SymTorch: A Framework for Symbolic Distillation of Deep Neural Networks

Elizabeth S.Z. Tan, Adil Soubki & Miles Cranmer
Department of Applied Mathematics and Theoretical Physics
University of Cambridge
United Kingdom
{eszt2,as3591,mc2473}@cam.ac.uk

Abstract

Symbolic distillation replaces neural networks, or components thereof, with interpretable, closed-form mathematical expressions. This approach has shown promise in discovering physical laws and mathematical relationships directly from trained deep learning models, yet adoption remains limited due to the engineering barrier of integrating symbolic regression into deep learning workflows. We introduce SymTorch, a library that automates this distillation by wrapping neural network components, collecting their input-output behavior, and approximating them with human-readable equations via PySR. SymTorch handles the engineering challenges that have hindered adoption: GPU-CPU data transfer, input-output caching, model serialization, and seamless switching between neural and symbolic forward passes. We demonstrate SymTorch across diverse architectures including GNNs, PINNs and transformer models. Finally, we present a proof-of-concept for accelerating LLM inference by replacing MLP layers with symbolic surrogates, achieving an 8.3% throughput improvement with moderate performance degradation.

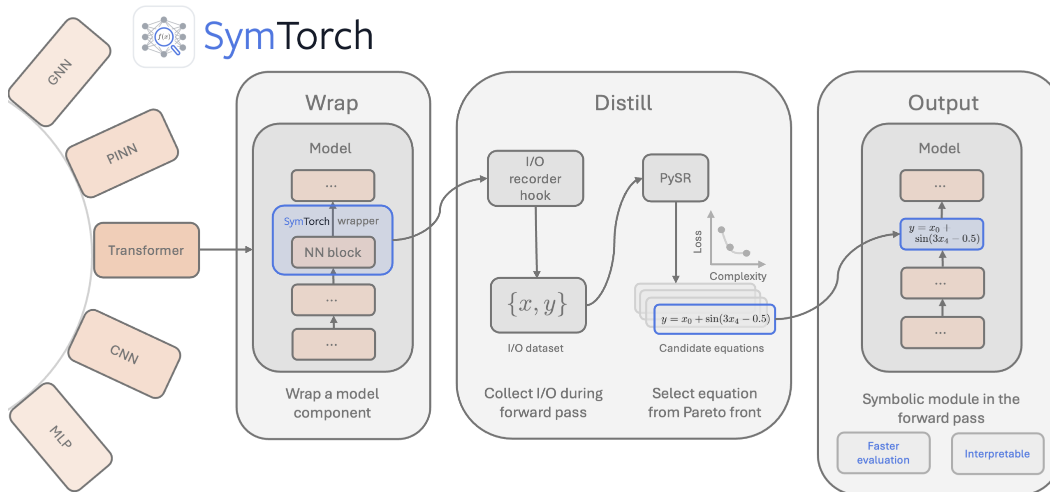

Figure 1: A cartoon depicting how SymTorch is used to perform symbolic distillation on model components. For a trained PyTorch model, SymTorch wraps around any NN component in the model. The user passes in sample data and in the forward pass, the inputs and outputs (I/O) of the component are collected. Using PySR, SymTorch performs a SR on the I/O to produce the best expressions approximating the behavior of the NN at different levels of complexity. Optionally, the user can select an equation from the Pareto front and replace the component with this chosen equation in the forward pass producing a hybrid neural-symbolic model.

1 INTRODUCTION

Deep learning models excel at analyzing large datasets, but remain largely uninterpretable. This has spurred extensive research into explainable AI. One promising direction is mechanistic interpretability, which focuses on identifying circuits (Elhage et al., 2021; Marks et al., 2025; Dunefsky et al., 2024), neurons, and activation dimensions (Heimersheim & Nanda, 2024; Makelov et al., 2023) within specific model architectures. These methods enable researchers to identify *which* components matter for specific behaviors. SymTorch addresses a complementary question: *what function* does a component compute? In doing so, we provide a more holistic approach to model interpretability.

In physics, interpretability means using concise equations that reliably explain phenomena (Kutz & Brunton, 2022). These equations are written in the language of mathematical operators and variables whose physical effects are understood. Newton's laws exemplify this: $F = ma$ is accurate in idealized settings yet deliberately inexact. Adding terms for drag or relativistic effects would sacrifice interpretability for minimal accuracy gains.

Inspired by this physics-centric view of interpretability, SymTorch brings a symbolic perspective to Neural Network (NN) analysis[1]. We use Symbolic Regression (SR) to distill NN components into human-readable mathematical formulas. Symbolic distillation provides architecture-agnostic interpretability by approximating component behavior with closed-form expressions. This enables direct inspection of input-output mappings and analysis of how input variations affect outputs. This is particularly useful in monitoring ML behavior under distribution shift. The ability to replace components with interpretable, efficient symbolic approximations could further support maintaining model reliability in non-stationary environments.

The user can further optionally replace these components with the symbolic expression in the forward pass. A practical consequence of this is potential inference speedup, as simple equations can be faster to evaluate than dense matrix operations. We explore this direction by presenting a framework to accelerate LLM inference through replacing Multi-Layer Perceptron (MLP) with symbolic surrogates.

SymTorch is built on PyTorch and automates the process of applying SR to arbitrary model components. It interfaces with PySR (Cranmer, 2023), which uses genetic algorithms to search the space of symbolic expressions. SymTorch lowers the barrier to applying SR to deep learning models by automating GPU–CPU data movement, caching model inputs and outputs, supporting native PyTorch model serialization, and allowing seamless transitions between the neural components and allowing seamless switching between neural components and symbolic surrogates during forward passes.

Our work makes two main contributions:

- **SymTorch Library:** We release an open-source PyTorch library that automates symbolic distillation of NN components. SymTorch handles the engineering challenges of GPU-CPU data transfer, activation caching, forward hook management, and hybrid neural-symbolic model serialization. Comprehensive documentation accompanies the package, including usage examples for all features and Jupyter notebooks that reproduce the case studies in this paper[2].
- **Comprehensive Case Studies:** To demonstrate SymTorch's capabilities, we present case studies across diverse architectures: (i) extracting empirical force laws from Graph Neural Networks (GNNs) trained on particle dynamics, (ii) recovering PDE solutions from Physics-Informed Neural Networks (PINNs), (iii) analyzing LLM-learned arithmetic operations (iv) a proof-of-concept approach to accelerate transformer inference by replacing MLP layers with symbolic surrogates, achieving measurable speedups with quantified accuracy tradeoffs.

By lowering the barrier to symbolic distillation, SymTorch enables researchers to apply this interpretability technique without reimplementing the underlying infrastructure.

2 BACKGROUND & RELATED WORK

2.1 ANALYSIS OF MODEL INTERNALS

Mechanistic interpretability research focuses on causal intervention, activation analysis, and circuit discovery. TransformerLens (Nanda & Bloom, 2022) provides interfaces for inspecting transformer

[1]Source code: `https://github.com/astroautomata/SymTorch`. Available on PyPI: `pip install torch-symbolic`.

[2]Documentation: `https://symtorch.readthedocs.io/en/latest/`

internals, while NNSight (Fiotto-Kaufman et al., 2025) offers general-purpose NN interpretability through deferred execution. These approaches primarily provide structural and causal insights into model behavior. In contrast, by fitting symbolic expressions directly to component input–output mappings, SymTorch focuses on recovering explicit functional forms.

2.2 Interpreting Whole Model Behavior

LIME and SHAP are common model-agnostic methods of explaining whole-model behavior. SHAP (Lundberg & Lee, 2017) assigns importance scores to inputs, whereas LIME (Ribeiro et al., 2016) fits a local linear approximation to the model by perturbing the model inputs around a point of interest.

Supralocal Interpretable Model-Agnostic Explanations Fong & Motani (2025) introduce Supralocal Interpretable Model-Agnostic Explanations (SLIME), a method similar to LIME. Rather than approximating local model behavior with linear models, SLIME employs symbolic surrogate models, enabling the capture of non-linear relationships. SLIME further augments the surrogate model training set with points sampled from the true data distribution, as opposed to only training on randomly perturbed samples as in LIME. SymTorch provides a native implementation of this approach for out-of-the-box use with black-box models.

2.3 Symbolic Regression with PySR

SymTorch extends PySR for deep learning applications. PySR performs multi-population evolutionary search over analytic expressions through mutation, crossover, simplification, and constant optimization. The algorithm maintains a Pareto front of expressions that best fit the data at different complexities.

