Mechanistic Interpretability for Neural TSP Solvers

Reuben Narad[1] **Léonard Boussioux**[1,2] **Michael Wagner**[1]
[1]University of Washington, Michael G. Foster School of Business
[2]University of Washington, Paul G. Allen School of Computer Science & Engineering
{rnarad, leobix, mrwagner}@uw.edu

Abstract

Neural networks have advanced combinatorial optimization, with Transformer-based solvers often outperforming classical algorithms on the Traveling Salesman Problem (TSP). However, these models remain black boxes, limiting our understanding of what optimization strategies they discover. We apply sparse autoencoders (SAEs) to interpret neural TSP solvers. To our knowledge, this is the first work to bring mechanistic interpretability from deep learning models to operations research. Our analysis reveals interpretable features that these solvers naturally developed that mirror fundamental TSP-solving concepts: boundary detection, spatial clustering, and geometric separations. These discoveries reveal how neural solvers approach combinatorial problems and suggest new directions for hybrid approaches that combine algorithmic transparency with neural performance. Interactive feature explorer: https://reubennarad.github.io/TSP_interp/

1 TSP Solver

The TSP – finding the shortest route visiting all cities exactly once – is fundamental to combinatorial optimization with applications in logistics, manufacturing, and circuit design. Classical approaches rely on exact algorithms or hand-crafted heuristics built on decades of mathematical insight. Transformers have recently revolutionized TSP solving, learning effective strategies through reinforcement learning that often outperform traditional methods. This paradigm shift from explicit algorithmic design to learned optimization strategies comes with a critical trade-off: while classical methods offer transparent reasoning, neural solvers operate as black boxes. We address this interpretability gap by applying sparse autoencoders (SAEs) to a Transformer-based TSP solver. Concretely, we (i) train a Transformer TSP solver, (ii) record many residual-stream neuron activations, (iii) fit a top-k sparse autoencoder on these activations to learn an overcomplete, sparse feature dictionary, and (iv) analyze and visualize the resulting features. We discover that the solver naturally develops features for boundary detection, spatial clustering, and geometric separations—fundamental patterns that mirror human TSP-solving intuition. SAEs have been used to find colors/texture features in vision models [10], features related to specific concepts [11] in LLMs, protein motifs in DNA foundation models [3], and anomaly detection in robots [7], among others. To our knowledge, our work is the first to apply the same tool to neural OR solvers.

1.1 Architecture

We use RL4CO's [2] implementation of Kool et al.'s Transformer Pointer Network [8], which performs next-node prediction to autoregressively construct TSP solutions. We focus on TSP instances with a fixed size of 100 nodes. The Transformer encoder processes all nodes through Transformer blocks of attention mechanisms and feed-forward networks, with residual connections. The decoder autoregressively constructs solutions using pointer networks: at each step, it computes

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: MLxOR: Mathematical Foundations and Operational Integration of Machine Learning for Uncertainty-Aware Decision-Making.

attention between the current partial tour and remaining nodes, then selects the next node via masked softmax over selection probabilities. We attach our SAE to the node embeddings at the encoder's final residual stream—after the last Transformer block processes the full spatial context.

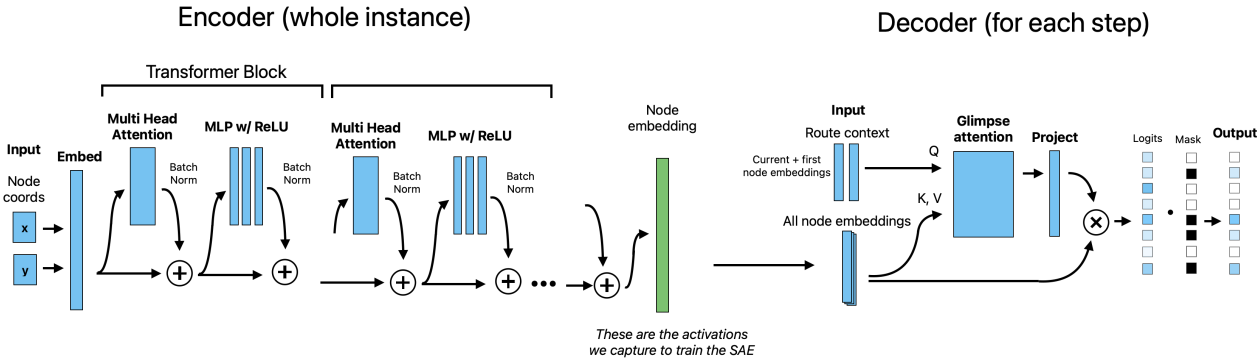

Figure 1: System architecture of the TSP solver Transformer model, adopting the architecture from [8].

