Machine Learning for Physics: from Symbolic Regression to Quantum Simulation

by

Owen Michael Dugan

Submitted to the Department of Physics
as a supplement to the requirements for the degree of

BACHELOR OF SCIENCE IN PHYSICS

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2024

Authored by:	Owen Michael Dugan Department of Physics May 10, 2024
Certified by:	Marin Soljačić Professor of Physics, Thesis Supervisor
Accepted by:	Lindley A. Winslow Associate Professor of Physics Associate Head, Department of Physics

Machine Learning for Physics: from Symbolic Regression to Quantum Simulation

by

Owen Michael Dugan

Submitted to the Department of Physics
as a supplement to the requirements for the degree of

BACHELOR OF SCIENCE IN PHYSICS

ABSTRACT

In this thesis, we explore the application of machine learning (ML) methods to problems in physics.

Because ML has revolutionized a wide range of fields, it is natural to ask whether it may be a valuable tool for physics. Physics applications present a challenge as many physics problems have a precise mathematical definition and a classical (non-ML-based) solution, making ML models less likely to outperform existing techniques.

In this paper, we focus on two general problems for which ML techniques provide an improvement as compared to existing techniques in physics: 1) fast simulation, and 2) discovering new physics. To illustrate the potential of ML to advance physics by solving these problems, we develop a physics-optimized ML model for each of the problems identified above, respectively: 1) Q-Flow, a technique for faster bosonic quantum simulation using normalizing flows to simulate a compressed representation of a quantum state, and 2) OccamNet, a framework for scientific discovery through novel algorithms for efficient and parallelizable symbolic regression. Our methods demonstrate the potential for ML as a valuable tool for physics research.

Thesis supervisor: Marin Soljačić
Title: Professor of Physics

Acknowledgments

I would like to thank:

- Professor Marin Soljačić for being an incredible PI and academic advisor. Thank you for helping me find and explore my passions in AI and physics.
- Rumen, Peter, Di, Zhuo, Donato, Charlotte, Viggo, Momchil, and all my other collaborators for making my research experience at MIT so fun.
- My friends at MIT for all the great times.
- My family for their constant support and belief in me.
- MIT for being even more amazing than I expected.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning (ML) has revolutionized a wide range of fields, including image recognition [1], natural language processing [2, 3], robotics [4], biology [5], and even areas of physics [6, 7]. This success stems from its ability to model arbitrary data [8, 9], or more generally to use gradient descent to find local optima for any differentiable objective [10]. As such, ML is often applied to tasks that lack formal definition or possess no obvious first-principles solution, such as image classification or natural language processing, where ML algorithms' ability to model arbitrary data makes them a viable option.

Given ML's success, it is natural to ask in what ways it may be a valuable tool for physics. Physicists have already used ML techniques for a variety of problems in physics, including event detection and reconstruction at experiments like the LHC [6] and DUNE [7], simulating quantum systems, and more.

However, physics applications can present a challenge for ML because most physics problems have a precise mathematical definition and a classical (non-ML-based) solution. This means that ML models must not only work but also outperform existing classical techniques in order to be useful. As such, in order to develop useful ML techniques for physics, it is important to identify areas where existing techniques fail and ML algorithms have strengths.

In this paper, we focus on two such areas of physics research in which ML techniques provide an improvement as compared to existing techniques:

1. Fast and memory-efficient simulation. Existing simulation techniques often struggle to scale to large systems [11]. For a quantum system, for example, the size of the state of the system grows exponentially with the number of components (particles, potential wells, etc.), so conventional simulators quickly run out of memory and compute as the size of the system grows [11]. Fields such as quantum computing and quantum engineering center around the study of macroscopic collections of particles, so the inability to simulate large quantum systems limits the progress in such fields [11].

2. Discovering new physics. Because of the open-ended nature of scientific discovery, and the enormous search-space of possible theories, scientific discovery is traditionally performed by hand, relying on human intuition. However, such a process can be difficult and slow. This problem naturally lends itself to ML techniques, which can assist in the high-dimensional search problem [12] and may even be infused with human-like intuition [3].

To illustrate the potential of ML to advance physics through these avenues, we develop a physics-optimized ML model for each:

1. In Chapter 2, we discuss Q-Flow, a technique for faster bosonic quantum simulation using normalizing flows to simulate a compressed representation of a quantum state [11]. Q-Flow enables simulating higher-dimensional quantum systems than is possible using standard finite-difference or finite-element solvers [11].

2. In Chapter 3, we discuss OccamNet, a framework for scientific discovery through novel algorithms for efficient and parallelizable symbolic regression [13]. By using reinforcement learning and symbolic inductive biases, OccamNet intelligently searches through the space of possible equations describing data, a step toward automated physics discovery [13].

Our methods demonstrate the potential for ML as a valuable tool for physics research.

Chapter 2

Q-Flow: Generative Modeling for Differential Equations of Open Quantum Dynamics with Normalizing Flows

Q-Flow [11] enables faster bosonic quantum simulation by using normalizing flows to simulate a compressed representation of a quantum state. This enables simulating higher-dimensional quantum systems than is possible using standard finite-difference or finite-element solvers. The below is taken from [11].

2.1 Introduction

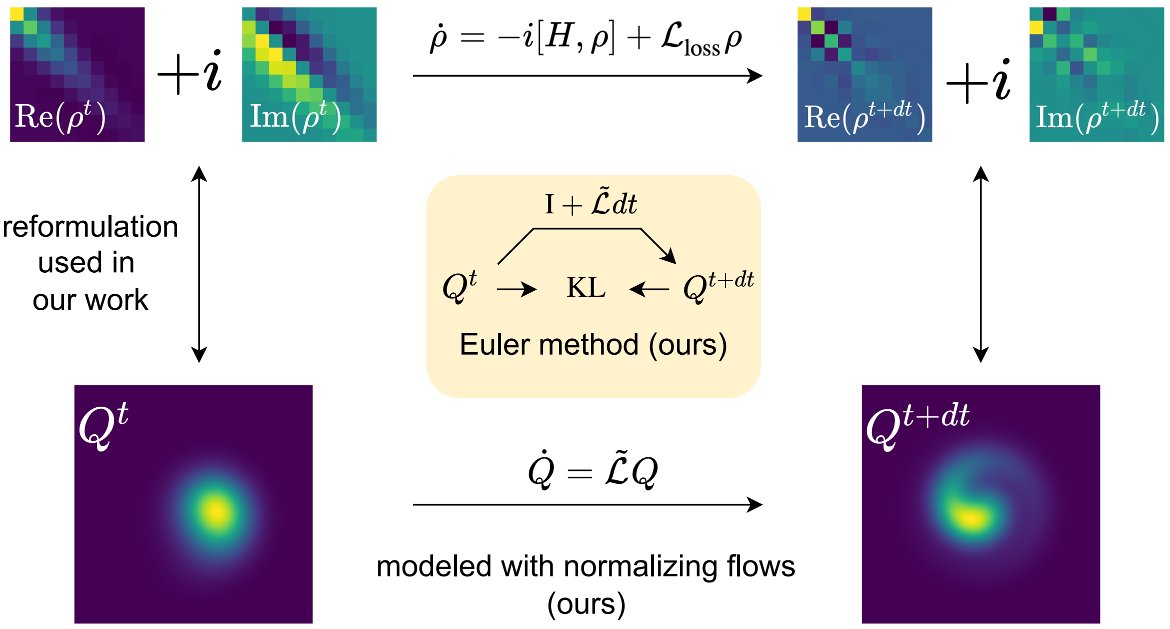
Understanding the dynamics of open quantum systems is a necessary step towards advances in fundamental physics and high-impact scientific applications such as quantum engineering and quantum computation [14, 15]. The dynamics of such systems depend on the *density matrix* ρ, which describes all the information of a quantum many-body system. The density matrix is an exponentially scaling object in the double Hilbert space whose complexity grows as N^{2k} for N local degrees of freedom and k sites. Thus, solving for ρ suffers from the curse of dimensionality.

Pioneering work on representing ρ in a compact form as a customized deep generative neural network has shown great promise in advancing the frontier of understanding high-dimensional quantum systems [16, 17, 18, 19]. However, a number of computational challenges remain when solving for ρ, which motivates the development of novel machine learning methods. Notable challenges are:

1. The density matrix ρ is *complex-valued* and has the constraint $\mathrm{tr}[\rho] = 1$. That makes it non-trivial to model with standard deep generative models, which are real-valued.

2. The partial differential equation (PDE) that governs the dynamics of ρ models complicated interactions, which hampers the application of standard machine learning PDE solvers.

3. Previous efforts on modeling ρ with neural networks are restricted to discrete spin systems, and it is unclear how to model ρ in continuous space or bosonic systems.



$\mathrm{Re}(\rho^t)$
$+i$
$\mathrm{Im}(\rho^t)$
$\dot{\rho} = -i[H,\rho] + \mathcal{L}_{\mathrm{loss}}\rho$
$\mathrm{Re}(\rho^{t+dt})$
$+i$
$\mathrm{Im}(\rho^{t+dt})$
reformulation used in our work
$\mathrm{I} + \tilde{\mathcal{L}}dt$
$Q^t \to \mathrm{KL} \leftarrow Q^{t+dt}$
Euler method (ours)
Q^t
$\dot{Q} = \tilde{\mathcal{L}}Q$
modeled with normalizing flows (ours)
Q^{t+dt}

Figure 2.1: Q-Flow. Reformulated density matrix dynamics in continuous systems to a PDE for probability distributions. Off-the-shelf normalizing flows, and our Euler-KL method are used to solve such PDEs.

The state-of-the-art literature has addressed Challenge 1 by developing customized neural architectures for ρ in spin systems with discrete degrees of freedom only [16, 17, 18, 19, 20, 21]. Challenge 2 has been attempted by exploring physics-inspired training objectives, such as Physics-informed neural networks (PINNs) [22, 23]. PINNs have shown promise in low-dimensional systems but it is not clear how they can scale to high-dimensional PDEs. Furthermore, the existing literature has not addressed Challenge 3 and missed an opportunity to establish a *direct* connection between progress in modeling open quantum dynamics and novel deep generative models for standard machine learning benchmarks. Such a connection would drive progress in both machine learning applications for open quantum dynamics and deep generative modeling.

In this paper, we address Challenges 1-3 by establishing a bridge between open quantum systems and continuous-variable generative modeling.

Firstly, we reformulate the problem by replacing the density matrix ρ with an alternative representation, the Husimi Q function Q [24], which can be practically considered as a probability distribution. Thus, we enable the use of *off-the-shelf* generative neural networks to model Q. Because variational Monte Carlo methods for quantum systems require access to both easy sampling and probability density values, we use *normalizing flows* [25, 26] as our generative model.