Problem Formulation For some metric $\mathcal{L}: \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}$, inputs $\mathbf{x}_i = (x_{1,i}, \cdots, x_{d,i}) \subset \mathcal{X} \in \mathbb{R}^d$ and target variables $y_i \subset \mathcal{Y} \in \mathbb{R}$ for $i = 1, \cdots N$, SR aims to find g

$$g = \arg\min_{g \in \mathcal{S}} \sum_{i}^{N} \mathcal{L}(y_i, g(\mathbf{x}_i)), \tag{1}$$

where S is the set of closed-form analytic expressions. Metric $\mathcal{L}$ is the data-fitting loss (eg. mean-squared error) that often times contains a penalty on the complexity of g. In PySR, complexity is measured as the number of nodes when g is written out as an expression tree.

Choosing the Best Equation PySR selects the best equation by maximizing the fractional drop in log mean absolute error relative to an increase in model complexity. Specifically, it chooses the expression on the Pareto front that maximizes the score given by

$$\text{score} = -\frac{\log(\text{loss}_i/\text{loss}_{i-1})}{\text{complexity}_i - \text{complexity}_{i-1}}. \tag{2}$$

3 Methods

In this section, we describe how SymTorch wraps NN components, collects input–output activations, performs SR, and replaces model blocks with symbolic surrogates. We further explain SymTorch's built-in SLIME implementation for explaining local model behavior using analytical equations.

3.1 Distilling Neural Networks with SymTorch

`SymbolicModel` is the entry point for all SymTorch functionality. Inheriting from PyTorch's `nn.Module`, it can wrap around both PyTorch NNs and callable functions, provided these functions form a mapping

$$f: \mathbf{x}_i \to \mathbf{y}_i, \tag{3}$$

where $\mathbf{x}_i = (x_{1,i}, \cdots, x_{d,i})$ and $\mathbf{y}_i = (y_{1,i}, \cdots, y_{D,i})$. SymTorch expands the SR problem in Equation (1) for many-dimensional outputs by fitting a separate symbolic model to each output

dimension. We define the original function f as a model *block*. For PyTorch NNs, the wrapper registers forward hooks to record the block's inputs and outputs over user-supplied data.

The `distill` method performs SR by calling PySR with user-specified parameters (operators, genetic algorithm hyperparameters, fitness function). The outputs are fitted as closed-form equations of the block's input variables. Users may pass `variable_transforms` to create derived features (e.g., $r = \sqrt{\Delta x^2 + \Delta y^2}$), which can improve SR efficiency by structuring the search space.

Switching to Symbolic Approximations After fitting, `switch_to_symbolic` replaces the original block with expressions selected from the Pareto front based on their complexity, so that subsequent forward passes evaluate the symbolic equations in place of the original computation. If no complexity is specified, SymTorch chooses the best expression according to Equation (2). The model can continue training with fixed equations while other weights update. `switch_to_block` restores the original function. Since the model remains a native PyTorch module, it can be used with `torch.compile` for additional optimization.

Saving and Caching Inputs and outputs are cached, allowing SR to be rerun without requiring additional forward passes through the model. SymTorch further works with PyTorch's native model saving and loading functions.

3.2 SLIME IMPLEMENTATION

SymTorch includes a native implementation of SLIME (Fong & Motani, 2025) for local model explanations. Given a black-box model f and a point of interest $\mathbf{x}^*$, we fit a symbolic surrogate $s \in \mathcal{S}$ to approximate f in a neighborhood of $\mathbf{x}^*$.

The training dataset $\mathcal{D}$ consists of: (1) the J nearest neighbors to $\mathbf{x}^*$ from the data distribution, and (2) $N_{\text{synthetic}}$ Gaussian-sampled points around $\mathbf{x}^*$. The Gaussian variance defaults to half the variance of the J neighbors, so the approximation region is controlled by J. We fit s by minimizing:

$$s = \arg\min_{s \in \mathcal{S}} \sum_{z_i \in \text{synthetic}} \pi_x(z_i)[f(z_i) - s(z_i)]^2 + M \sum_{z_j \in \text{neighbors}} [f(z_j) - s(z_j)]^2,$$

where $\pi_x(z_i) = \exp(-||x_0 - x_i||^2/\sigma^2)$ is a proximity-weighted kernel and M weights points from the true distribution. The `distill` method supports SLIME via a built-in flag.
SLIME could be used to probe how a model's behavior changes under out-of-distribution inputs by constructing local symbolic surrogates around out-of-distribution points of interest.

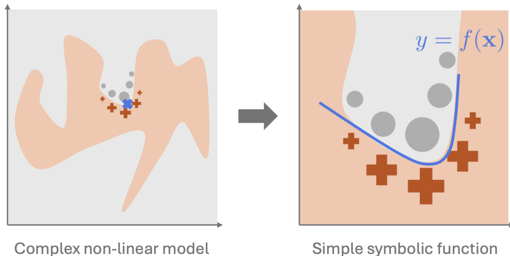

Figure 2: Approximating local model behavior with SLIME. For a complex non-linear model, we choose the point of interest $\mathbf{x}^*$. We sample points around this region and fit a symbolic model to these points.

4 CASE STUDIES AND PERFORMANCE

4.1 SYMBOLIC SURROGATES OF TRANSFORMER MLPS

MLP layers constitute a substantial portion of transformer inference compute (Wei et al., 2024), motivating techniques such as quantization (Dettmers et al., 2022), pruning (Lagunas et al., 2021),

and speculative decoding (Leviathan et al., 2023). We explore a different approach: replacing MLP layers with symbolic expressions via SymTorch[3].
The accuracy-speed tradeoff we achieve (8.3% speedup for a perplexity increase of 3.14 from baseline 10.62) may not be competitive with established methods. However, we present this framework because: (1) it demonstrates SymTorch's applicability to LLMs, (2) the primary driver of the performance degradation is the dimensionality reduction step whilst the replacement of the MLP with the SymTorch symbolic function only contributes to 1% of the perplexity increase, so (3) the approach may improve with better dimensionality reduction, and finally (4) regardless of the performance degradation, our SymTorch neuro-symbolic model is on the Pareto front of token throughput versus perplexity on Wikitext-2, when compared to similarly-sized open-source LLMs (see Figure 4). We further discuss these limitations and points for further research in this section.

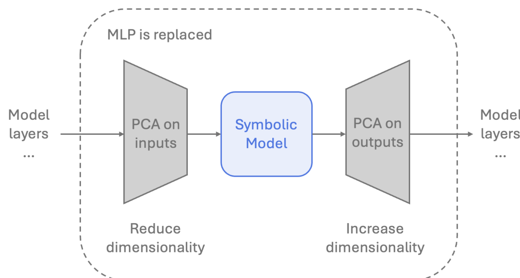

Figure 3: Framework for reducing inference compute in transformer models by replacing MLP layers with symbolic surrogate models. Inputs to the MLP layer undergo dimensionality reduction via PCA. A symbolic model maps the inputs to the outputs. Output activations have their dimensionality increased, again through PCA, to match model dimensionality.

Framework We first apply Principal Component Analysis (PCA) to the input and output activations of each MLP block. PCA provides a linear compression that preserves the dominant variance directions, allowing us to retain the most informative components of the representation while significantly reducing dimensionality. Despite the decrease in performance caused by this dimensionality reduction, this step is required to make SR more tractable. Then, we distill the MLP into a symbolic model using SymTorch. Generally, we want to be harsher on reducing the dimensionality of the MLP inputs as opposed to the outputs because SR scales exponentially with the number of input variables, and linearly with the number of output variables (as each output dimension requires a new expression).

Setup We used the Qwen2.5-1.5B-Instruct model (Qwen, 2024) for this experiment and the Wikitext-2-v1 dataset (Merity et al., 2016). The dataset was split into a training and test set each consisting of 178k tokens. The training set was used to train the PCA models and provide the sample data for the SR. We used perplexity to quantify the changes in performance of the LLM. We chose to intervene on MLP layers (SwiGLU activations) 7, 14 and 21. Before performing SR, we first determine the minimum number of principal components required to preserve the behavior of the model. To this end, we vary the dimensionality used for PCA on both the input and output activations of the MLP and measure the resulting change in model perplexity. Specifically, the input activations are projected to a lower-dimensional subspace via PCA and then reconstructed before being passed through the MLP; similarly, the MLP outputs are projected and reconstructed prior to being passed to the remainder of the model. All interventions on the transformer model are performed using forward hooks and pre-forward hooks, enabling localized replacement of MLP components without modifying the surrounding architecture. Additional experimental details and extended results, including the PCA sensitivity analysis, are provided in Section D.