1.2 TSP instance training data

Our TSP policy is trained on instances with node coordinates represented as 2D vectors (x, y). We focus on instances where nodes are drawn uniformly over the unit square $[0, 1]^2$, but we also experiment with structured distributions like clustered Gaussians, concentric shapes, and real road-network waypoints.

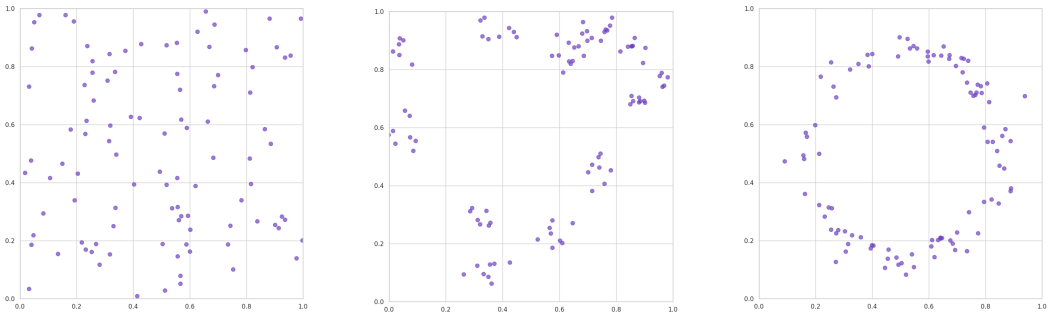

Figure 2: Example TSP instances drawn from different distributions.

During training, we resample the instances every epoch. For most settings, we used 100,000 instances per epoch for 300 epochs, or 30 million unique instances. Alongside the training instances, we freeze a separate validation environment that resamples the same instances at each validation step. We also solve those validation tours optimally with the Concorde solver [1] – a highly optimized exact TSP solver– so that we can record our model's performance in terms of suboptimality throughout training.

1.3 TSP solver Training Method

Due to the nature of the TSP, generating optimal solutions at scale for supervised learning is computationally intractable. Instead, we train our solver using reinforcement learning, following the work of [6], allowing the model to learn directly from interaction with the environment rather than relying on supervised labels. Training is conducted using the REINFORCE algorithm [12] with gradient clipping to ensure stable parameter updates. The reward signal is the negative tour length, incentivizing the model to find shorter tours. Each training run begins with a warm-up phase consisting of 1,000 greedy rollouts, which helps initialize the baseline and stabilize early policy learning. Afterwards, node selection during rollouts is sampled from a temperature-controlled softmax distribution. Once REINFORCE produces gradients, we update parameters with the Adam optimizer.

2 Sparse Autoencoder

2.1 Interpretability Goal

Understanding what the neural TSP solver has learned requires dissecting its latent space in a human-interpretable way. We adopt the language of [4], where “features" are important directions in the model’s latent space that behave like variables: each has a value (its activation) on every forward pass. "Circuits," on the other hand, are relationships between these features that behave like functions. We focus on finding "features." If we can reliably name such variables ("this node is on the convex hull" or "that node is this node’s nearest neighbor," for example), we build a mechanistic narrative of *why* the policy selects one next node over another[4].

2.2 SAE Background

A common obstacle in interpreting neural networks is *polysemanticity,* a phenomenon where individual neurons encode multiple concepts [9]. In language models, for example, this would look like a neuron that fires for both French negations and HTML tags, confounding interpretation. A popular tool for "disentangling" the neurons is the Sparse Autoencoder (SAE), a secondary ML model trained on the activations of the model being interpreted. SAEs learn an *over-complete* and *sparse* representation of the activations.