Secondly, we develop novel methods for training normalizing flows that obey complicated high-dimensional PDEs, which are an excellent fit for approximating Q. We propose a training method, the stochastic *Euler-KL* method, which is based on the forward discretization of the differential equation for Q and the Kullback-Lieber matching of probability distributions. Our normalizing flows approach can also be equipped with the Time-Dependent Variational

Principle (TDVP) method [27], which can be derived from the Euler method and can be thought of as an analog of the natural gradient method [28, 29].

We name our contributions *Q-Flow* (see Figure 2.1). Q-Flow is a new approach to solving open quantum systems based on off-the-shelf normalizing flows and the Euler/ TDVP methods for evolving such flows in complicated PDEs. We demonstrate that Q-Flow is scalable and efficient for simulating various open quantum systems. Our contributions can be summarized as follows:

- New generative modeling approach for open quantum dynamics with continuous degrees of freedom based on the Husimi Q function, which allows for using normalizing flows off-the-shelf.
- New methods for solving open quantum dynamics PDEs using normalizing flows with stochastic Euler-KL method and TDVP.
- Demonstration of the scalability and efficiency of our methods on simulations of dissipative harmonic oscillator and dissipative bosonic models by surpassing conventional PDE solvers and state-of-the-art machine learning PDE solvers, physics-informed neural networks (PINN).

Importantly, with Q-Flow, the difficulty in simulating quantum dynamics is no longer the dimension of the simulation but instead the complexity of the Q function and its evolution, which opens a new avenue for research.

2.2 Related Work

2.2.1 Neural Network Quantum States

Neural network quantum states are generative neural network architectures—including restricted Boltzmann machines [30], autoregressive models [31, 32, 33, 34], and determinant neural network models [35, 36, 37]—that have been adapted to represent quantum wave functions or density matrices (in the case of open quantum systems) rather than probability distributions. They are optimized using variational quantum Monte Carlo methods and have primarily been applied to model discrete spin systems [30, 31, 32] as well as tackle the continuous many-body wave function in quantum chemistry [35, 36], condensed matter [38, 39] and quantum field theories [40, 41].

In contrast with prior deep learning-based approaches that directly model the wave function or density matrix, our work focuses on the Q function representation of the quantum state—a continuous quasiprobability distribution [24] that can be modeled using an appropriate generative model, e.g. normalizing flows.

2.2.2 Partial Differential Equation (PDE) Solvers

To model the dynamics of an open quantum system using the Q function formulation, we are required to solve a high-dimensional PDE. By parameterizing the Q function using a normalizing flow, our approach can efficiently solve this PDE. For comparison, we benchmark our work against alternative PDE solvers.

Traditional PDE solvers struggle to handle high-dimensional PDEs due to the curse of dimensionality, where even storing the state of the system on a grid or mesh grows expontentially with the dimension of the problem. As a comparison, we use a simple pseudo-spectral method [42] as our traditional solver benchmark. While there are specialized methods for solving high-dimensional PDEs, they are often complex to set up and only apply to a few restricted classes of PDEs, e.g. parabolic PDEs [43].

We also benchmark against physics informed neural networks (PINNs)—a promising deep learning-based variational approach for solving PDEs [22, 23, 44]. PINNs, however, have been shown to have limitations related to the difficulty of the variational optimization problem [45] and, in their standard form, may also suffer from the curse of dimensionality.

2.3 Solving Open Quantum Dynamics with Q-Flow

In this work, we develop Q-Flow, an approach for solving open quantum dynamics based on flow-based models under the Q function partial differential equation formulation. The key contributions of our work are twofold. Firstly, we establish a general framework for solving open quantum dynamics learning through the flow-based model representation. Secondly, we develop optimization algorithms for solving high dimensional partial differential equations and apply them to PDEs for the Q function.

2.3.1 Open Quantum System

A generic Markovian open quantum system starts with the following form

$$\dot{\rho} = \mathcal{L}\rho = -i[H, \rho] + \mathcal{L}_{\text{loss}}\rho, \tag{2.1}$$

where ρ is the density matrix, H is the Hamiltonian, $\mathcal{L}_{\text{loss}}$ is the dissipative operator, and $[\cdot, \cdot]$ is the commutation operator between matrices, i.e. $[A, B] = AB - BA$.

In general, ρ is a complex-valued density matrix whose size grows exponentially with the number of particles. The density matrix ρ is a generalization of the wave function in the Schrödinger equation, which can be viewed as an ensemble of wave functions. We note that Eq. 2.1 is a complex-valued high-dimensional differential equation, which is challenging to be solved in general.

In this work, we focus on bosonic open quantum dynamics with continuous degree of freedom, which arises in a variety of contexts [46, 47]. A bosonic particle, also known as a *boson*, is one kind of fundamental particle in quantum mechanics. A boson has continuous degrees of freedom, which could be more challenging to represent compared to discrete variable systems such as spin systems. In practice, one workaround is to truncate the infinite continuous degree of freedom to some large finite degree K. Regardless of the truncation effect, bosons at M Wells still live in an exponentially-large-dimensional Hilbert space of size M^K, which is intractable to simulations in general.

2.3.2 Q Function Formulation

The Husimi Q function [48] provides an exact reformulation of Eq. 2.1 into a probabilistic differential equation:

$$\dot{Q} = \tilde{\mathcal{L}} Q \tag{2.2}$$

where Q is called the Husimi Q function and $\tilde{\mathcal{L}}$ is the corresponding operators transformed from the Hamiltonian operator H and the dissipative operator $\mathcal{L}_{\text{loss}}$.

Mathematically, the Q function of one particle is related to density matrix ρ through $Q(\alpha, \alpha^*) = \frac{1}{\pi} \langle \alpha | \rho \alpha \rangle$, where $\alpha = x + iy$ is a complex number, and $|\alpha\rangle$ is known as the coherent state (see Appendix A.1.1 for more details). $Q(\alpha, \alpha^*) \geq 0$ for any α and $\int Q = 1$, so Q can be interpreted as a probability distribution in practice.

To efficiently apply the Q function formalism, we must be able to easily convert between the ρ and Q functions and obtain $\tilde{\mathcal{L}}$. We supplement the key conversion formulas and the corresponding proofs in Appendix A.1.

2.3.3 Q-Flow representation: Flow-based Generative Models of Q function

One important feature of our work is to represent the Q function with off-the-shelf flow-based generative models. Thus, our work is distinguished from the previous works [16, 17, 18, 19] that represent the high dimensional complex-valued density matrix using customized neural networks. There are several advantages of our approach. Firstly, we do not need to work with complex-valued functions, which could be complicated by the sign structure problem [49]. Secondly, Q-Flow is natural for systems with continuous degrees of freedom. Thirdly, Q-Flow allows normalized probability modeling with exact sampling, which is important for solving high dimensional probabilistic PDEs with the stochastic Euler method.

Normalizing Flows. Normalizing flows are generative models for continuous probability distributions that provide both normalized probabilities and exact sampling—making them ideal for modeling the continuous Q function in our approach. Normalizing flows transform a simple initial density p_X (often a unit normal distribution) to a target density p_Y (i.e. the distribution that we want to model) via a sequence of invertible transformations [25, 26]. The invertible transformations are usually parameterized by an invertible neural network architecture $y = f_\theta(x)$ with $x \sim p_X$ and $y \sim p_Y$. The target probability density is then given by

$$p_Y(y) = p_X(f_\theta^{-1}(y)) \left| \frac{\partial f_\theta^{-1}(y)}{\partial y} \right|.$$

Many choices of f_θ are available, including affine coupling layers (RealNVP) [50], continuous normalizing flows (CNF) [51], and convex potential flows (CP-Flow) [52]. While RealNVP is the simplest to implement, affine coupling layers are less expressive than CNFs or CP-Flows, which are provably universal density estimators [52]. In addition, because of Equation A.1, we would like our flow to be infinitely differentiable which are satisfied by the above flow architectures.

Algorithm 1 Stochastic Euler-KL Method

```
Input: normalizing flow models for $Q_\theta^{t+dt}$ and $Q^t$, total time $T$, time step $dt$, $n_{iter}$, optimizer Adam.
Output: Optimal parameters $\boldsymbol{\theta}^*$ at time step $t + dt$
Initialization: Random $\boldsymbol{\theta}(t_0)$
for $j$ in range($T/dt$) do
    for $i = 0$ to $n_{\text{iter}}$ do
        update $\theta$ using Eq. 2.5 and optimizer Adam
    end for
    $Q^t \leftarrow Q_{\theta^*}^{t+dt}$
end for
```

Theorem 2.3.1. *For a Q function from a given density matrix ρ, there exists a universal approximation with a Q-Flow representation.*

Proof. For any given density matrix ρ, there is a corresponding Q_ρ which satisfies $Q_\rho \geq 0$ and $\int Q_\rho = 1$. Here, one can view Q_ρ practically. Since it has been shown that normalizing flows are universal approximations of probability distributions [52], there exists a Q-Flow representation Q_f such that Q_f can be arbitrarily close to Q_ρ. □

Theorem 2.3.2. *For any local observable expected value to be computed with respect to ρ, there exists a Q-Flow representation which can compute the observable efficiently.*

Proof. Consider the corresponding Q function Q_ρ of ρ. Consider a local observable in the form of $O = a^m a^{\dagger n} + a^n a^{\dagger m}$. WLOG, we can consider $O = a^m a^{\dagger n}$, the reasoning for the remaining part is analogous. The expectation is $\langle O \rangle_\rho = \text{tr}(\rho a^m a^{\dagger n})$. Eq. A.1.5 in the Appendix shows that it can be equivalently computed by $\int (q + ip)^m (q - ip)^n Q_\rho(p, q) dp dq$, which is a certain polynomial moment of the Q function. Since normalizing flows are a universal approximators, there exists a Q-Flow representation Q_f that can be arbitrarily close to Q_ρ, which implies that $\langle Q \rangle_f$ can be arbitrarily close to $\langle Q \rangle_\rho$. Even though computing $\langle Q \rangle_f = \sum_{p,q \sim Q_f} (q + ip)^m (q - ip)^n$ has stochastic fluctuations, the exact sampling nature of the flow-based model can suppress the statistical error, which will decay with the increasing sample size N_s as $\frac{1}{\sqrt{N_s}}$ due to the Central Limit Theorem. □

2.3.4 Q-Flow Optimization: Stochastic Euler-KL Method

In the previous section, we discuss the representation of the Q function with flow-based models. To solve the real-time dynamics given by Eq. 2.2, we further develop the high dimensional stochastic Euler-KL method.