Symbolic Surrogate Model Based on the PCA sensitivity analysis, we select 32 principal components for the MLP inputs and 8 for outputs. This configuration incurs a notable perplexity increase (+3.11 from baseline 10.62) but reduces the SR problem to tractable dimensions.
Using SymTorch, we fit symbolic surrogates mapping reduced inputs to reduced outputs. After

[3]Source code: `https://github.com/astroautomata/LLM_PCA`

Table 1: Perplexity comparison between baseline, PCA+MLP, PCA+SymTorch and Control using the test set. There were 32 and 8 principal components for the input and output PCAs respectively. For the Control result, the MLPs were replaced with identity functions.

Perplexity Baseline	Δ Perplexity PCA+MLP	Δ Perplexity PCA+SymTorch	Δ Perplexity Control
10.62	+3.11	+3.14	+6.97

substitution, the perplexity increase (+3.14) is comparable to PCA compression alone (+3.11), as shown in Table 1. This indicates the symbolic model captures the MLP's behavior within the reduced subspace, with minimal additional degradation from the symbolic approximation itself.

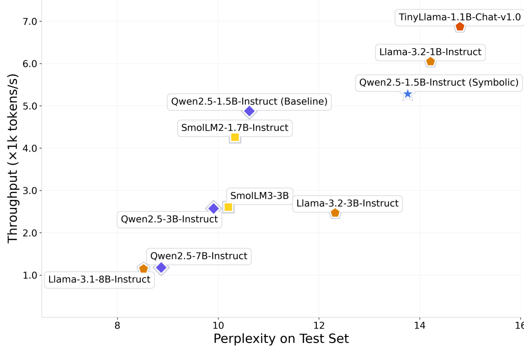

Figure 4: Inference throughput (in tokens per second) versus perplexity on the test set for various language models. The test set is made up of a random chunk of 175k characters from the Wikitext-2-v1 dataset.

Inference Speed Benchmarking To measure the impact of this symbolic surrogate intervention on inference speed, we compute token throughput of the model in three conditions: (1) a baseline with no intervention, (2) a control where MLPs are replaced by an identity function, and (3) with the symbolic surrogates replacing the MLPs. The results find that SymTorch distillation of 3/28 layers delivers an 8.3% increase in token throughput over the baseline, roughly matching the inference speed of the control condition. We performed an identical inference speed benchmarking for various popular and small open-source LLMs, and measured their perplexity on the test set. The results are shown in Figure 4. Notably, even with the perplexity increase from symbolic replacement, the modified Qwen2.5-1.5B model remains on the Pareto front of the perplexity-throughput tradeoff.

4.2 Discovering Empirical Physical Laws from Symbolic Distillation of Models with Inductive Biases

We demonstrate SymTorch on a reproduction and slight extension of a prior work by Cranmer et al. (2020): training a GNN on empirical particle data and performing SR on the latent representations of the trained model to recover true interaction laws[4]. Complete details of the replication including training details, experimental results, and framework limitations, are provided in Section E.

Setup The GNNs consist of two components: the edge model and the node model, both of which are MLPs. The edge model, ϕ^e, maps the information from two nodes ($\mathbf{v}_i, \mathbf{v}_j \in \mathcal{V} \subset \mathbb{R}^{L^v}$), connected with an edge and outputs a message vector for this edge. The node model, ϕ^v, takes the aggregated message vectors to a receiving node and the information of the receiving node as inputs and outputs the updated node state, $\mathbf{v}_i'$, which is the output of the GNN. We train the model by minimizing the mean absolute error between $\mathbf{v}_i'$ and the target variable $\hat{\mathbf{v}}_i'$.
In our setup, the node features contain information about a particle:

$$\mathbf{v}_i = [x_i, y_i, \dot{x}_i, \dot{y}_i, q_i, m_i], \tag{4}$$

where x_i, y_i are the 2-D positions of the particle i, $\dot{x}_i$ and $\dot{y}_i$ are the discretised velocities (distance moved in simulation time Δt), q_i is the charge and m_i is the mass. Hence, the dimensionality of the

[4]Source code: https://github.com/astroautomata/SymTorch_symbolic_distillation_GNNs

node features, L^v, depends on the dimensionality of the system. The inputs to the model comprise this information. The target variables in this pipeline, $\hat{\mathbf{v}}_i'$, are the individual particle accelerations. We test out this framework to on particles acting under the following pair-wise interaction forces: (a) gravitational force; (b) spring force; (c) force that scales as $1/r$ and (d) force that scales as $1/r^2$, where r is the displacement between the two particles.

Edge Messages as Forces For GNNs trained in this manner, it can be shown that the edge messages are linear combinations of the true forces acting between particles, provided the dimensionality of the edge messages are the same as that of the system. The full derivation for this result is in Section E.2. Because of this, we trained several several variants of the GNN to explore different methods to encourage sparse representation of the edge messages.

Symbolic Distillation with SymTorch For a trained GNN, SymTorch wraps the edge model to enable direct symbolic distillation of the MLP. Using this framework, we reproduce the symbolic recovery results reported in Cranmer et al. (2020). SymTorch substantially simplifies experiments of this kind by removing the need for manual extraction of input–output activations from model components. In addition, the framework supports user-defined transformations of input variables, which further improve the tractability of SR in this setting.

4.3 Extracting PDE Solutions from a PINN

We compared a Physics-Informed Neural Network (PINN) with a standard NN of identical architecture for predicting temperatures governed by the 1-D heat equation. While both networks were trained directly on the same data, the PINN incorporated knowledge of the system's governing Partial Differential Equation (PDE), Initial Conditions (ICs), and Boundary Conditions (BCs) via additional regularization terms in its loss. Trained on only ten data points, the PINN achieved substantially better predictions than the standard network, shown in Figure 8, as its inductive bias enforced physically consistent solutions. Using SymTorch, we further distilled the trained PINN into a closed-form analytic expression, successfully recovering the solution to the 1-D heat equation. Using a PINN is advantageous compared to applying SR directly to the dataset, as the PINN can generate additional training data that guides symbolic regression toward equations consistent with the underlying physical solution. More information on this experiment can be found in Section F.

4.4 Interpreting LLM-Learned Operations

LLMs often fail at elementary numerical tasks, such that commercial systems rely on external tool calls for reliable computation (Dziri et al., 2023; Golkar et al., 2024). In this case study, we use SR to analyze the LLM-learned operations to analyze the true function computed by the LLM. Symbolic distillation is well-suited to this task: rather than treating the LLM as a black-box that is only right or wrong, we can recover an explicit analytic approximation of the computation it is performing. This enables direct inspection of systematic numerical errors and reveals how, and where, the model's internal heuristics deviate from the intended operations.

Setup We study the model Llama-3.2-1B-Instruct as a representative small LLM and analyze the operation the model has learnt for (a) addition of two 3-digit numbers; (b) multiplication of two 3-digit numbers; (c) counting the number of 1s in a string of 1s and 0s; (d) converting Celsius to Fahrenheit. More information on the experiment including specific prompts and SR parameters can be found in Section G.

Results The results of this analysis can be found in Table 2. It should be noted that the correct equation was present in the Pareto front of the SR for all of these tasks except counting. However it was not the chosen 'best' equation suggesting that the LLM gets these tasks mostly correct with the addition of some systematic errors. It is known that the expressivity of finite-precision transformers (Chiang et al., 2023), or even log-precision transformers (Merrill & Sabharwal, 2023), is limited to variants of first-order logic with counting or majority quantifiers. These logics cannot, in general, implement an algorithm to perform exact counting over arbitrary-length inputs, which might be related to why this task proved more difficult.

5 Discussion

SymTorch demonstrates that symbolic regression can serve as a practical interpretability tool when integrated with deep learning frameworks. Our case studies show successful symbolic distillation

Table 2: Symbolically distilled LLM-learned operations. The best equation, as determined by Equation (2), is shown here. The ϵ denotes a small ($\ll 1$) term.

