hidden layer) and a ReLU activation function. The input and output sizes are both equal to n_d, and the hidden size is $4n_d$. We loosely based these choices on the feed-forward network components of transformer language models.

```python
def generate_sharkey(
    num_samples: int,
    num_inputs: int,
    num_features: int,
    avg_active_features: float,
    lambda_decay: float,
) -> tuple[Tensor, Tensor]:
    """
    Args:
        num_samples (int): The number of samples to generate.
        num_inputs (int): The number of input dimensions.
        num_features (int): The number of ground truth features.
        avg_active_features (float): The average number of
            ground truth features active at a time.
        lambda_decay (float): The exponential decay factor for
            feature probabilities.
    """
    features = torch.randn(num_inputs, num_features)
    features /= torch.norm(features, dim=0, keepdim=True)

    covariance = torch.randn(num_features, num_features)
    covariance = covariance @ covariance.T
    correlated_normal = MultivariateNormal(
        torch.zeros(num_features), covariance_matrix=covariance
    )

    samples = []
    for _ in range(num_samples):
        p = STANDARD_NORMAL.cdf(correlated_normal.sample())
        p = p ** (lambda_decay * torch.arange(num_features))
        p = p * (avg_active_features / (num_features * p.mean()))
        p = torch.bernoulli(p.clamp(0, 1))
        coef = p * torch.rand(num_features)

        sample = coef @ features.T
        samples.append(sample)

    return torch.stack(samples), features
```

Figure 21: A Python implementation of the data-generation procedure introduced by Sharkey et al. (2022) and used in Section 4.