The algorithm represents the Q function at time t with a flow-based model and iteratively updates the representation at the next time $t + dt$ based on the Euler method. Concretely, it requires two copies of flow-based models for Q^{t+dt} and Q^t. Based on the first-order Euler method with time step dt, Eq. 2.2 yields

$$Q^{t+dt} = Q^t + \tilde{\mathcal{L}} Q^t \, dt = (\mathrm{I} + \tilde{\mathcal{L}} \, dt) Q^t \equiv Q^t_{\mathcal{L}}. \tag{2.3}$$

Notice that Q^{t+dt} represents the Q function that we obtain in the next time step. At each learning step, we fix Q^t and optimize the parameters θ in Q^{t+dt} to match the above relation. Hence, we also denote Q^{t+dt} by Q_θ^{t+dt}. It facilities us to define the following loss function through KL divergence.

$$KL(Q_\theta^{t+dt}||Q_\mathcal{L}^t) = \int Q_\theta^{t+dt} \ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t} \tag{2.4}$$

The gradient of Eq. 2.4 can be derived with a control variance technique as follows (see Appendix for a derivation):

$$\frac{1}{N} \sum_{x \sim Q_\theta^{t+dt}} \left[\ln \frac{Q_\theta^{t+dt}(x)}{Q_\mathcal{L}^t(x)} - b \right] \nabla_\theta \ln Q_\theta^{t+dt}(x) \tag{2.5}$$

where $b = \frac{1}{N} \sum_{x \sim Q_\theta^{t+dt}} \ln \frac{Q_\theta^{t+dt}(x)}{Q_\mathcal{L}^t(x)}$ is the baseline for control variance.

The stochastic Euler-KL method is summarized in Algorithm. 1.

Theorem 2.3.3. *The global error $\epsilon(t_n)$ of the n-step stochastic Euler method is bounded by $|\epsilon_E(t_n)| + |\epsilon_{NN}(t_n)|$, where $\epsilon_E(t_n)$ is the global error of the exact Euler method and $\epsilon_{NN}(t_n) = -P^{-1} \sum_{i=0}^n P^i r^{n+1-i}$ with $P = \mathrm{I} + \tilde{\mathcal{L}} dt$ and r^i being the i-th step stochastic Euler optimization error with neural network representation of the Q-Flow.*

Proof. $\epsilon(t_n) = Q^{t_n} - Q_{NN}^{t_n} = (Q^{t_n} - Q_E^{t_n}) + (Q_E^{t_n} - Q_{NN}^{t_n}) \equiv \epsilon_E(t_n) + \epsilon_{NN}(t_n)$, where $Q_E^{t_n}$ and $Q_{NN}^{t_n}$ are the Q function from the exact Euler method and the neural network Q-Flow at time step t_n. By the triangular inequality, $|\epsilon(t_n)| \leq |\epsilon_E(t_n)| + |\epsilon_{NN}(t_n)|$. Since the Euler method is a first order method, it has global error with order $O(dt)$ where dt the time step.

Denote the optimization error of Eq. 2.4 at time step t_{n+1} as r^{n+1}, such that $Q_{NN}^{t_{n+1}} - PQ_{NN}^{t_n} = r^{n+1}$. It follows that $Q_E^{t_{n+1}} - \epsilon_{NN}(t_{n+1}) - P(Q_E^{t_n} - \epsilon_{NN}(t_n)) = r^{n+1}$, which implies that $\epsilon_{NN}(t_{n+1}) = P\epsilon_{NN}(t_n) - r^{n+1}$ due to the cancellation of $Q_E^{t_{n+1}} - PQ_E^{t_n}$ from the exact Euler method. By induction, we have $\epsilon_{NN}(t_n) = -P^{-1} \sum_{i=0}^n P^i r^{n+1-i}$. □

Time Dependent Variational Principle (TDVP). Instead of taking gradient with respect to the KL divergence as Eq. 2.5 shows, [53] demonstrate that the minimization of Eq. 2.4 is equivalent to the time dependent variational principle, which provides a nonlinear differential equation on the parameter space θ as $S_{kk'} \dot{\theta}_{k'} = F_k$, where $S_{kk'} = \mathbb{E}[(\partial_{\theta_k} \ln Q)(\partial_{\theta'_k} \ln Q)]$ is the Fisher information matrix, and $F_k = \mathbb{E}[(\partial_{\theta_k} \ln Q)(\partial_t \ln Q)]$.

[53] apply TDVP only to solving classical PDEs. Under our Q-Flow approach, we can also apply TDVP to simulate open quantum dynamics.

Complexity Analysis. Even though the stochastic Euler-KL method and the TDVP method are equivalent mathematically, they share different algorithmic complexity. TDVP requires to solve the nonlinear different equation, which needs to invert the Fisher information matrix $S_{kk'}$. Besides potential instability, this procedure has complexity scaling as $O(N^3)$ for explicit inversion or $O(N^2)$ at least with the conjugate gradient approach, where N is the number of parameters. It may limit its application for parameters beyond the orders of ten thousands. Meanwhile, the stochastic Euler method only requires first order optimization based on Eq. 2.5, the main cost of which comes from the number of optimization steps in each dt.

2.3.5 Q-Flow Initialization: Initial State Pretraining

Using a Q-Flow to simulate a quantum system requires initializing the flow to the correct starting Q function. For some simple initial states, we find that it is sufficient to simply make the initial state the prior for the flow and initialize the flow to the identity. However, we find that using more complex initial distributions as priors to a flow tends to hamper their ability to model a system's evolution. In these cases, we instead use the standard Gaussian prior, but we use a two-step process to pretrain the flow to match the initial distribution Q_{init}.

First, we sample from the desired initial distribution using the Metropolis-Hastings Monte Carlo method and update the flow parameters to minimize the negative log likelihood $-\sum_{x\sim Q_{\text{init}}} \ln Q_\theta(x)$. This ensures that the model has some overlap with Q_{init} which helps the next step's training algorithm converge more quickly.

Second, we sample from the flow and update the flow parameters to minimize the KL Loss, $KL(Q_{\text{init}}||Q_\theta)$. We compute the gradient according to

$$\nabla_\theta KL \approx -\frac{1}{N}\sum_{x\sim Q_\theta} \frac{Q_{\text{init}}(x)}{Q_\theta(x)} \nabla_\theta \ln Q_\theta(x). \tag{2.6}$$

2.4 Experiments

For our experiments, we focus on two types of open quantum systems: dissipative harmonic oscillators and dissipative bosonic systems. We test on dissipative harmonic oscillators because they have an analytic solution, which makes them useful for benchmarking high-dimensional PDE solvers beyond the limits of conventional solvers. We then test on dissipative bosonic systems because they are commonly studied and of practical use in physics.

In these experiments, we compare Euler and TDVP methods to PINNs and pseudo-spectral (PS) solvers. Although we do not develop the TDVP method, we propose a method to apply it to open bosonic quantum systems. As such, we sometimes describe the Euler and TDVP methods as "our methods."

For our experiments, we use Affine Coupling Flows and Convex-Potential Flows for the Euler and TDVP methods. Affine Coupling Flows are fast but less expressive, so we use them for the dissipative harmonic oscillator experiments. On the other hand, Convex Potential Flows are slow but more expressive, so we use them for problems involving more complex Q functions.

To run our experiments, we use the Jax library [54] for Euler and TDVP methods. We make use of the jax-flows library. To implement the TDVP method, we make use of the NetKet library [55, 56] and its Stochastic Reconfiguration [57, 58] feature, which is mathematically equivalent to TDVP. For distributed training, NetKet uses the mpi4jax package [59]. For PINNs, we use the PINA library which is built on top of PyTorch [60]. Finally, for our pseudo-spectral methods we use Julia [61].

Table 2.1: The loss $L_1[Q_{\text{sim}}, Q_{\text{exact}}]$ for each simulation method over time (in a unit inversely proportional to the unit of γ_j). For each row, we mark the best result in bold. PS data for 20-Well does not exist because PS does not scale to high-dimensional problems.

Time	Q-Flow(Euler) (ours)	Q-Flow(TDVP) (ours)	PINN	PS	FD
1-Well					
3	$2.08 \cdot 10^{-3}$	$5.11 \cdot 10^{-3}$	$1.79 \cdot 10^{-1}$	$\mathbf{3.47 \cdot 10^{-4}}$	$8.90 \cdot 10^{-4}$
6	$5.10 \cdot 10^{-4}$	$1.17 \cdot 10^{-3}$	$1.84 \cdot 10^{-1}$	$\mathbf{3.47 \cdot 10^{-4}}$	$9.01 \cdot 10^{-4}$
9	$\mathbf{1.01 \cdot 10^{-4}}$	$2.16 \cdot 10^{-4}$	$1.91 \cdot 10^{-1}$	$3.47 \cdot 10^{-4}$	$9.01 \cdot 10^{-4}$
12	$\mathbf{1.68 \cdot 10^{-5}}$	$3.58 \cdot 10^{-5}$	$1.91 \cdot 10^{-1}$	$3.47 \cdot 10^{-4}$	$9.01 \cdot 10^{-4}$
15	$1.58 \cdot 10^{-5}$	$\mathbf{5.55 \cdot 10^{-6}}$	$1.98 \cdot 10^{-1}$	$3.47 \cdot 10^{-4}$	$9.01 \cdot 10^{-4}$
2-Well					
3	$\mathbf{3.91 \cdot 10^{-3}}$	$1.23 \cdot 10^{-2}$	$2.10 \cdot 10^{-1}$	$1.83 \cdot 10^{-1}$	$6.12 \cdot 10^{-2}$
6	$\mathbf{1.91 \cdot 10^{-3}}$	$4.66 \cdot 10^{-3}$	1.00	$1.82 \cdot 10^{-1}$	$6.09 \cdot 10^{-2}$
9	$\mathbf{7.59 \cdot 10^{-4}}$	$1.77 \cdot 10^{-3}$	1.00	$1.81 \cdot 10^{-1}$	$6.09 \cdot 10^{-2}$
12	$\mathbf{2.92 \cdot 10^{-4}}$	$6.21 \cdot 10^{-4}$	1.00	$1.81 \cdot 10^{-1}$	$6.09 \cdot 10^{-2}$
15	$\mathbf{1.47 \cdot 10^{-4}}$	$2.05 \cdot 10^{-4}$	1.00	$1.81 \cdot 10^{-1}$	$6.09 \cdot 10^{-2}$
20-Well					
3	$\mathbf{9.94 \cdot 10^{-2}}$	$1.08 \cdot 10^{-1}$	$2.17 \cdot 10^{31}$	-	-
6	$\mathbf{3.29 \cdot 10^{-2}}$	$4.10 \cdot 10^{-2}$	$2.38 \cdot 10^{30}$	-	-
9	$\mathbf{2.02 \cdot 10^{-2}}$	$2.44 \cdot 10^{-2}$	$1.34 \cdot 10^{29}$	-	-
12	$\mathbf{1.46 \cdot 10^{-2}}$	$1.68 \cdot 10^{-2}$	$1.46 \cdot 10^{28}$	-	-
15	$\mathbf{1.07 \cdot 10^{-2}}$	$1.23 \cdot 10^{-2}$	$7.07 \cdot 10^{26}$	-	-