Operation	Expected Equation	LLM-Learned Operation
Addition	$x_0 + x_1$	$x_1 \cdot ((\text{inv}(x_0 - 70.16) + 1.07) + (\text{inv}(\sin(x_1) + 0.80) + x_1) \cdot (-\epsilon))\, x_0$
Multiplication	$x_0 x_1$	$x_1(\text{inv}(1.66 - 31.3 \sin(x_0)) + x_0)$
Counting	$x_0 + \cdots + x_5$	$(-0.11x_0 + 1.67)(x_1 + (x_1(x_0 + 1.43) - 2.50)(x_2 - 0.53)(x_0 + x_3 - 2.15) + x_2)$
Temperature Conversion	$\frac{9}{5}x + 32$	$\text{inv}(x_0 - 168.86)\text{inv}(x_0 - 129.65) + \text{inv}(x_0 - 51.87)x_0 + 33.79$

across GNNs, PINNs, and LLMs, recovering known physical laws and revealing LLM arithmetic biases. We have further demonstrated a novel framework to speed up LLM inference by replacing MLP blocks with symbolic expressions found using SymTorch. Despite the performance degradation due to the reduced expressive capacity of symbolic expressions relative to dense neural layers, the resulting neuro-symbolic model remains on the Pareto front of token throughput versus perplexity when compared to popular LLMs of similar scale on the WikiText-2 benchmark.
By handling the engineering complexity of GPU-CPU transfer, activation caching, and hybrid model management, SymTorch lowers the barrier to symbolic distillation. We hope this accessibility enables SR to become a practical tool for uncovering the functional structure learned by deep models.

5.1 Limitations & Future Work

Computation & Hyperparameters: SR is a brute-force process. Runtime grows exponentially with the number of variables, operator set size, and dataset size. For complex NNs, obtaining accurate symbolic approximations may require large datasets and extended SR runs. We also perform SR per output dimension, which becomes expensive for wide outputs. Hyperparameter tuning may further be necessary to ensure human-readable outputs and tractable SR.

Highly Complex Functions: For highly complex model components, SR may struggle to produce accurate approximations. Nevertheless, it remains one of the few viable routes to functional interpretability; without it, such components would remain entirely opaque.

Symbolic Surrogates of Transformer MLPs: The framework we propose for accelerating LLM inference may not be competitive with existing approaches. In particular, the performance degradation introduced by symbolic approximation is non-trivial. Furthermore, we evaluate model quality only on WikiText-2, which matches the training distribution of the symbolic surrogates, and performance may degrade further under distribution shift or on downstream tasks. Despite these limitations, there exist important regimes where throughput is prioritized over general-purpose flexibility. In many deployment settings, models are fine-tuned for a single downstream task, for example Low-Rank Adaptation (LoRA) (Hu et al., 2021), and are not required to generalize broadly. In such scenarios, sacrificing some expressive capacity in exchange for substantial inference speedups may represent a favorable trade-off, particularly for latency-critical or resource-constrained applications. We emphasize that this framework is in no way optimized; rather, its primary purpose is to illustrate the experimental workflows made accessible by SymTorch. Hence, possible areas for future work to improve this framework include,

- **Improved Dimensionality Reduction:** The PCA-based compression in Section 4.1 is the major source of performance degradation. Learned linear projections may preserve more relevant information while maintaining SR tractability.
- **Cross-Domain Generalization:** The symbolic surrogates for LLM components are trained and evaluated on the same distribution, leaving open the question of how well such surrogates generalize across domains. We hope to determine whether domain-agnostic symbolic approximations are feasible, or whether task- and domain-specific surrogates are necessary.
- **Framework Optimization and Extension:** We arbitrarily replace three MLP layers in the LLM, and future work will explore which layers are most amenable to symbolic approximation and how many layers can be replaced before performance degrades substantially. We further want to test out this framework on larger LLMs.

REFERENCES

Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks, 2018. URL `https://arxiv.org/abs/1806.01261`.

David Chiang, Peter Cholak, and Anand Pillay. Tighter bounds on the expressivity of transformer encoders, 2023. URL `https://arxiv.org/abs/2301.10743`.

Miles Cranmer. Interpretable machine learning for science with pysr and symbolicregression.jl, 2023. URL `https://arxiv.org/abs/2305.01582`.

Miles Cranmer, Alvaro Sanchez-Gonzalez, Peter Battaglia, Rui Xu, Kyle Cranmer, David Spergel, and Shirley Ho. Discovering symbolic models from deep learning with inductive biases, 2020. URL `https://arxiv.org/abs/2006.11287`.

Miles D. Cranmer. Discovering symbolic models from deep learning with inductive biases code repository, 2020. URL `https://github.com/MilesCranmer/symbolic_deep_learning`.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022. URL `https://arxiv.org/abs/2208.07339`.

Jacob Dunefsky, Philippe Chlenski, and Neel Nanda. Transcoders find interpretable llm feature circuits, 2024. URL `https://arxiv.org/abs/2406.11944`.

Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Sean Welleck, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. Faith and fate: Limits of transformers on compositionality, 2023. URL `https://arxiv.org/abs/2305.18654`.

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html.

Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric, 2019. URL `https://arxiv.org/abs/1903.02428`.

Jaden Fiotto-Kaufman, Alexander R. Loftus, Eric Todd, Jannik Brinkmann, Koyena Pal, Dmitrii Troitskii, Michael Ripa, Adam Belfki, Can Rager, Caden Juang, Aaron Mueller, Samuel Marks, Arnab Sen Sharma, Francesca Lucchetti, Nikhil Prakash, Carla Brodley, Arjun Guha, Jonathan Bell, Byron C. Wallace, and David Bau. Nnsight and ndif: Democratizing access to open-weight foundation model internals, 2025. URL `https://arxiv.org/abs/2407.14561`.

Kei Sen Fong and Mehul Motani. Slime: Supralocal interpretable model-agnostic explanations via evolved equation-based surrogates. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '25 Companion, pp. 267–270, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400714641. doi: 10.1145/3712255.3726721. URL `https://doi.org/10.1145/3712255.3726721`.

Siavash Golkar, Mariel Pettee, Michael Eickenberg, Alberto Bietti, Miles Cranmer, Geraud Krawezik, Francois Lanusse, Michael McCabe, Ruben Ohana, Liam Parker, Bruno Régaldo-Saint Blancard, Tiberiu Tesileanu, Kyunghyun Cho, and Shirley Ho. xval: A continuous numerical tokenization for scientific language models, 2024. URL `https://arxiv.org/abs/2310.02989`.

Stefan Heimersheim and Neel Nanda. How to use and interpret activation patching, 2024. URL `https://arxiv.org/abs/2404.15255`.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

J. Kutz and Steven Brunton. Parsimony as the ultimate regularizer for physics-informed machine learning. *Nonlinear Dynamics*, 107:1–17, 01 2022. doi: 10.1007/s11071-021-07118-3.

François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M. Rush. Block pruning for faster transformers, 2021. URL https://arxiv.org/abs/2109.04838.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, 2023. URL https://arxiv.org/abs/2211.17192.

Scott Lundberg and Su-In Lee. A unified approach to interpreting model predictions, 2017. URL https://arxiv.org/abs/1705.07874.

Aleksandar Makelov, Georg Lange, and Neel Nanda. Is this the subspace you are looking for? an interpretability illusion for subspace activation patching, 2023. URL https://arxiv.org/abs/2311.17030.

Samuel Marks, Can Rager, Eric J. Michaud, Yonatan Belinkov, David Bau, and Aaron Mueller. Sparse feature circuits: Discovering and editing interpretable causal graphs in language models, 2025. URL https://arxiv.org/abs/2403.19647.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

William Merrill and Ashish Sabharwal. A logic for expressing log-precision transformers. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 52453–52463. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/a48e5877c7bf86a51395.ab23b360498-Paper-Conference.pdf.

Neel Nanda and Joseph Bloom. Transformerlens. https://github.com/TransformerLensOrg/TransformerLens, 2022.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Qwen. Qwen2.5: A party of foundation models, September 2024. URL https://qwenlm.github.io/blog/qwen2.5/.

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "why should i trust you?": Explaining the predictions of any classifier, 2016. URL https://arxiv.org/abs/1602.04938.

Xiuying Wei, Skander Moalla, Razvan Pascanu, and Caglar Gulcehre. Building on efficient foundations: Effectively training llms with structured feedforward layers, 2024. URL https://arxiv.org/abs/2406.16450.

A APPENDIX A

B SYMBOLIC REGRESSION WITH PYSR DETAIL

B.1 ALGORITHM OVERVIEW

PySR implements the following genetic algorithm (summarized from Cranmer (2023)):

1. We begin with several independent populations of individual equations. This allows expressions in each population to evolve simultaneously.
2. In each population, a tournament selection process is run: a random subset of individuals (usually two) are evaluated on their 'fitness', a metric combining both expression accuracy and complexity. The complexity of an expression is determined by the number of nodes on the binary expression tree.
3. The fittest individual in the subset is chosen as the winner with probability p; otherwise the next fittest is chosen with the same probability, and so on.
4. A copy of the winning individual undergoes some form of mutation, where a node on the expression tree can be changed, a node added or removed. Or the individual may undergo a cross-over operation with the next fittest expression, where parts of the expression tree may swap between these two individuals. This mutation may be accepted or rejected with a probability relating to the fitness of the mutated expression.
5. A set number of tournaments constitutes a round of evolution for the population. At the end of each evolution round, expressions may be simplified to an equivalent form (for example $x + x + x \to 3 * x$) or the constants may be optimized.
6. After a specified number of evolutions, individuals may migrate between populations.