The SAE architecture has 3 components: an encoder, an activation function, and a decoder. Its training objective, as an autoencoder, is to reconstruct the input after passing it through its encoder’s latent space. As a compression task, autoencoders typically have a smaller-dimensional latent space than the input. The key difference with *Sparse Autoencoders* is that the latent space is higher-dimensional, with the additional sparsity constraint:

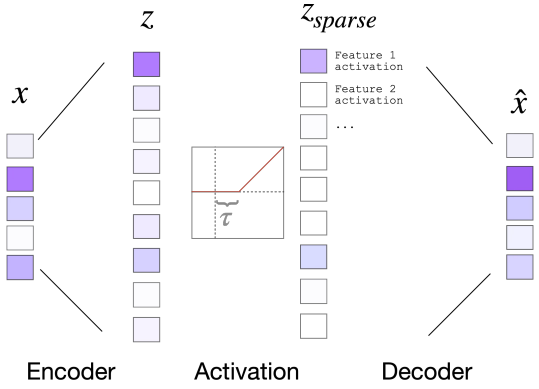

Figure 3: The SAE learns an overcomplete and sparse representation of neuron activations of the TSP solver model

Mathematically, the SAE forward pass consists of three steps. Given a node embedding $x \in \mathbb{R}^d$, the *encoder* first projects to the latent space:

$$z = xW_{\text{enc}}^{\top} + b,$$

where $W_{\text{enc}} \in \mathbb{R}^{n \times d}$ contains the learned feature directions and $b \in \mathbb{R}^n$ is a bias vector. Next, the *sparsification* step applies top-k activation:

$$z_{\text{sparse}} = \text{TopK}(z) = \text{ReLU}(z - \tau),$$

where τ is the k-th largest value in z, effectively zeroing all but the k largest activations, creating the sparse bottleneck. Finally, the *decoder* reconstructs the input:

$$\hat{x} = z_{\text{sparse}}^{\top} W_{\text{dec}} + b_{\text{dec}},$$

where $W_{\text{dec}} \in \mathbb{R}^{n \times d}$ maps the sparse latent representation back to the original space.

Because of the sparsity constraint, the discovered features are less likely to be polysemantic, and more interpretable to humans.

2.3 SAE Training Details

Recall that the TSP model works by first embedding the nodes, then autoregressively decoding the solution. To capture the richest version of the graph's embedding, we attach the SAE to the encoder output, at the residual stream. Our SAE follows OpenAI's top-k SAE recipe [5]. At each forward pass, we zero all but a token's top-activating 10% of SAE neurons, giving us an effective ℓ_0 constraint while still enjoying gradient flow through the soft top-k mask. The key hyperparameters are the *expansion factor* (how much larger the SAE embedding is than the TSP model embedding), the *k-ratio* (sparsity of the SAE) and the ℓ_1 *coefficient* (weight between reconstruction and sparsity objectives). After some tuning, we found $\texttt{expansion factor} = 4, \texttt{k\_ratio} = 0.1, \texttt{l1\_coefficient} = .001$ to be a good configuration.

We collect the SAE's training data by running inference on the TSP model for 100,000 instances, collecting the graph embedding for each one. Crucially, these instances were drawn from the same distribution as the TSP solver's training data. After hyperparameter tuning, the resulting dictionary reconstructs residual activations near-perfectly, with increasing sparsity as training progresses.

3 Feature Analysis

3.1 Feature Activation Mechanics

After finding a set of features with the SAE, we want to understand how each feature corresponds to different TSP attributes. A "feature activation" is simply the activation value of a specific neuron (feature) in the SAE's latent space during a forward pass. For a given node embedding x, we compute the SAE's sparse latent representation z_{sparse} using the encoder and top-k sparsification described above. The activation of feature i is then just $z_{\text{sparse}}[i]$—the i-th component of this sparse vector.

Since each TSP instance contains multiple nodes, feature i produces a collection of activations $\{z_{\text{sparse}}[j, i]\}_{j=1}^{N}$ across all N nodes. To summarize how strongly a feature responds to an entire instance, we compute the mean activation, that helps us sort features by relevance for different types of TSP instances:

$$\mu_i \;=\; \tfrac{1}{N} \sum_{j=1}^{N} z_{\text{sparse}}[j, i].$$

3.2 Visualizing a Feature

To see what a feature is doing, we overlay ten 100-node instances from the same distribution in a single x-y plot. Each point's color encodes the corresponding value of $z_{\text{sparse}}[j, i]$ (dark = 0, bright = high activation), while marker shape distinguishes which of the ten instances the node belongs to. Because activations are node-wise, this composite heat-map lets us spot geometric or combinatorial regularities at a glance—e.g., gradients that track tour direction, clusters of high-activation nodes near dense regions, or symmetry patterns that correlate with particular edge layouts.