2.4.1 Dissipative Harmonic Oscillator

Experimental Setup. The multi-well dissipative harmonic oscillator evolves according to Eq. 2.1 with Hamiltonian $H = \sum_j \omega_j a_j^\dagger a_j$ [48] and loss term

$$\begin{aligned}\mathcal{L}_{\text{loss}}\rho = &\sum_j \gamma_j \left[\frac{1}{2}(2a_j\rho a_j^\dagger - a_j^\dagger a_j \rho - \rho a_j^\dagger a_j)\right.\\ &\left. + \bar{n}_j(a_j\rho a_j^\dagger + a_j^\dagger \rho a_j - a_j^\dagger a_j \rho - \rho a_j a_j^\dagger)\right].\end{aligned} \tag{2.7}$$

Here, j labels what we will call *Wells.*

To convert to the Q function formalism, we first define notation: For a Q function $Q(\alpha_1, \ldots, \alpha_n)$, we define q and p such that $\alpha_j = q_j + ip_j$ and set $x = (\vec{q}, \vec{p})$. We use the notations $Q(\alpha_1, \ldots, \alpha_n)$, $Q(\vec{q}, \vec{p})$, and $Q(x)$ interchangeably.

Now, converting to the Q function formalism with real inputs gives [24]

$$\begin{aligned}\dot{Q} =& \sum_j \left[\gamma_j + \frac{1}{4}\gamma_j(\bar{n}_j+1)\left(\frac{\partial^2}{\partial q_j^2} + \frac{\partial^2}{\partial p_j^2}\right)\right. \\ &\left. + \left(\frac{\gamma_j}{2}q_j - \omega_j p_j\right)\frac{\partial}{\partial q_j} + \left(\frac{\gamma_j}{2}p_j + \omega_j q_j\right)\frac{\partial}{\partial p_j}\right]Q.\end{aligned}$$

We test the simulation methods on three problems of increasing dimensionality: a 1-Well system, a 2-Well system, and a 20-Well system. For each system, we use a coherent state initial condition, which corresponds to a Gaussian with variance 1/2. We center the Gaussian at $(-1,\ldots,-1)$. As time passes this Gaussian spirals toward the origin and changes its standard deviation. To make the simulation more challenging, for every Well j we uniformly sample the system's parameters $\bar{n}_j \in [3,7)$, $\gamma_j \in [0.5,1.5)$, and $\omega_j \in [0.5,1.5)$.

Metrics. To evaluate a simulation method's performance, we compute the L_1 Loss between each simulation and the exact distribution:

$$\begin{aligned}L_1[Q_{\text{sim}}, Q_{\text{exact}}] \equiv& \int \mathrm{d}^d x\, |Q_{\text{sim}}(x) - Q_{\text{exact}}(x)| \\ \approx& \frac{1}{N}\sum_{x\sim Q_{\text{exact}}}\left|\frac{Q_{\text{sim}}(x)}{Q_{\text{exact}}(x)} - 1\right|.\end{aligned} \tag{2.8}$$

Although the L_1 Loss is a useful metric, it is also illustrative to examine observables of the system. One observable is the centroid, $\mathbb{E}[\vec{x}] \approx \frac{1}{N}\sum_{x\sim Q_{\text{sim}}}\vec{x}$. With more Wells, we cannot easily plot the centroid trajectory, so instead we compute the centroid's distance from the origin, $\|\mathbb{E}[\vec{x}]\|$.

Additionally, we compute a second observable, $\mathbb{E}[(\tilde{\mathcal{L}}Q/Q)^2]$, which we refer to as the *Liouvillian loss*. The Liouvillian loss measures the magnitude of the dynamics relative to the distribution, an indicator of how perturbed the system is from equilibrium.

For the Euler and TDVP methods, we sample directly from the flow to compute expected values. For PINNs, we use Markov chain Monte Carlo (MCMC) to obtain samples. For pseudo-spectral results we compute expected values by summing over the grid and scaling by Q.

Results and Discussion. Table 2.1 shows the L_1 Loss between each simulation and the exact distribution for a number of simulation times. Although we do not include error bounds in the table for ease of viewing, the error is always at least an order of magnitude smaller than the value in question. The one exception is for the Pseudo-spectral method results, where because of our grid integration method, we do not compute error bounds. We also exclude the pseudo-spectral method from the 20-Well system because a grid size of only 10 would require at least 10^{40} values to be stored.

Both the Euler and TDVP methods have extremely low L_1 Loss. In fact, both methods perform better than the Pseudo-spectral method in the 2-Well case and in the later times of the 1-Well case. While increasing the number of Wells, we find that the Euler and TDVP methods continue to perform well while PINNs and pseudo-spectral methods struggle increasingly. Pseudo-spectral methods cannot simulate the 20-Well system due to the curse of dimensionality, and while PINNs can in principle simulate the system, in practice they

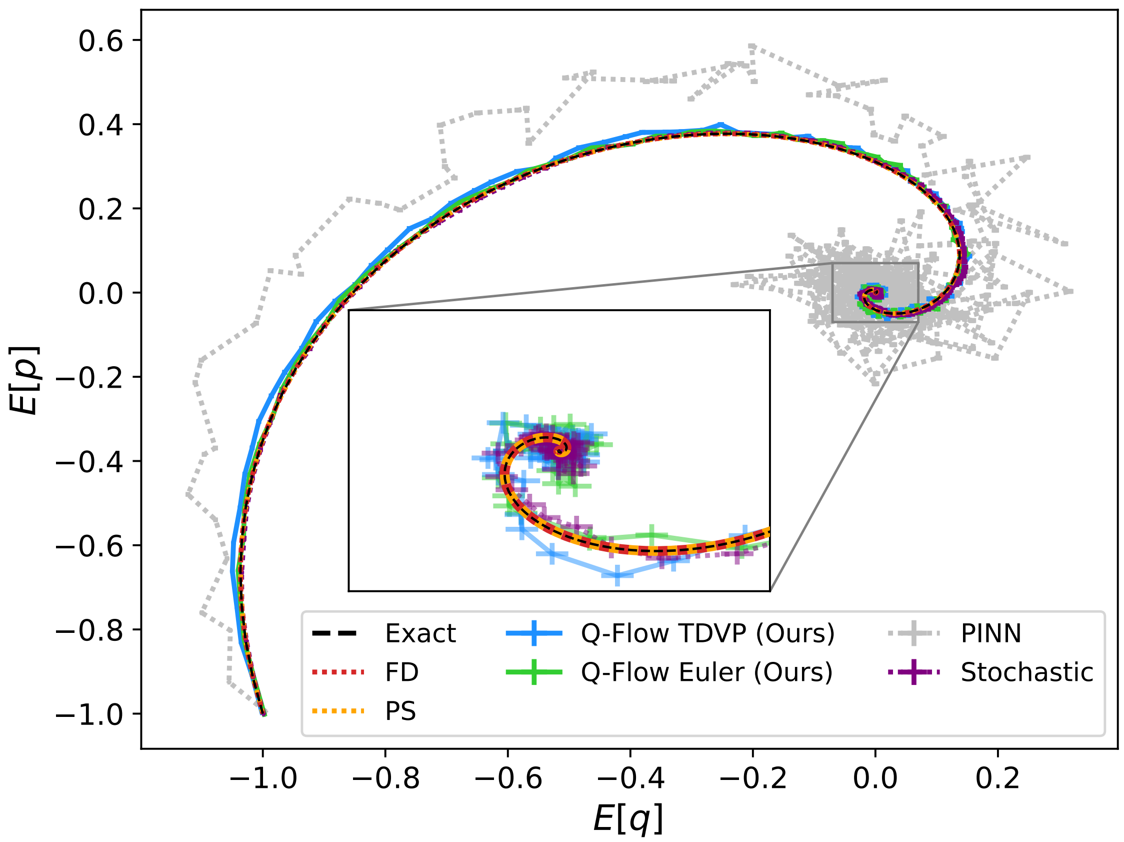


Figure 2.2: The trajectory of the centroids of the simulated distributions. The PINN baseline is excluded from the inset. Error bars are included for all except for the Exact and PS methods.

perform extremely poorly. Indeed, the poor performance of PINN is also related to the curse of dimensionality in sampling and optimization. This is because to train PINN, the standard practice is to randomly sample points or use the grid points from the problem domain, the complexity of which grows exponentially with the dimensionality. As a consequence, the loss and the gradients from finite samples are not accurate enough for PINN. In addition, PINN tries to learn the dynamics of multiple time steps simultaneously, which also increases the challenges to its representational ability. On the other hand, both the Euler and TDVP methods still consistently report low fidelities. Finally, note that the Euler method has a consistently lower L_1 loss than the TDVP method.