At each iteration, the Pareto front is updated with the best-performing equations at different complexity levels.

C SYMTORCH EXTRA DETAILS

C.1 DEFAULT PYSR PARAMETERS

Users configure SR via the `sr_params` dictionary in `distill` (defaults in Table 3). These parameters go directly to PySR's `PySRRegressor`. Parameters for the `fit` method (e.g., variable names or complexities) use the `fit_params` dictionary.

Table 3: Default SymTorch SR configurations.

SR Parameter	**Configuration**
Binary operators	`+, *`
Unary operators	`inv(x)=1/x, sin, exp`
Extra sympy mappings	`"inv": lambda x: 1/x`
Number of iterations	400
Complexity of operators	`sin: 3, exp: 3`

C.2 SAVING SR RESULTS AND SYMBOLIC MODELS

SymTorch saves PySR SR results in `SR_output/mlp_name`, organized by output dimension and timestamp. SymTorch works seamlessly with PyTorch's `torch.save` and `torch.load` functions.

D SYMBOLIC SURROGATES OF TRANSFORMER MLPS DETAILS

D.1 SETUP

Dataset We used the Wikitext-2-v1 dataset which was already split into train, test and validation sets. In this experiment, only the train and validation (quoted as the 'test set' in this paper) sets were used. We further selected a random chunk of 750k characters from the each dataset which corresponded to approximately 178k tokens. The PCA and symbolic models were trained purely on this subset of the validation set.

PCA Sensitivity Analysis To train the PCA models, we used Scikit-learn (Pedregosa et al., 2011). The input data to the PCA were centred but not whitened (the PCA components were not scaled to have unit variance).
Figure 5 shows the changes in perplexity as we varied the number of principal components of the input and output to the MLP. Just for reference, the original SwiGLU projects inputs from a 1536 dimensional subspace to an increased 8960 dimensional subspace and back down again and the baseline model perplexity on the test set is given in Table 1. Notably, for a fixed number of PCA components retained at the input, the change in perplexity initially decreases as the number of output PCA components increases, before rising again. One possible explanation is that moderate output compression removes low-variance or noisy directions while preserving the dominant functional subspace, whereas excessive expansion reintroduces poorly conditioned or misaligned directions that degrade downstream representations. The explained variance ratio from the PCA sensitivity analysis

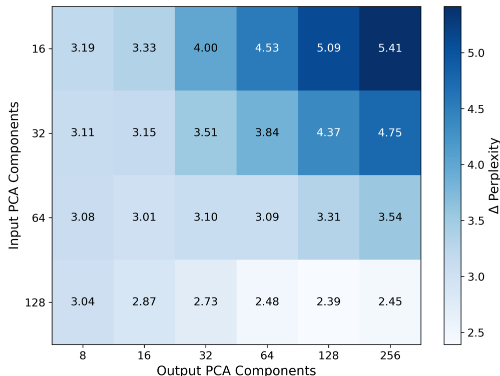

Figure 5: Change in test set perplexity under PCA compression and reconstruction of MLP activations relative to the baseline perplexity of 10.62. For layers 7, 14, and 21, the MLP inputs are projected to a lower-dimensional subspace via PCA and then reconstructed prior to the MLP, while the MLP outputs are similarly projected and reconstructed before being passed to the remainder of the model.

is shown in Figure 6.

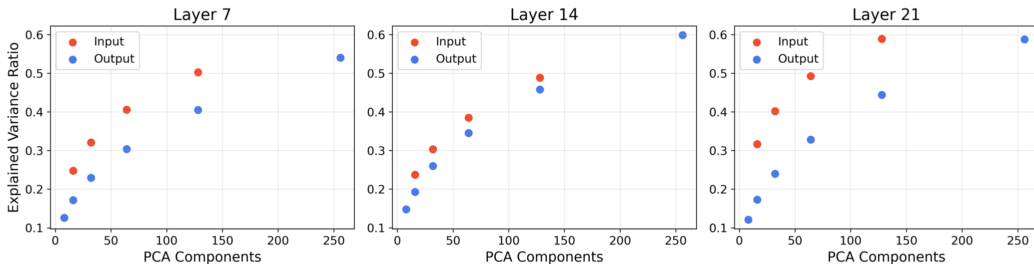

Figure 6: The explained variance ratio for the PCA models trained on the pre-activations and activations of the MLP layers for the PCA sensitivity analysis.

Training the Symbolic Models After the PCA models were trained, we defined a Python callable function that took the MLP reduced dimensionality pre-activations as input and returned the MLP reduced dimensionality activations. SymTorch was used to wrap and perform SR on this function. We used a random subset of 6000 samples from the training set to perform the SR. The SR parameters used were the default parameters in Table 3 but we increased the number of iterations to 5000.

Results and Benchmarking When calculating the perplexity values across the dataset, chunk sizes of 1048 were used. The perplexity results of this experiment on the training set are shown in Table 4. We further ran an inference speed versus perplexity on the Wikitext validation set benchmarking on popular open-source LLMs, as shown in Figure 4. Benchmarking was performed by first running five warm-up passes, followed by timing 100 forward passes of the test set through the model with KV caching disabled. The elapsed times were averaged to estimate token throughput for each condition. This benchmark was run on an NVIDIA A100-SXM4-80GB GPU for comparison.

Table 4: Perplexity comparison between baseline, PCA+MLP, PCA+SymTorch and Control using the training set. There were 32 and 8 principal components for the input and output PCAs respectively. For the Control result, the MLPs were replaced with identity functions.

Perplexity Baseline	Δ Perplexity PCA+MLP	Δ Perplexity PCA+SymTorch	Δ Perplexity Control
10.74	+3.60	+3.63	+8.12

SR Wall Clock Time To fit the SymTorch surrogate model to the three MLP blocks it took between 7-8 hours on an Apple M4 Max SoC (14-core CPU: 10 performance + 4 efficiency cores, 36 GB unified memory).

E Discovering Empirical Physics Laws from Symbolic Distillation of Models with Inductive Biases Details

E.1 Model Architecture

We follow the notation as outlined in Battaglia et al. (2018). There are two MLPs involved in the GNN. The first is the edge model (or edge function), $\phi^e : \mathcal{V} \times \mathcal{V} \to \mathcal{E}$, where $\mathcal{V} \subset \mathbb{R}^{L^v}$ and $\mathcal{E} \subseteq \mathbb{R}^{L^e}$. L^v and L^e are the number of node features and the dimensionality of the messages respectively. In a forward pass of the GNN, the edge model takes the node features, $\mathbf{v} \in \mathcal{V}$, of two connected nodes as inputs and outputs the edge message, $\mathbf{e}'_k \in \mathcal{E}$, where k denotes the edge. We concatenate the node features when inputting them into the MLP. This function can be written as

$$\mathbf{e}'_k = \phi^e(\mathbf{v}_{r_k}, \mathbf{v}_{s_k}), \tag{5}$$

where $\mathbf{v}_{r_k}$ denotes the features of the receiving node, and $\mathbf{v}_{s_k}$ denotes the sending node of edge k. The messages to receiving node i are aggregated through an element-wise summation,

$$\bar{\mathbf{e}}'_i = \sum_{j \neq i} \phi^e(\mathbf{v}_i, \mathbf{v}_j), \tag{6}$$

where $\bar{\mathbf{e}}'_i$ is the aggregated message.
The second MLP involved in the GNN is the node model (or node function), $\phi^v : \mathcal{V} \times \mathcal{E} \to \mathcal{D}$, where $\mathcal{D} \subseteq \mathbb{R}^D$ and D is the dimensionality of the target variable, the updated node features. This model outputs the updated node features for a specific node. It takes the node features and the aggregated message for that node as an input and calculates the node update, $\hat{\mathbf{v}}'_i$:

$$\hat{\mathbf{v}}'_i = \phi^v(\mathbf{v}_i, \bar{\mathbf{e}}'_i). \tag{7}$$

E.2 Derivation: Edge Messages as Linear Transformations of the True Forces

For a GNN that has learned to predict accelerations from particle properties, the message vectors are linear transformations of the true forces, provided that the message dimension is equal to the

Table 5: Perplexity on the test set (as used in Section 4.1) and inference benchmarking for various open-source language models.