3.3 Discussion

For demonstration, we trained an SAE on a model trained on uniform distributions. As shown in Figure 4 in the Appendix, we found many recurring themes among the features. These feature categories suggest that the SAE successfully disentangles different aspects of spatial reasoning that are important for TSP solving, from boundary detection to spatial clustering and geometric separations. Our next goal is to recover circuits: use cross-layer transcoders to map features in the encoder residual stream to node-selection behavior in the decoder, then confirm causality with patching and head/MLP ablations. We therefore treat feature–behavior links as correlational, not causal, and do not make claims about their influence on decisions. Ultimately, we aim to distill these circuits into transparent, reusable OR primitives, and understand what these models are really doing.

Acknowledgments and Disclosure of Funding

We thank the University of Washington's HYAK compute cluster for providing computational resources, and the RL4CO and Concorde developers for open-source tools.

References

[1] David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, Princeton, NJ, 2006. ISBN 9780691129938.

[2] Federico Berto, Chuanbo Hua, Junyoung Park, Laurin Luttmann, Yining Ma, Fanchen Bu, Jiarui Wang, Haoran Ye, Minsu Kim, Sanghyeok Choi, et al. RL4CO: an Extensive Reinforcement Learning for Combinatorial Optimization Benchmark. *arXiv preprint arXiv:2306.17100*, 2024. `https://github.com/ai4co/rl4co`.

[3] Myra Deng, Daniel Balsam, Liv Gorton, Nicholas Wang, Nam Nguyen, Eric Ho, and Thomas McGrath. Interpreting evo 2: Arc institute's next-generation genomic foundation model, February 2025. URL `https://www.goodfire.ai/blog/interpreting-evo-2`. Goodfire Research blog; published Feb. 20, 2025.

[4] Nelson Elhage, Nicholas Joseph, Tom Henighan, et al. A mathematical framework for transformer circuits. *Distill (Transformer Circuits Thread)*, 2021.

[5] Leo Gao, Chris Olah, Nick Cammarata, et al. Scaling and evaluating sparse autoencoders. *arXiv preprint arXiv:2306.04093*, 2024. OpenAI Technical Report on Sparse Autoencoders.

[6] Chaitanya K. Joshi, Thomas Laurent, and Xavier Bresson. On learning paradigms for the travelling salesman problem. In *NeurIPS Graph Representation Learning Workshop*, 2019.

[7] Taewook Kang, Bum-Jae You, Juyoun Park, and Yisoo Lee. A real-time anomaly detection method for robots based on a flexible and sparse latent space, April 2025. URL `https://arxiv.org/abs/2504.11170`. arXiv:2504.11170v2 [cs.RO].

[8] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations (ICLR)*, 2019.

[9] Catherine Olsson et al. In-context learning and induction heads. Technical report, 2022. Technical report on transformer circuits.

[10] Samuel Stevens, Wei-Lun Chao, Tanya Berger-Wolf, and Yu Su. Sparse autoencoders for scientifically rigorous interpretation of vision models, February 2025. URL `https://arxiv.org/abs/2502.06755`.

[11] Adly Templeton, Tom Conerly, Jonathan Marcus, Jack Lindsey, Trenton Bricken, Brian Chen, Adam Pearce, Craig Citro, Emmanuel Ameisen, Andy Jones, Hoagy Cunningham, Nicholas L Turner, Callum McDougall, Monte MacDiarmid, Alex Tamkin, Esin Durmus, Tristan Hume, Francesco Mosconi, C. Daniel Freeman, Theodore R. Sumers, Edward Rees, Joshua Batson, Adam Jermyn, Shan Carter, Chris Olah, and Tom Henighan. Scaling monosemanticity. *Transformer Circuits Forum*, 2024. URL `https://transformer-circuits.pub/2024/scaling-monosemanticity/`. * equal contribution; Anthropic; Published May 21, 2024.