Figure 2.2 shows the trajectory of each simulation method's centroid for the 1-Well case. Once again, the Euler and TDVP methods both closely match the exact evolution and the Euler method performs slightly better in general. On the other hand, the PINN solution exhibits consistently biased and rapidly fluctuating estimates of the centroid. As expected, the Pseudo-spectral method closely tracks the exact trajectory. However, we note that although our methods appear to match the exact results less accurately, the large error bars in the cutout demonstrate that this is in large part due to sampling error. We could have computed the centroid for the Euler and TDVP methods using grid integration as with the Pseudospectral method, but we instead choose to use sampling because this better

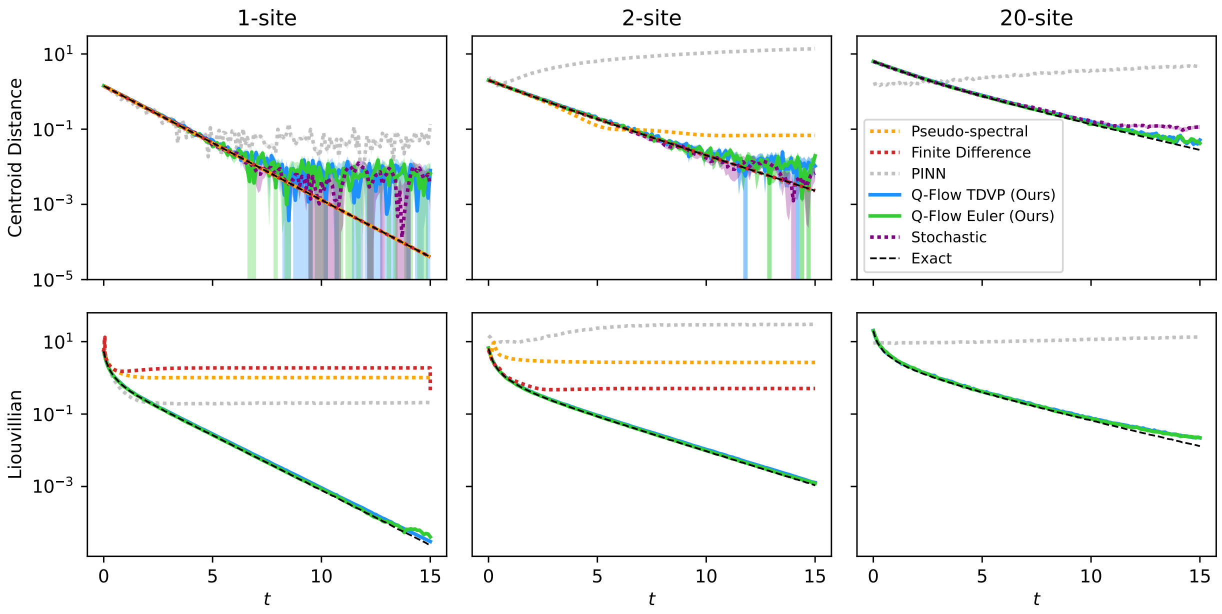


Figure 2.3: The simulated evolution of two observables for 1-Well, 2-Well, and 20-Well Fokker-Planck systems. Error bars are included for all except for the Exact and Pseudo-spectral results. The unit of time t is inversely proportional to the unit of γ_j.

generalizes to higher dimensions.

Figure 2.3 shows the evolution of the centroid distance and the Liouvillian Loss for all three problems. Euler and TDVP closely match the exact evolution of the two observables. Athough the two methods' estimates of the centroid distance begin to diverge from the exact centroid distance at around time 10, once again the large error bars demonstrate that this is due to error in the sampling estimate. Although the PINN centroid distance also begins to diverge from the desired value, the small error bars for this estimate suggest that the deviation does not come from sampling error.

In terms of the Liouvillian loss, both the Euler method and TDVP decrease consistently. At around time 15 the Euler Liouvillian loss jumps slightly, but this can most likely be improved by increasing the number of training steps per time step for the Euler method and by decreasing the learning rate. Further, this occurs at a Liouvillian loss of less than 10^{-7}, so the simulation is still likely precise enough for most applications. In practice, we find that decreasing the step size improves both the Euler and TDVP methods' performance. Interestingly, the Pseudo-spectral method provides a very poor estimate of the Liouvillian loss. We suspect that this is due to error in numerical derivatives.

Finally, note that unlike the other methods, our methods continue to correctly simulate the system for large numbers of Wells. It is only toward the end of time evolution in the 20-Well case that our methods begin to show some deviation from the exact observables. Again, this can likely be reduced by decreasing the step size and taking more samples.

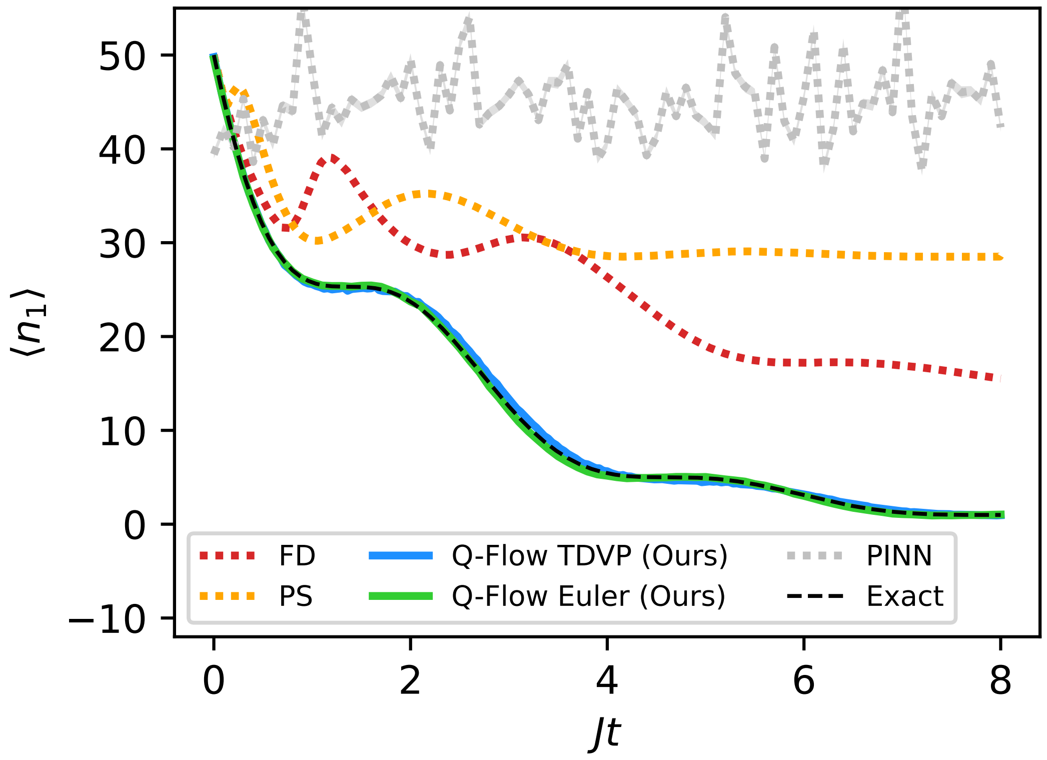


Figure 2.4: Dynamics of $\langle n_1 \rangle$ in a 2-Well dissipative bosonic model.

2.4.2 Dissipative Bosonic Model

Experimental Setup. The dissipative bosonic model is a frequently studied open quantum system [62, 44]. We test our methods on this model because it has a more complex evolution equation and because it has greater real-world applicability.

The dissipative bosonic model we use has [62] $H = -J \sum_j \left(a_{j+1}^\dagger a_j + a_j^\dagger a_{j+1} \right)$ and

$$\mathcal{L}_{\text{loss}}\rho = -\frac{1}{2} \sum_j \gamma_j \left(n_j \rho + \rho n_j - 2 a_j \rho a_j^\dagger \right) \tag{2.9}$$

where $n_j = a_j^\dagger a_j$ and j enumerates the Wells. Converting to the Q function formalism gives that $\tilde{\mathcal{L}}$ is

$$\sum_j \gamma_j \left(\frac{1}{4} \left(\frac{\partial^2}{\partial q_j^2} + \frac{\partial^2}{\partial p_j^2} \right) + \frac{1}{2} \left(q_j \frac{\partial}{\partial q_j} + p_j \frac{\partial}{\partial p_j} + 1 \right) \right)$$
$$+ J \sum_j \left(p_{j+1} \frac{\partial}{\partial q_j} - q_{j+1} \frac{\partial}{\partial p_j} + p_j \frac{\partial}{\partial q_{j+1}} - q_j \frac{\partial}{\partial p_{j+1}} \right).$$

Following Figure 3 of [63], we consider a 2-Well system with $J = 1$, $\gamma_1 = 1$, and $\gamma_2 = 0$. We simulate the evolution of an antisymmetric Bose-Einstein Condensate (BEC) with 50 particles in each Well, which has a Q function $Q(q_1, p_1, q_2, p_2)$ given by

$$Q = \frac{\left[(q_1 - q_2)^2 + (p_1 - p_2)^2\right]^{100}}{\pi^2 \cdot 2^{100} \cdot 100!} e^{-(q_1^2 + p_1^2 + q_2^2 + p_2^2)}.$$

Because of the complex multimodal initial distribution, we use the Convex Potential Flow for these experiments. We pretrain the flow as described in Section 2.3.5.

Metric. For this system, we compute the observable $\langle n_1 \rangle \approx \frac{1}{N} \sum_{x \sim Q_{\text{sim}}} (q_1^2 + p_1^2 - 1)$ because it's exact evolution is given in [63].

Results and Discussion. We show the simulated evolution of $\langle n_1 \rangle$ in Figure 2.4. Both the Euler and TDVP methods closely match the exact evolution, demonstrating that our methods' can handle multi-Well interactions correctly. In particular, the J term is responsible for the oscillations shown because it causes the two Wells to exchange particles.

2.5 Conclusion

In this work, we made a important contribution to the problem of simulating open quantum systems. We used a reformulation of the density matrix to the Husimi Q function, which allowed us to study open quantum systems as an evolution of a probability distribution under dynamics, described by a partial differential equation that we derive for each system. This allowed us to establish a direct connection between simulating continuous or bosonic open quantum systems and the rich literature on generative models in standard machine learning. With off-the-shelf normalizing flows, Affine Coupling Flows and Convex Potential Flows, and a new efficient method for solving high-dimensional PDEs, Euler-KL, we established Q-Flow, a new and efficient approach to simulation of open quantum systems.

We compared Q-Flow to the state-of-the-art numerical and deep learning approaches on two important systems to the field, the dissipative harmonic oscillator and dissipative bosonic models. We established superior performance across the board, especially when the dimensionality of the system increases rapidly in an exponential manner.

We believe the significance of our results is twofold. On one hand, Q-Flow accurate simulation of open quantum systems can be further developed to aid progress in fundamental physics and engineering applications, such as superconductors and quantum computers. On the other hand, through our reformulation from evolving the density matrix to evolving the Q function, we shifted the modeling challenges from the curse of dimensionality to the accurate evolution of a high-dimensional deep generative model. Q-Flow can aid progress in evolving probability distributions under PDE dynamics and inspire future work on deep generative models.

Chapter 3

OccamNet: A Fast Neural Model for Symbolic Regression at Scale

OccamNet [13] is framework for scientific discovery through novel algorithms for efficient and parallelizable symbolic regression. By using reinforcement learning and symbolic inductive biases, OccamNet intelligently searches through the space of possible equations describing data, a step toward automated physics discovery. The below is taken from [13]. In [64], we extend the work below by applying it to scientific discovery in social science.

3.1 Introduction

Deep learning has revolutionized a variety of complex tasks, ranging from language modeling to computer vision [65]. Key to this success is designing a large search space in which many local minima sufficiently approximate given data [66]. This requires large, complex models, which often conflicts with the goals of sparsity and interpretability, making neural nets not optimally suited for a myriad of physical and computational problems with compact and interpretable underlying mathematical structures [67]. Neural networks also might not preserve desired physical properties (e.g., time invariance) and are typically unable to generalize much beyond observed data.