Model	Perplexity on Validation Set	Avg. Latency (ms)	p95 Latency (ms)	Throughput (tokens/s)
Qwen2.5-1.5B (Baseline)	10.62	209.89	211.76	4878.82
Qwen2.5-1.5B (Symbolic)	13.76	193.89	195.34	5281.42
Qwen2.5-3B-Instruct	9.91	397.85	401.55	2573.84
Qwen2.5-7B-Instruct	8.87	866.06	867.71	1182.36
Llama-3.1-8B-Instruct	8.52	891.84	892.89	1148.18
Llama-3.2-1B-Instruct	14.21	169.31	170.08	6048.19
Llama-3.2-3B-Instruct	12.32	414.51	416.03	2470.36
SmolLM2-1.7B-Instruct	10.33	240.48	240.89	4258.23
SmolLM3-3B	10.20	392.89	393.52	2606.35
TinyLlama-1.1B-Chat-v1.0	14.80	149.00	149.44	6872.68

dimensionality of the forces. The mathematical derivation for this is, as first presented in Cranmer et al. (2020), is shown below.

In Newtonian mechanics, the resultant force, $\bar{\mathbf{f}}_i$, acting on particle i is equal to the sum of the individual forces, $\mathbf{f}_k$:

$$\sum_k \mathbf{f}_k = \bar{\mathbf{f}}_i, \tag{8}$$

If we ignore the particle mass, the node model predicts the resultant force on the receiver particle r_k:

$$\bar{\mathbf{f}}_i = \hat{\mathbf{v}}'_i = \phi^v(\mathbf{v}_i, \bar{\mathbf{e}}'_i) = \phi^v\left(\mathbf{v}_i, \sum_{r_k=i} \mathbf{e}'_k\right). \tag{9}$$

If we only consider a single interaction, and hence a single edge, the force is

$$\bar{\mathbf{f}}_i = \mathbf{f}_{k,r_k=i} = \phi^v(\mathbf{v}_i, \mathbf{e}'_{k,r_k=i}). \tag{10}$$

Again, the resultant force is the sum of the individual forces, so we can use the above equation in the many-particle case and equate this to Equation (9). Explicitly,

$$\sum_{r_k=i} \phi^v(\mathbf{v}_i, \mathbf{e}'_k) = \phi^v\left(\mathbf{v}_i, \sum_{r_k=i} \mathbf{e}'_k\right) = \bar{\mathbf{f}}_i, \tag{11}$$

which demonstrates that ϕ^v is a linear function in its second argument:

$$\phi^v(\mathbf{v}_i, \mathbf{e}'_a + \mathbf{e}'_b) = \phi^v(\mathbf{v}_i, \mathbf{e}'_a) + \phi^v(\mathbf{v}_i, \mathbf{e}'_b). \tag{12}$$

Provided ϕ^v is invertible in $\mathbf{e}'_k$, which is true when the dimensionality of $\mathbf{e}'_k$ matches the dimensionality of $\hat{\mathbf{v}}'_i$, then you can invert Equation (10):

$$\mathbf{e}'_k = (\phi^v(\mathbf{v}_i, \cdot))^{-1}(\mathbf{f}_k). \tag{13}$$

Hence, the messages $\mathbf{e}'_k$ are just linear transformations of the true forces $\mathbf{f}_k$. If we constrain the message dimensionality to match that of the physical system, we can fit the learned messages using a linear regression on the true forces. A strong linear correspondence indicates that the model has successfully captured the underlying physical forces.

E.3 Why the GNN is required

We note that the use of a GNN is essential for this problem, as applying symbolic regression directly to the raw dataset fails to recover the true interaction forces; assume a dataset $\{(y_i, \{\mathbf{x}_{1,i}, \cdots, \mathbf{x}_{100,i}\}\}_{i=1\cdots N}$ where y_i is the target variable which could for example be the acceleration of a single particle in a system of 100 particles. Assume the true relationship to be a composite function $a\left(\sum_j b(\mathbf{x}_i, \mathbf{x}_j)\right)$. If the SR needs to consider N equations for functions a and b then we must consider N^2 equations to get the true expression for y. In contrast, when a and b can be fitted independently - as in our GNN architecture, where separate MLPs learn each function - the search space decomposes, and only $2N$ candidate expressions need to be considered. Hence the GNN makes SR much more tractable (Cranmer et al., 2020).

E.4 Model Variants

Details of the GNN variants trained in this case study are as follows:

1. **Standard**. Consisted of a GNN where the message dimension, L^e, was set to 100.
2. **Bottleneck**. The message dimension was set to match the dimensionality of the system, which was 2 in all the experiments.
3. **L1**. We applied L1 regularization, $\mathcal{L}_e$ to the edge messages,
$$\mathcal{L}_{L1} = \frac{\alpha_{L1}}{N^e} \sum_{k=0}^{N^e-1} |\mathbf{e}'_k|, \tag{14}$$
where N^e is the total number of parameters in the edge messages, and $\alpha_{L1} = 10^{-2}$ is the regularization constant. The dimension of the message vectors were set to be 100, the same as the standard model variation. This regularization encourages sparse representation of messages by the absolute values of the message components, effectively driving many of them toward zero.
4. **Kullback–Leibler (KL)**. We added a standard Gaussian prior, $\mathcal{N} \sim (\mathbf{0}, \mathbf{1})$, to the components of the messages. Unlike the other model variants, the edge model in this case outputs both the mean and log-variance for each message element, which doubles the output dimensionality. This allows the messages to be treated as samples from a Gaussian distribution rather than fixed feature vectors,
$$\mathbf{e}'_k \sim \mathcal{N}(\boldsymbol{\mu}'_k, \mathbf{diag}[\boldsymbol{\sigma}'^2_k]), \tag{15}$$
where $\boldsymbol{\mu}'_k = \phi^e_\mu$ and $\boldsymbol{\sigma}'^2_k = \exp(\phi^e_{\sigma^2})$. We trained the model such that ϕ^e_μ are all even outputs and $\phi^e_{\sigma^2}$ are all odd outputs from the edge model. During training, the edge messages are sampled from the distribution defined by Equation (15) before being aggregated and inputted to the node model. A regularization term equivalent to the KL-divergence between the standard normal prior and the probability distribution defined in Equation (15) is added to the loss:
$$\mathcal{L}_{KL} = \frac{1}{N^e} \sum_{k=0}^{N^e-1} \sum_{j=0}^{L^e-1} \frac{1}{2}(\mu'^2_{k,j} + \sigma'^2_{k,j} - \log(\sigma'^2_{k,j})). \tag{16}$$
The KL regularization term encourages sparsity in the messages by penalizing deviations from a standard normal distribution, effectively pushing the learned mean and variance of each message component toward zero mean and unit variance. If the model is in an evaluation setting, we do not sample and just take the message elements to equal the means.
5. **Pruning**. The dimensionality of the edge messages is gradually reduced during training until it matches that of the system using SymTorch's pruning functionality. To prune the MLP, we chose a random 10,240 data points from the validation set as sample data points and inputted these through the MLP. The dimensions with the lowest variance across these sample points were deemed as 'unimportant' and we zero-masked these dimensions. We used a Cosine Annealing pruning schedule, with pruning completing at 65% of the way through training. SymTorch contains its own built-in method to prune MLPs in this way. This model variation is an extension to the original paper. Pruning is a similar model variation to bottleneck, but has the advantage that we do not require knowledge of the dimensionality of the system beforehand.

E.5 Setup and Training

Dataset The dataset used in training the GNN was created using the source code from the original paper code repository (Cranmer, 2020). Each sample comprised a tensor containing particle information, as shown in Equation (4), of four interacting particles. The target data included the accelerations of the particles, found by taking the negative derivative of their respective potentials (to calculate the force) and dividing by their mass. Our system has dimensionality of two.
Each dataset was created by running 10,000 different simulations over 1000 time steps and then only collecting data from every 5th time step to reduce correlation between samples. This resulted in datasets containing 1 million samples, which were then randomly split into training, validation and test sets with ratio 70/15/15.

Model Configuration To create and train the GNNs, we used PyTorch and PyTorch Geometric (Fey & Lenssen, 2019). In all experiments, the edge model and node model MLPs contained three hidden layers each with 300 hidden units and ReLU activations between each layer.

Data Augmentation The node model outputs predictions of the instantaneous accelerations of particles, as shown in Section E.1. Before passing the node features into the model, we augment the data by adding Gaussian noise with a standard deviation of 3 to the position coordinates of all nodes simultaneously. This follows the approach used in the original paper's code and was likely introduced to improve model robustness by reducing overfitting to precise spatial positions, while also simulating the presence of noise in real-world data.

