Key Results on “SEAs for gLMs”

**Model:** [InstaDeepAI/nucleotide-transformer-v2-50m-multi-species · Hugging Face](https://huggingface.co/InstaDeepAI/nucleotide-transformer-v2-50m-multi-species)   
d(hidden) = 512

**SAE:** d\_SAE = 4096

I trained an SAE on the MLP activations in (1) the penultimate (L10) and (2) last (L11) transformer block of the Nucleotide Transformer -50m model.

# Interpretable SAE latents

*For Layer 11,* inspecting the most activating tokens for many SAE latents, I found

* ~15 **‘syntactic features**’ - which appear to share some kmer or all have some nucleotide in a certain position:
  + E.g. one latent that most strongly activates on tokens with “GTGT”
  + Found one latent that somewhat selectively corresponds to “is palindrome” (high-level syntactic concept)
* some ~6 **“functional features”** in the sense that the most activating tokens shared some plasmid annotation
  + E.g. “CMV promoter/enhancer” or “Cas9”
  + Many annotations seem to be highly linearly separable in the activation space but don’t have a dedicated SAE unit

*Comparing MLP layers:* most features (I tested so far) that were present in the SAE for L11 also seem to be present in the SAE L10. An exception appears to be the “f1 ori” feature, even though the linear separability (i.e. classification performance of linear probe) is similar in both layers.

## Measuring Monosemanticity

For the “CMV enhancer/promoter” feature I tried measuring the the precision and recall of the SAE latent:

P(token contains CMV): 0.033

P(token contains CMV|activation > 0.0): 0.543

P(activation > 0.0): 0.046

P(activation > 0.0|token contains cmv): 0.763

Strength of evidence for act > 0.0 from cmv (as BayesF): 66.768

Strength of evidence for cmv from act > 0.0 (as BayesF): 35.085

Finding that the *latent firing* provides strong, though not definitive, evidence for *input being annotated with ‘CMV promoter/enhancer’*. This holds in both directions.

# Steering with SAE latents

I tried steering the model towards some feature F by

1. adding a vector to the residual stream during the forward pass, where
2. The vector is a reconstruction using the SAE where we manually increase the hidden activation by 50-1000 of the target feature F before reconstruction

I then observed which tokens the model assigns most probability mass to (a) with steering and (b) without steering. For 3-5 features, I observed that the model started shifting almost all its probability mass towards tokens with that feature, assuming that we sufficiently increased the activation of that feature before reconstruction.

## Case Study: ‘contains TAG’

SAE latent 1755 activates most strongly on tokens containing the kmer ‘TAG’. When using the above procedure to increase the presence of this feature in the residual stream, the probability of next tokens with “TAG” increases from 0% (no intervention) to 64%. The same procedure with the latent for ‘TCAT’ raises the probability of the next token containing ‘TCAT’ from 0 to 60-80%.

# SAE Architecture

## Jump ReLU

Instead of using the standard ReLU, there’s a threshold above 0 (in my case 0.3) s.t. All activations below that threshold are set to 0. This leads to greater sparsity (i.e. more latents being zero)

## Continuous L0 approximation

(This is applied pre-thresholding)

Key mechanism:

* Uses sigmoid((|x| - threshold)/temperature) to create a differentiable function
  + Threshold = 0.3
* Outputs < 1/2 when |x\_i| < threshold
* Outputs converge to 1 when (|x\_i| > threshold) -> inf
* Temperature controls how sharp the transition is
  + Currently just set to 1, no effect

Motivation:

* Makes the non-differentiable L0 norm optimization-friendly
* Maintains bounded [0,1] range like true L0

Limitations:

* Overestimates zeros (output > 0 when should be 0)
* Underestimates non-zeros (output < 1 when should be 1)
* Treats distance from zero as key factor rather than just non-zeroness

Why does this work?

The original L0 norm is I(x > 0) e.g. the indicator function applied to x being positive. If x <= 0 then the value is 1 and 0 otherwise.

Now a sigmoid function maps numbers from the real line onto (0,1) so a sigmoid can map to a step function 0 on the non-positive values 1 otherwise. If the sigmoid slope part is straight up at 0. However, we can relax this sharp transition by considering a differentiable slop up from 0 to 1. The parameters threshold and temperature control the exact location of the slope and how steep it is. As temperature gets smaller (close to 0) the fraction approaches infinity and so sigmoid(\infty) -> 1 getting you the approximation.

There are many ways to do this but the nice differentiability of the exponential that helps. The original paper is here: <https://hal.science/hal-00173357/document>.

Another way to think about it is create a little bump using the gaussian distribution around 0.

## Forward Pass (code)

**def get\_continuous\_l0(self, x):**

**"""**

**Compute continuous relaxation of L0 norm using sigmoid**

**This provides useful gradients unlike the discrete L0**

**"""**

**# Shifted sigmoid to approximate step function**

**return torch.sigmoid((x.abs() - self.threshold) / self.temperature)**

**def forward(self, x):**

**# encoding and decoding of input vec**

**x\_cent = x - self.b\_dec**

**pre\_acts = x\_cent @ self.W\_enc + self.b\_enc**

**acts = F.relu(pre\_acts)**

**# Compute continuous L0 approximation before thresholding**

**l0\_proxy = self.get\_continuous\_l0(acts)**

**# Apply hard threshold for forward pass --- This is actually jumprelu (I think!)**

**acts\_sparse = (acts.abs() > self.threshold).float() \* acts**

**x\_reconstruct = acts\_sparse @ self.W\_dec + self.b\_dec**

**# L2 Loss (Reconstruction Loss)**

**l2\_loss = F.mse\_loss(x\_reconstruct.float(), x.float(), reduction='none')**

**l2\_loss = l2\_loss.sum(-1)**

**l2\_loss = l2\_loss.mean()**

**# Normalized MSE for reporting**

**nmse = torch.norm(x - x\_reconstruct, p=2) / torch.norm(x, p=2)**

**# Continuous L0 loss (using sigmoid approximation)**

**l0\_loss = l0\_proxy.sum(dim=1).mean()**

**# Total Loss: reconstruction + sparsity**

**loss = l2\_loss + self.l0\_coeff \* l0\_loss**

**# For monitoring: true L0 count (not used in optimization)**

**true\_l0 = (acts\_sparse.float().abs() > 0).float().sum(dim=1).mean()**

**# For monitoring: L1 loss**

**l1\_loss = acts\_sparse.float().abs().sum(-1).mean()**

**return loss, x\_reconstruct, acts\_sparse, l2\_loss, nmse, l1\_loss, true\_l0**