In contrast, Evolutionary Algorithms (EAs), in particular genetic programming, can find interpretable, compact models that explain observed data [68, 69, 70]. EAs have been employed as an alternative to gradient descent for optimizing neural networks in what is known as *neuroevolution* [71, 72, 73]. Recently, evolutionary strategies that model a probability distribution over parameters, updating this distribution according to their own best samples (i.e., selecting the fittest), were found advantageous for optimization on high-dimensional spaces, including neural networks' hyperparameters [74, 75].

A number of evolution-inspired, probability-based models have been explored for Symbolic Regression [76]. Along these lines, [77] explore deep symbolic regression by using an RNN to define a probability distribution over a space of expressions and sample from it using autoregressive expression generation. More recently, [78] have pretrained Transformer models that receive input-output pairs as input and return functional forms that could fit the data. In the related field of program synthesis, probabilistic program induction using domain-specific languages [79, 80, 81] has proven successful. [82] first train a machine learning model to predict a DSL based on input-output pairs and then use methods from satisfiability modulo theory [83] to search the space of programs built using the predicted DSL.

One approach to symbolic regression which can integrate well with deep learning is the Neural Arithmetic Logic Unit (NALU) and related models [84, 85], which provide neural inductive bias for arithmetic in neural networks by shaping a neural network towards a gating interpretation of the linear layers. Neural Turing Machines [86, 87] and their stable versions [88] can also discover interpretable programs, simulated by neural networks connected to external memory, via observations of input-output pairs. Another option is Equation Learner (EQL) Networks [89, 90, 91], which identify symbolic fits to data by training a neural network with symbolic activation functions, such as multiplication or trigonometric functions. However, these methods require strong regularization to be interpretable. NALUs and to a lesser extent EQL Networks can also only use a restricted set of differentiable primitive functions, and Neural Turing Machines do not include the concept of a "primitive." Additionally, these methods often converge to local minima and often converge to uninterpretable models unless they are carefully regularized for sparsity.

In this paper, we consider a mixed approach of connectionist and sample-based optimization for symbolic regression. We propose a neural network architecture, OccamNet, which preserves key advantages of EQL networks and other neural-integrable symbolic regression frameworks while addressing many of these architectures' limitations. Inspired by neuroevolution, our architecture uses a neural network to model a probability distribution over functions. We optimize the model by sampling to compute a reinforcement-learning loss, tunable for different tasks, based on the training method presented in Risk-Seeking Policy Gradients [77]. Our method handles non-differentiable and implicit functions, converges to sparse, interpretable symbolic expressions, and can work across a wide range of symbolic regression problems. Further, OccamNet consistently outperforms other symbolic regression algorithms in testing on real-world regression datasets. We also introduce a number of strategies to induce compactness and simplicity a la Occam's Razor.

The main goal of this study is not to replace existing symbolic regression methods, but rather to create a novel hybrid approach that combines the strengths of neural networks and evolutionary algorithms. Our proposed OccamNet method consistently achieves state-of-the-art performance across a wide range of tasks, including a diverse range of synthetic functions, simple programs, raw data classification, and real-world tabular tasks. Additionally, we show how to connect OccamNet to state-of-the-art pretrained vision models, such as ResNets [92]. OccamNet has also shown promise in discovering quantitative and formal laws in social sciences, indicating its potential to aid scientific research [93]. By striking a delicate balance between expressiveness and interpretability, OccamNet presents a versatile and powerful solution for symbolic regression challenges.

3.2 Model Architecture

In Figure 3.1 we sketch the OccamNet architecture and the method for training it, before following with a more detailed description. We can view OccamNet as a fully-connected

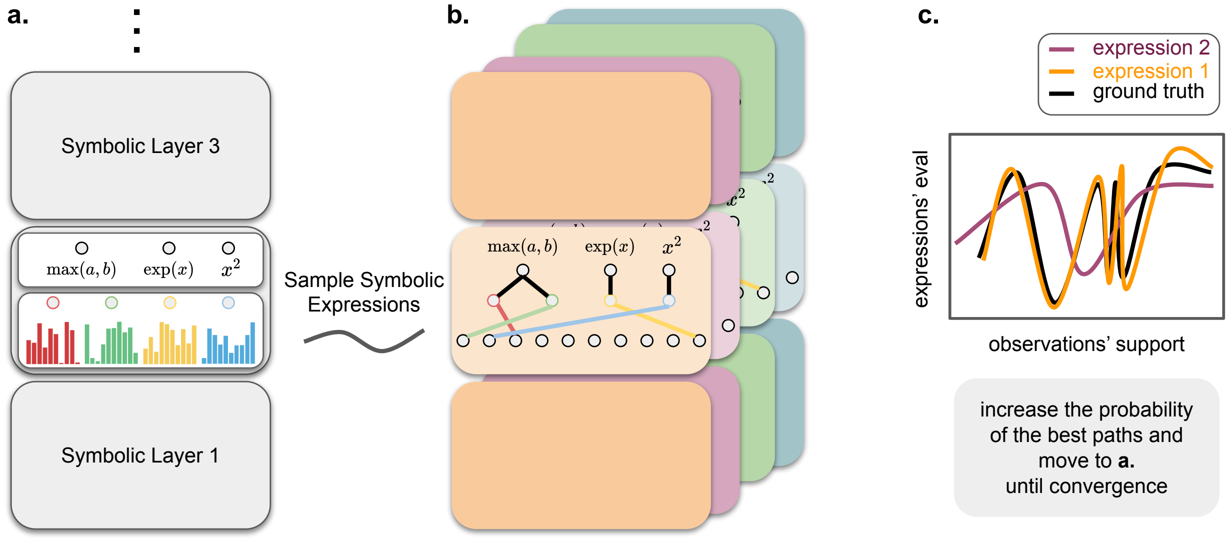


Figure 3.1: OccamNet architecture and training. **a.** OccamNet is a stack of "symbolic layers" each described by a collection of learned distributions (over the neurons from the previous layer) for each neuron within the layer, as well as non-linearities that are collections of symbolic expressions. **b.** By sampling from each distribution independently, we are able to sample paths from OccamNet that represent symbolic expressions, ready for evaluation. **c.** We evaluate each expression by feeding the observations' support data and comparing the outputs with the ground truth. The probability of the best paths is increased and the process is repeated until convergence.

feed-forward network (a stack of fully connected linear layers with non-linearities) with two key unique features. First, the parameters of the linear layer are substituted with a learned probability distribution associated with the neurons from the preceding layer for each neuron within the layer. Second, the non-linearities form a collection of symbolic expressions. Thus we obtain a collection of "symbolic layers" that form OccamNet (Figure 1a). Figure 1b shows a variety of symbolic expressions, representing paths within OccamNet from sampling each probability distribution independently. Figure 1c shows OccamNet's training objective, which increases the probability of the paths that are closest to the ground truth. Below we formalize OccamNet in detail.

3.2.1 Layer structure

A dataset $\mathcal{D} = \{(\vec{x}_p, \vec{y}_p)\}_{p=1}^{|\mathcal{D}|}$ consists of pairs of inputs $\vec{x}_p$ and targets $\vec{y}_p = \vec{f}^*(\vec{x}_p) = [f^*_{(0)}(\vec{x}_p), \ldots, f^*_{(v-1)}(\vec{x}_p)]^\top$. Our goal is to compose either $f^*_{(i)}(\cdot)$ or an approximation of $f^*_{(i)}(\cdot)$ using a predefined collection of N primitive functions $\boldsymbol{\Phi} = \{\phi_i(\cdot, \ldots, \cdot)\}_{i=1}^N$. Note that primitives can be repeated, their arity (number of arguments) is not restricted to one, and they may operate over different domains. The concept of a set of primitives Φ is similar to that of DSL, *domain-specific languages* [94].

To solve this problem, we follow a similar approach as in EQL networks [90, 89, 91], in which the primitives act as activation functions on the nodes of a neural network. Specifically, each hidden layer consists of an *arguments* sublayer and an *images* sublayer, as shown in Figure 3.2a. We use this notation because the arguments sublayer holds the inputs, or arguments, to the activation functions and the images sublayer holds the outputs, or images, of the activation functions. The primitives are stacked in the images sublayer and act as activation functions for their respective nodes. Each primitive takes in nodes from the arguments sublayer. Additionally, we use skip connections similar to those in DenseNet [95] and ResNet [92], concatenating image states with those of subsequent layers.

Next, we introduce a probabilistic modification of the network: instead of computing the inputs to the arguments sublayers using dense feed-forward layers, we compute them probabilistically and sample through the network. This enables many key advantages: it enforces sparsity and interpretability without requiring regularization, it allows the model to avoid backpropagating through the activation functions, thereby allowing non-differentiable and fast-growing functions in the primitives, and it helps our model avoid premature convergence to local minima.

Because they behave probabilistically, we call nodes in the arguments sublayer *P-nodes*. Figure 3.2 highlights this sublayer structure, while the methods section describes the complete mathematical formalism behind it.

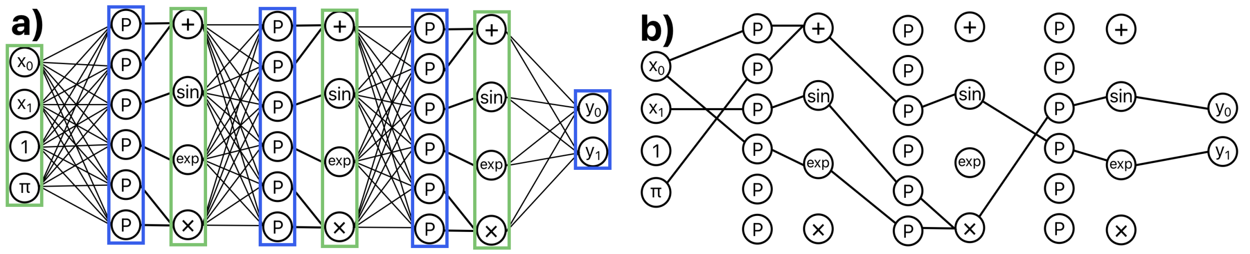


Figure 3.2: (a) A two-output network model with depth $L = 2$, $\vec{x} = [x_0, x_1]$, user-selected constants $\mathcal{C} = [1, \pi]$, and set of primitive functions $\mathbf{\Phi} = (+(\cdot,\cdot), \sin(\cdot), \exp(\cdot), \times(\cdot,\cdot))$. Boxed in blue are the arguments sublayers (composed of P-nodes). For each arguments sublayer, the associated image sublayer (composed of the basis functions from $\mathbf{\Phi}$) is boxed in green and to the right of the corresponding arguments sublayer. Together, these two sublayers define a single hidden layer of our model. The input layer can be thought of as an image layer and the output layer can be thought as an arguments layer. (b) An example of function-specifying directed acyclic graphs (DAGs) that can be sampled from the network in (a). These DAGs represent the functions $y_0 = \exp[\sin(x_0 + \pi)]$ and $y_1 = \sin(\exp(x_0)\sin(x_1))$.