Predictions and Loss The base loss on our model was calculated to be the mean absolute error between the predicted accelerations, $\hat{\mathbf{v}}'_i$, and the actual accelerations, $\mathbf{v}'_i$, from our dataset;

$$\mathcal{L} = \frac{1}{N^v} \sum_{i=0}^{N^v-1} |\mathbf{v}'_i - \hat{\mathbf{v}}'_i|. \tag{17}$$

L2 regularization was also used in every training instance,

$$\mathcal{L}_{L2} = \frac{\alpha_{L2}}{N^l} \sum_{l=0}^{N^l} |w_l|^2, \tag{18}$$

where N^l are the total number of network parameters, denoted w_l, and $\alpha_{L2} = 10^{-8}$ is the L2 regularization constant. Hence, the total loss is $\mathcal{L} + \mathcal{L}_{L2}$ for the standard, bottleneck and pruning model variant. For the L1 model variation, the loss is $\mathcal{L} + \mathcal{L}_{L2} + \mathcal{L}_{L1}$, and for the KL model variation the loss is $\mathcal{L} + \mathcal{L}_{L2} + \mathcal{L}_{KL}$.

Training To train our models, we performed gradient descent using the Adam (Kingma & Ba, 2017) optimiser with a Cosine Annealing learning rate scheduler. Each model was trained for 100 epochs with a batch size of 64 on a training set of 700,000 samples, resulting in approximately 1.1 million optimization steps. The training and validation loss was monitored every epoch.

E.6 SYMBOLIC DISTILLATION WITH SYMTORCH

We used SymTorch to perform the SR on the edge model. For the standard and L1 model variant, we performed SR on the top two most important dimensions as determined by SymTorch's `get_importance` method. Whereas for the KL model variation, we chose the two most importance message dimensions as the ones with the highest KL divergence as calculated in Equation (16).

Variable transforms To improve the efficiency of the SR, we performed the following variable transforms on the input data: $\Delta x = x_1 - x_2$, $\Delta y = y_1 - y_2$, and $r = \sqrt{\Delta x^2 + \Delta y^2 + 10^{-2}}$. We added a small constant to the distance r to match the form used in the original potential equations when generating the dataset. Thus the inputs to the SR were these transformed variables as well as m_1, m_2, q_1, q_2.

SR parameters The operators that were allowed in the SR were $+, -, \times, \texttt{inv}(\cdot)$, which had complexity of 1, as well as `exp` and `log`, which we set to have complexity of 3. The complexity of constants and input variables were set to be 1. We chose a random set of 5,000 examples from the test set for the SR. For all of the tests, we ran the SR for 7,000 iterations. The parsimony argument was set to 0.05 and the maximum size of equations permitted (measured in terms of complexity) was set to 25. Lastly, we also set constraints of `exp` and `log` to be one.

E.7 RESULTS

The final loss on the trained GNNs are shown in Table 6 and the symbolic regression results are shown in Table 3. An example Pareto front of equations is shown in Table 8 and the corresponding scores are shown in Figure 7.

Table 6: Test set mean absolute error (MAE) for different GNN model variants across four force law simulations. Lower values indicate better performance. The pruning variant (introduced by SymTorch) achieves comparable performance to the bottleneck model while automatically discovering the optimal dimension. Variance values provide scale context for the prediction errors.

Simulation	Standard	Bottleneck	L1	KL	Pruning	*Variance*
Charge	18.20	19.12	18.06	39.80	19.40	56417.66
r^{-1}	0.26	0.25	0.32	15.40	0.28	87.69
r^{-2}	24.07	25.25	21.67	57.80	23.77	98641.90
Spring	0.24	0.16	0.20	7.29	0.23	55.84

Table 7: Symbolic regression results for each message component. ✓= correct form of force law recovered; ✗= failure. * Correct form, but with a small constant added to a term (e.g., $1/(r + \text{const.})$). † Correct form, but only Δy apparent in both messages. ‡ Correct form with Δx in one message and Δy in the other. § Correct form with only Δx or Δy in at least one message.

Simulation	Message	Standard	Bottleneck	L1	KL	Pruning
Charge	1	✗	✓‡	✗	✗	✓
	2	✗	✓‡	✓§	✗	✓§
r^{-1}	1	✗	✓	✓	✗	✓
	2	✓	✓	✓	✗	✓*
r^{-2}	1	✓§	✓	✓§	✓‡	✓
	2	✗	✓	✗	✓‡	✓
Spring	1	✓†	✓	✓	✓‡	✓
	2	✓†	✓	✓§	✓‡	✓

Examples of successful reconstructions Below are some examples of successful reconstructions of the true forces:

- Spring; bottleneck

$$\texttt{msg1} = \left(\frac{1}{\texttt{r}} - 0.99950946\right) \cdot (0.8855752\Delta\texttt{y} + 1.8560125\Delta\texttt{x}) + 0.031805687$$

- r^{-2}; L1

$$\texttt{msg1} = \texttt{m}_2((\Delta\texttt{y} + 0.43513915) + \Delta\texttt{x}) \cdot \frac{1}{\texttt{r}^3}$$

- Charge; pruning

$$\texttt{msg2} = \frac{1}{(\texttt{r} + 0.036421545)\texttt{r}^2} \cdot \texttt{q}_1\texttt{q}_2 \cdot (\Delta\texttt{x} - 0.0331669) + 0.08641323$$

(shows the correct functional form except there is a small constant added to one of the $1/r$ terms and a Δy term is not present).

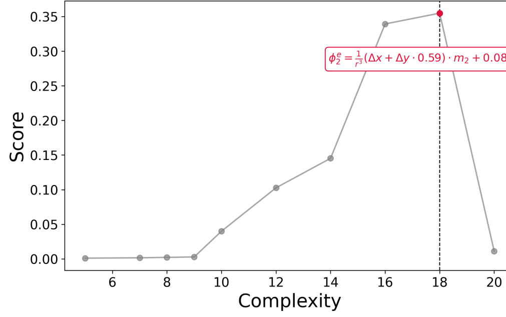

Figure 7: The score of the equations found from PySR for the pruning model trained on the r^{-2} as given in Table 8. The score is calculated from Equation (2). The best equation, highlighted in red, produces the largest drop in $\log(\text{loss})$ per unit of additional complexity.

Table 8: Pareto front from PySR results on message 2 of pruning model on r^{-2} dataset. The highlighted equation shows the 'best equation' chosen by PySR according to the metric given in Equation 2. The constants have been truncated to two decimal points.

Complexity	Equation	Loss
1	$\phi_2^e = 0.08$	0.0888
5	$\phi_2^e = (\Delta x \cdot 0.00) + 0.08$	0.0885
7	$\phi_2^e = ((\Delta x + \Delta y) \cdot 0.00) + 0.08$	0.0882
8	$\phi_2^e = ((\Delta x \cdot \text{inv}(r)) \cdot 0.00) + 0.08$	0.0880
9	$\phi_2^e = (m_2 \cdot (\Delta y + \Delta x) \cdot 0.00) + 0.08$	0.0877
10	$\phi_2^e = (\text{inv}((r + \Delta x) \cdot r) \cdot -0.00) + 0.08$	0.0843
12	$\phi_2^e = (\text{inv}(r^3) \cdot (\Delta x \cdot 0.02)) + 0.08$	0.0687
14	$\phi_2^e = ((m_2 \cdot 0.01) \cdot (\Delta x \cdot \text{inv}(r^3))) + 0.08$	0.0513
16	$\phi_2^e = ((\Delta y + \Delta x) \cdot ((m_2 \cdot \text{inv}(r^3)) \cdot -0.01)) + 0.08$	0.0260
18	$\phi_2^e = ((\text{inv}(r^3) \cdot (\Delta x + \Delta y \cdot 0.59)) \cdot m_2) + 0.08$	0.0128
20	$\phi_2^e = (\text{inv}(r^3) \cdot (((m_2 + 0.06) \cdot 0.01) \cdot (\Delta x + \Delta y \cdot 0.59))) + 0.08$	0.0125

Limitations of the framework In the reconstructions for the r^{-1} and r^{-2} force laws, the mass of the receiving node, m_1, is absent. As described in Equation (5), the edge model is intended to learn the interaction forces between particles, with the node model aggregating these messages and outputting the resulting accelerations, as shown in Equation (19). However, this setup is mathematically equivalent to having the edge model learn accelerations directly, with the node model simply outputting the aggregated accelerations, as shown in Equation (20).