[12] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3–4):229–256, 1992.

A Example visualizations of discovered SAE features

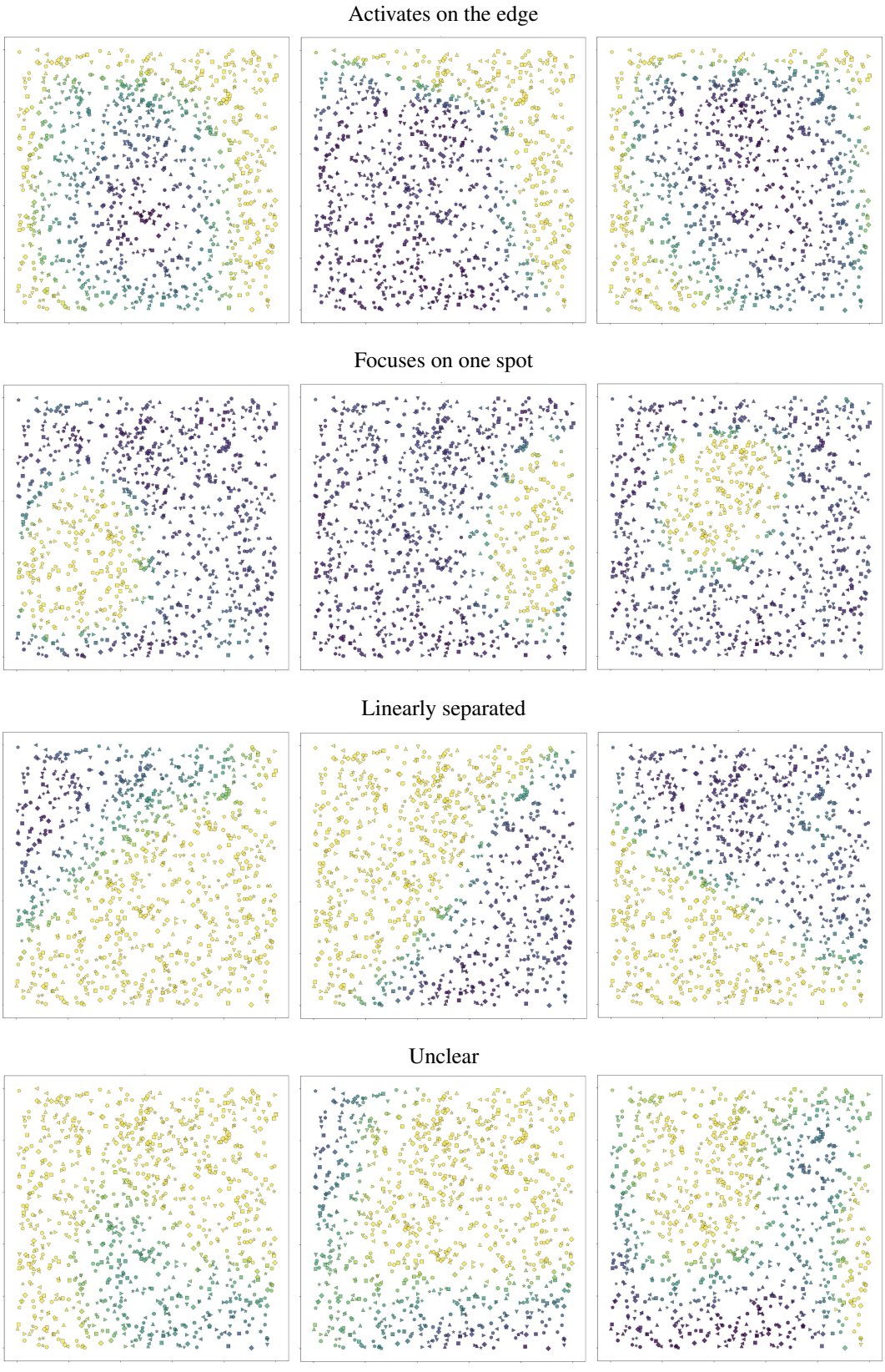

Figure 4: Each panel is a visualization of a single SAE feature. It overlays nodes from 10 TSP instances, with the marker shape indicating the instance this node was drawn from. Color shows SAE activation, with purple = low and yellow = high.