3.2.2 Temperature-controlled connectivity

Instead of dense linear layers, we use *T-softmax layers*. For any temperature $T > 0$, we define a T-softmax layer as a standard T-controlled softmax layer with weighted edges connecting an images sublayer and the subsequent arguments sublayer, in which each P-node from the arguments sublayer probabilistically samples a single edge between itself and a node in the images sublayer. Each node's sampling distribution is given by

$$\mathbf{p}^{(l,i)}(T_l) = \text{softmax}(\mathbf{w}^{(l,i)}; T_l),$$

where $\mathbf{w}^{(l,i)}$ and $\mathbf{p}^{(l,i)}$ are the weights and probabilities for edges leading to the ith P-node of the lth layer and T_l is the fixed temperature for the lth layer. Selecting these edges for all T-softmax layers produces a sparse directed acyclic graph (DAG) specifying a function $\vec{f}$, as seen in Figure 3.2b. While controlling the temperature adjusts the entropy of the distributions over nodes, OccamNet automatically enforces sparsity by sampling a single input edge to each P-node. Adjusting the temperature has no impact on sparsity, but it allows for balancing exploration and exploitation during training.

3.2.3 A neural network as a probability distribution over functions

Through the temperature-controlled connectivity described above, OccamNet can be sampled to produce DAGs corresponding to functions $\vec{f}$. Based on the weights of OccamNet, some DAGs may be sampled with higher or lower probability. In this way, OccamNet can be considered as representing a probability distribution over the set of all possible DAGs, or equivalently over all possible functions sample-able from OccamNet.

Let $\mathbf{W} = \left\{\mathbf{w}^{(l,i)}; 1 \le l \le L, 1 \le i \le N\right\}$. The probability of the model sampling $f_{(i)}$ as its ith output, $q_i(f_{(i)}|\mathbf{W})$, is the product of the probabilities of the edges of $f_{(i)}$'s DAG. Similarly, $q(\vec{f}|\mathbf{W})$, the probability of the model sampling $\vec{f}$, is given by the product of $\vec{f}$'s edges, or $q(\vec{f}|\mathbf{W}) = \prod_{i=0}^{v-1} q_i(f_{(i)}|\mathbf{W})$. For example, in Figure 3.2b, the probabilities of sampling the DAG shown is given by the product of the probabilities sampling each of the edges shown.

In practice, we compute an approximation of this probability which we denote q_{apx}, as described in Methods Section 3.6.1. We find that OccamNet performs well with this approximation. For all other sections of this paper, unless explicitly mentioned, we use q to mean q_{apx}.

We initialize the network with weights $\mathbf{W}_{\text{i}}$ such that $q_{apx}(\vec{f}_1|\mathbf{W}_{\text{i}}) = q_{apx}(\vec{f}_2|\mathbf{W}_{\text{i}})$ for all $\vec{f}_1$ and $\vec{f}_2$ in $\mathcal{F}^L_{\mathbf{\Phi}}$. After training (Section 3.3), the network has weights $\mathbf{W}_{\text{f}}$. The network then selects the function $\vec{f}_{\text{f}}$ with the highest probability $q_{apx}(\vec{f}_{\text{f}}|\mathbf{W}_{\text{f}})$. We discuss our algorithms for initialization and function selection in the Methods section. A key benefit of OccamNet is that, unlike other approaches such as [77], it allows for efficiently identifying the function with the highest probability.

3.3 Training

To express a wide range of functions, we include non-differentiable and fast-growing primitives. Additionally, in symbolic regression, we are interested in finding global minima. To address these constraints, we implement a loss function and training method that combine gradient-based optimization and sampling-based strategies for efficient global exploration of the possible functions. Our loss function and training procedure are closely related to those proposed by [77], differing mainly in the fitness function and regularization terms.

Consider a mini-batch $\mathcal{M} = (X, Y)$ and a sampled function from the network $\vec{f}(\cdot) \sim q(\cdot|\mathbf{W})$. We compute the *fitness* of each $f_{(i)}(\cdot)$ with respect to a training pair $(\vec{x}, \vec{y})$ by evaluating

$$k_i\left(f_{(i)}(\vec{x}), \vec{y}\right) = (2\pi\sigma^2)^{-1/2} \exp\Big(-\left[f_{(i)}(\vec{x}) - (\vec{y})_i\right]^2/(2\sigma^2)\Big),$$

which measures how close $f_{(i)}(\vec{x})$ is to the target $(\vec{y})_i$. The total fitness is determined by summing over the entire mini-batch: $K_i\left(\mathcal{M}, f_{(i)}\right) = \sum_{(\vec{x},\vec{y})\in\mathcal{M}} k_i\left(f_{(i)}(\vec{x}), \vec{y}\right)$.

We then define the loss function

$$H_{q_i}[f_{(i)}, \mathbf{W}, \mathcal{M}] = -K_i\left(\mathcal{M}, f_{(i)}\right) \cdot \log\left[q_i(f_{(i)}|\mathbf{W})\right]. \tag{3.1}$$

as in [77]. As in [77], we train the network by sampling functions, selecting the number λ of functions with the highest fitness for each output, and performing a gradient step based on these highest-fitness functions using the loss defined in Equation 3.1. In practice, λ is a critical hyperparameter to tune as it adjusts the balance between updating toward higher-fitness functions and receiving information about all sampled functions.

To improve implicit function fitting, we implement regularization terms that punish trivial solutions by reducing the fitness K, as discussed in the Methods (Section 3.6.1). We also introduce regularization to restrict OccamNet to solutions that preserve units (Section 3.6.1).

OccamNet can also be trained to find recurrent functions, as discussed in the Methods (Section 3.6.1).

3.4 Results

To empirically validate our model, we first develop a diverse collection of benchmarks in four categories: *Analytic Functions*, simple, smooth functions; *Implicit Functions*, functions specifying an implicit relationship between inputs; *Non-Analytic Functions*, discontinuous and/or non-differentiable functions; *Image/Pattern Recognition*, patterns explained by analytic expressions. We then test OccamNet's performance and ability to scale on real-world symbolic regression datasets. The purpose of these experiments is to demonstrate that OccamNet can perform competitively with other symbolic regression frameworks in a diverse range of applications.

We compare OccamNet with several other symbolic regression methods: Eureqa [68], a genetic algorithm with Epsilon-Lexicase (Eplex) selection [96], AI Feynman 2.0 (AIF) [69, 97], and Deep Symbolic Regression (DSR) [77]. We do not compare to Transformer-based models such as [78] because, unlike our method, these methods utilize a prespecified and immutable set of primitive functions which are not always sufficiently general to fit our experiments. The results are shown in Tables 3.1, 3.2, and 3.3, and we discuss them below. More details about the experimental setup are given in the methods section.

3.4.1 Analytic functions

In Figure 3.3 and Table 3.1 (in the Methods) we present our results on analytic functions. Figure 3a presents an analytic function that is particularly challenging for Eureqa. Figure 3b shows that OccamNet gives competitive success rate to state-of-the-art symbolic regression methods. For all methods besides OccamNet, there is at least one function for which the method gets zero accuracy; in contrast, OccamNet gets non-zero accuracy on every single considered function (Figure 3c).

We highlight the large success rate for function 4, which we originally speculated could easily trick the network with the local minimum $f(x) \approx x + 1$ for large enough x. In

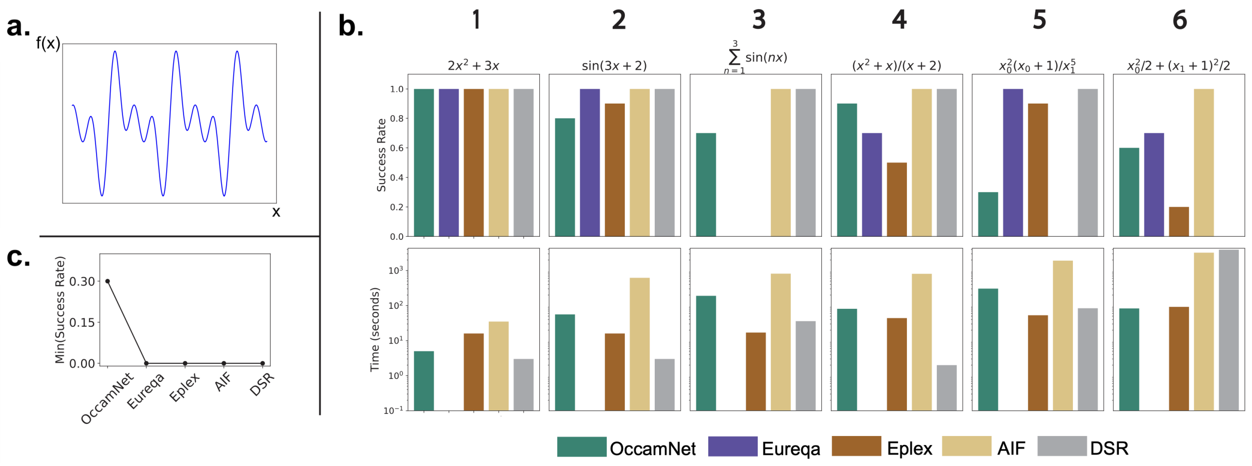


Figure 3.3: Experiment on analytic functions. **a.** A sketch of the function $\sum_{n=1}^{3}\sin(nx)$ as an example of the analytics functions we consider in our work. **b.** Success rate (out of 10 trials) for each of the five methods considered: OccamNet, Eureqa, Eplex, AI Feynman 2.0 (AIF) and Deep Symbolic Regression (DSR) (at the top). Training time for the methods (at the bottom). Eureqa almost always finishes much more quickly than the other methods, so we do not provide training times for Eureqa. We enumerate the functions to ease the discussion. **c.** The "worst-case" performance for each methods, showing the minimal success rate across the six tasks.

contrast, as with the difficulties faced by AI Feynman 2.0, we find that OccamNet often failed to converge for function 5 because it approximated the factor $x_0^2(x_0+1)$ to x_0^3; even when convergence did occur, it required a relatively large number of steps for the network to resolve this additional constant factor. Notably, Eureqa and Eplex had difficulty finding function 3.