Edge model learns forces:

$$\phi^e(\mathbf{v}_{r_k}, \mathbf{v}_{s_k}) \approx \text{Force on } r_k \text{ by } s_k,$$
$$\phi^v\left(\mathbf{v}_{r_k}, \sum_{j \neq r_k} \phi^e(\mathbf{v}_{r_k}, \mathbf{v}_j)\right) \approx \phi^v(\mathbf{v}_{r_k}, \text{Resultant force on } r_k) \approx \text{Acceleration of } r_k. \tag{19}$$

Edge model learns accelerations:

$$\phi^e(\mathbf{v}_{r_k}, \mathbf{v}_{s_k}) \approx \text{Acceleration on } r_k \text{ caused by interaction with } s_k,$$
$$\phi^v\left(\mathbf{v}_{r_k}, \sum_{j \neq r_k} \phi^e(\mathbf{v}_{r_k}, \mathbf{v}_j)\right) \approx \phi^v(\mathbf{v}_{r_k}, \text{Resultant acceleration of } r_k) \approx \text{Acceleration of } r_k. \tag{20}$$

SR Wall Clock Time To perform SR on the edge model it took approximately 10 minutes on an Apple M4 Max SoC (14-core CPU: 10 performance + 4 efficiency cores, 36 GB unified memory).

F EXTRACTING PDE SOLUTIONS FROM A PINN DETAILS

F.1 THEORY

PINN architecture PINNs augment standard neural network training with regularization terms encoding physical laws (PDEs, boundary conditions, and initial conditions). We trained both a PINN and regular NN on temperature data governed by the 1-D heat equation:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \tag{21}$$

where u is the temperature, $x \in (0,1)$ is a 1-D spatial coordinate, $t \in (0,1]$ is time and α is the diffusion coefficient. The BCs for this physical system are

$$u(0,t) = 0, \quad u(1,t) = 0, \tag{22}$$

and the IC is

$$u(x,0) = \sin(\pi x). \tag{23}$$

Both the regular NN and the PINN share the same architecture, but the PINN augments its loss function with additional regularization terms that encode the governing physical laws. In our case, these regularization terms, $\mathcal{L}_{\cdot}$, are

$$\mathcal{L}_{\text{PDE}} = \lambda_{\text{PDE}} \frac{1}{N} \sum_{i=1}^{N} \left(u_t(x_i,t_i) - \alpha\, u_{xx}(x_i,t_i)\right)^2, \tag{24}$$

$$\mathcal{L}_{\text{BC}} = \lambda_{\text{BC}} \frac{1}{N} \sum_{i=1}^{N} \left(u(0,t_i)^2 + u(1,t_i)^2\right), \tag{25}$$

$$\mathcal{L}_{\text{IC}} = \lambda_{\text{IC}} \frac{1}{N} \sum_{i=1}^{N} \left(u(x_i,0) - \sin(\pi x_i)\right)^2. \tag{26}$$

with regularization strengths denoted by $\lambda_{\cdot}$.

F.2 METHODS

Dataset The dataset consisted of ten data points, $\mathbf{x} \in \mathcal{X} \subset \mathbb{R}^2$, sampled from the true solution of the PDE,

$$u(\mathbf{x}) = u(x,t) = e^{-\pi^2 \alpha t} \sin(\pi x), \tag{27}$$

where we set $\alpha = 0.2$.

Model and training The NN and PINN consisted of a single MLP $\phi : \mathcal{X} \to \mathcal{Z}$, where $\mathcal{Z} \subseteq \mathbb{R}$, with three hidden layers and Tanh activations. Each hidden layer contained 32 hidden units. For the PINN, we added additional regularization terms (from Equation (24)-Equation (26) with regularization strengths $\lambda_{\text{PDE}} = 1$ and $\lambda_{\text{BD}} = \lambda_{\text{IC}} = 5$) to the loss. We also set α to be a differentiable parameter in the PINN such that, through training, the model can predict its value. Both models were trained for 30,000 optimizer steps.

Results A comparison of the trained NN, PINN and ground truth for a grid of data is shown in Figure 8.

Symbolic distillation We used SymTorch to wrap both MLPs. For the SR, the default SymTorch parameter, as from Table 3. However, we further added a constraint on the $\exp$ and $\sin$ operators to only take arguments up to complexity 3, added a parsimony coefficient of 0.01 and ran the search for 1000 iterations. The sample batch for the SR consisted of 5000 randomly sampled data points x and t in their respective allowed intervals.

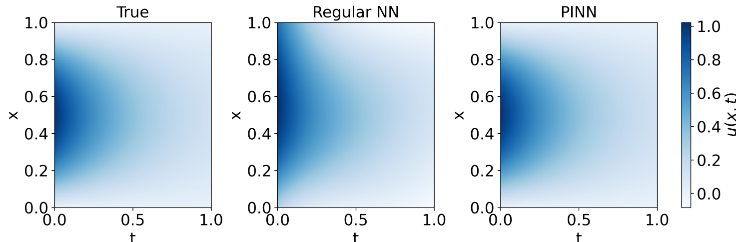

Figure 8: Comparison of regular NN and PINN predictions and the true solution of the 1D heat equation ($\alpha = 0.2$).

F.3 Results

To compare the performances of the NN and the PINN we evaluate their Mean-Square Error (MSE) losses on a 2-D space-time grid of granularity 100×100. Figure 8 shows a comparison of the NN's and the PINN's predictions on the grid and the ground truth. The MSE losses on the grid are shown in Table 9. The PINN outperformed the NN, especially at boundaries. The PINN further predicted the value of α correctly to 4 decimal points.

Table 9: Mean squared errors (MSEs) of the regular neural network and Physics-Informed Neural Network (PINN) predictions on the $u(x, t)$ space–time grid for the 1D heat equation. The PINN, trained with only 10 data points, achieves three orders of magnitude lower error than the regular NN due to its physics-informed regularization. Variance of the ground-truth solution shown for scale.

	Regular NN	PINN	*Variance*
MSE loss	7.81×10^{-3}	7.40×10^{-6}	4.82×10^{-2}

Extracting the true solution From the trained PINN, we were able to distill the correct form of the 1-D heat equation solution as given in Equation (27) with the constants correct to 2 decimal points. However, we were unable to do this for the regular NN.

SR Wall Clock Time To perform SR on the PINN it took less than 3 minutes on an Apple M4 Max SoC (14-core CPU: 10 performance + 4 efficiency cores, 36 GB unified memory).

G Interpreting LLM-Learned Operations Details

G.1 Setup

The specific prompts used to query the Llama-3.2-1B-Instruct model are shown in Table 10. When generating text, we configured the LLM to perform greedy decoding (`do_sample=False`) with a maximum generation limit of 250 new tokens. Since the LLM responses included explanatory text beyond the numeric answer (eg. step-by-step reasoning), we implemented a regex-based extraction function to parse the final numeric result from the LLM's output. We then created a wrapper function for each mathematical operation (addition, multiplication, temperature conversion, and counting) that: (1) took the numerical inputs as a NumPy array (eg. pairs of 3-digit integers for addition), (2) formatted these inputs into natural language prompts, (3) queried the LLM, (4) extracted the numeric answer, and (5) returned the result as a NumPy array. Finally, we used SymTorch's model-agnostic mode to wrap these functions and perform SR on the LLM's input-output behavior.
The SR parameters used are the default ones as in Table 3 but we added a constraint on $\sin$ and $\exp$ to only take arguments of up to complexity 1 and ran the SR for 5000 iterations.

SR Wall-Clock Time Symbolic distillation required 5-15 minutes on an Apple M4 Max SoC (14-core CPU: 10 performance + 4 efficiency cores, 36 GB unified memory) for each of these operations.

Table 10: Prompts used to query Llama-3.2-1B-Instruct on four mathematical tasks: addition, multiplication, Celsius-to-Fahrenheit conversion, and counting 1s in binary strings. All prompts request answers in $boxed$ format to enable regex-based extraction of numeric results. For the temperature conversion task, we only inputted temperatures between $-20°$C and $200°$C.

Operation	Example Prompt
Addition of two 3-digit numbers	Return only the numeric answer in the format $boxed$. What is $193 + 374 =$?
Multiplication of two 3-digit numbers	Return only the numeric answer in the format $boxed$. What is $484 * 726 =$?
Counting 1s in a 6-digit string of 1s and 0s	Return only the numeric answer in the format $boxed$. How many 1s are there in the string 000101?
Converting from Celsius to Fahrenheit	Return only the numeric answer in the format $boxed$. What is 30 degrees Celsius in Fahrenheit?