AI Feynman 2.0 consistently identifies many of the functions, but it struggles with function 5 and is also generally much slower than other approaches. Eplex also performs well on most functions and is fast. However, like Eureqa, Eplex struggles with functions 3 and 6. We suspect that this is because evolutionary approaches require a larger sample size than OccamNet's training procedure to adequately explore the search space. DSR consistently identifies many of the functions and is very fast. However, DSR struggles to fit Equation 6, which we suspect is because such an equation is complex but can be simplified using feature reuse. OccamNet's architecture allows such feature reuse, demonstrating an advantage of OccamNet's inductive biases.

3.4.2 Non-analytic functions

In Figure 3.4 and Table 3.2 (in the Methods) we benchmark the ability to find several non-differentiable, potentially recursive functions. From our experiments, we highlight both the network's fast convergence to the correct functional form and the discovery of the correct recurrence depth of the final expression. This is pronounced for function 7 in, which is a challenging chaotic series on which Eureqa and Eplex struggle. Interestinly, Eplex fails to identify the simpler functions 1-3 correctly. We suspect that this may be because, for these

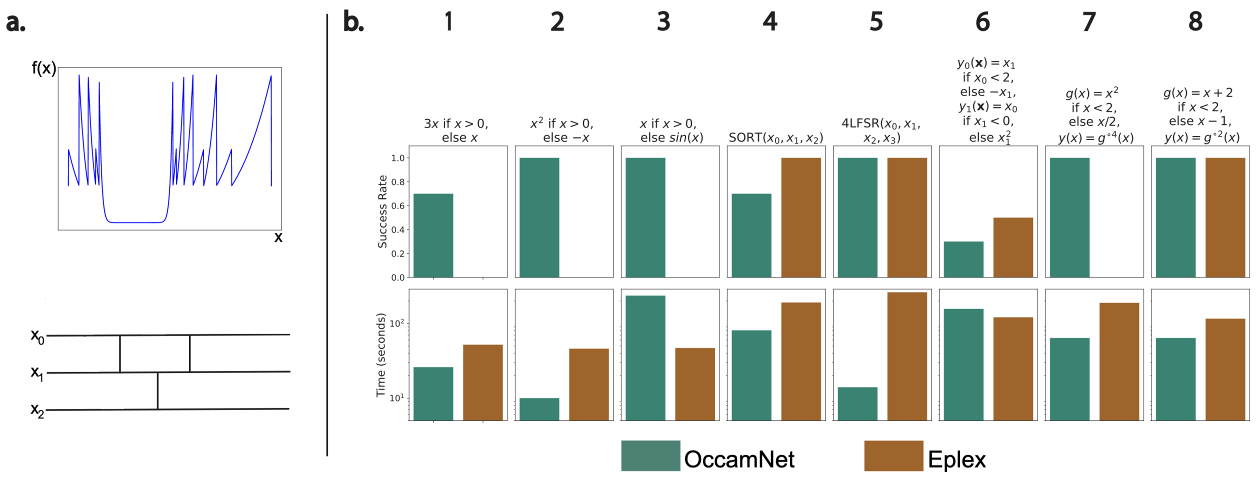


Figure 3.4: Experiments on non-analytic functions. **a.** Two prominent examples of non-analytic functions: The challenging recursion $g(x) = x^2$ if $x < 2$, else $x/2$, $y(x) = g^{\circ 4}(x) = g(g(g(g(x))))$ (top) and a sorting circuit of three numbers (bottom). **b.** Success rate (out of 10 trials) and training time for OccamNet and Eplex. We enumerate the functions to ease the discussion.

experiments, we restrict both OccamNet and Eplex to smaller expression depths. Although OccamNet is able to identify the correct functions with small expression depth, we suspect that Eplex often identifies expressions by producing more complex equivalents to the correct program and so cannot identify the correct function when restricted to simpler expressions.

We also investigated the usage of primitives such as MAX and MIN to sort numbers (function 4), obtaining relatively well-behaved final solutions: the few solutions that did not converge fail only in deciding the second component, y_2, of the output vector. Finally, we introduced binary operators and discrete input sets for testing function 5, a simple 4-bit Linear Feedback Shift Register (LFSR), the function $(x_0, x_1, x_2, x_3) \to (x_0 + x_3 \mod 2, x_0, x_1, x_2)$, which converges fast with a high success rate.

We do not compare to AI Feynman 2.0 in these experiments because AI Feynman does not support the required primitive functions.

3.4.3 Implicit Functions and Image Recognition

Figure 3.5a and Table 3.3 show OccamNet's performance on implicit functions. OccamNet demonstrates an advantage on challenging implicit functions. Notably, Eureqa is unsuccessful in fitting $m_1v_1 - m_2v_2 = 0$ (conservation of momentum). Note that we only compare OccamNet to Eureqa for Implicit Functions because none of the other methods include the regularization that would be necessary to fit such functions.

Figure 3.5b and Table 3.3 demonstrate applications of OccamNet in image recognition, a domain that are not natural for standard symbolic regression baselines, but is somewhat more natural for OccamNet due to its interpretation as a feed-forward neural network.

We train OccamNet to classify MNIST [99][1] in a binary setting between the digits 0 and

[1]Creative Commons Attribution Share Alike 3.0 License

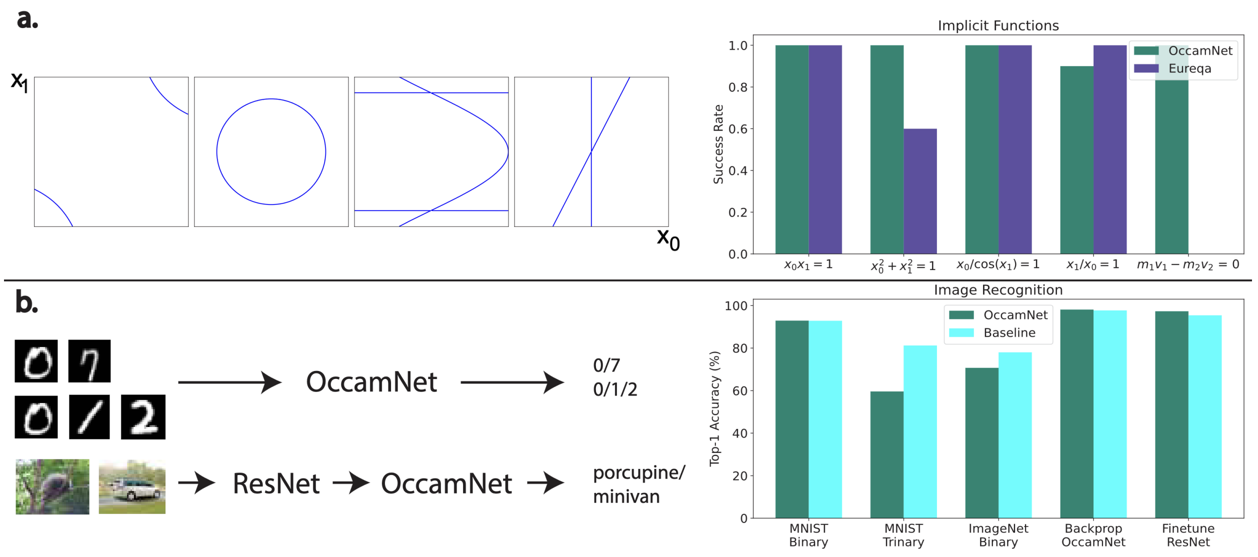


Figure 3.5: Experiments on implicit functions and standard vision benchmarks. **a.** Examples of implicit functions' loci (left) and the corresponding success rate on a suite of implicit functions (right). **b.** Examples of image recognition tasks (left) and the best accuracy from 10 trials for both OccamNet and the baseline. The baseline for MNIST Binary/ Trinary and ImageNet Binary is the HeuristicLab symbolic regression algorithm [98]. The baseline for Backprop OccamNet and Finetune ResNet is a feed-forward neural network with the same number of parameters as OccamNet.

7 (*MNIST Binary*). For this high-dimensional task, we implement OccamNet on an Nvidia V100 GPU, yielding a sizable 8x speed increase compared to a CPU. For MNIST Binary, one of the successful functional fits that OccamNet finds is $y_0(\vec{x}) = \tanh(10(\max(x_{25,15}, x_{26,19}) + \tanh(x_{15,15}) + 2x_{25,10} + 2x_{25,13}))$ and $y_1(\vec{x}) = \tanh(10\tanh(10(x_{18,8} + x_{20,6})))$. The model learns to incorporate pixels into the functional fit that are indicative of the class: here $x_{18,8}$ and $x_{20,6}$ are indicative of the digit 7. These observations hold when we further benchmark the integration of OccamNet with deep feature extractors. We extract features from ImageNet [100][2] images using a ResNet 50 model, pre-trained on ImageNet [92]. For simplicity, we select two classes, "minivan" and "porcupine" (*ImageNet Binary*). OccamNet significantly improves its accuracy by backpropagating through our model using a standard cross-entropy signal. We either freeze the ResNet weights (*Backprop OccamNet*) or finetune ResNet through OccamNet (*Finetune ResNet*). In both cases, the converged OccamNet represents simple rules, ($y_0(\vec{x}) = x_{1838}$, $y_1(\vec{x}) = x_{1557}$), suggesting that replacing the head in deep neural networks with OccamNet might be promising.

3.4.4 Real-world regression datasets

We also test OccamNet's ability to fit real-world datasets, selecting 15 datasets with 1667 or fewer datapoints from the Penn Machine Learning Benchmarks (PMLB[3]) regression datasets

[2]The Creative Commons Attribution (CC BY) License
[3]Creative Commons Attribution 4.0 International License

[101]. These are real-world datasets, and based on their names, we infer that many are from social science, suggesting that they are inherently noisy and likely to follow no known symbolic law. Additionally, 1/3 of the datasets we choose have feature sizes of 10 or greater. These factors make the PMLB datasets challenging symbolic regression tasks. We again compare OccamNet to Eplex and AI Feynman 2.0.[4]

We test OccamNet twice. For the first test, "OccamNet-Small," we test exactly 1,000,000 functions, the same number as we test for Eplex. For the second test, "OccamNet-GPU," we exploit our architecture's integration with the deep learning framework by running OccamNet on an Nvidia V100 GPU and testing a much larger number of functions. We allow AIF to run for approximately as long or longer than OccamNet for each dataset.

As discussed in the SM, we perform grid search on hyperparameters and identify the fits with the best training, validation, and testing Mean Squared Error (MSE) losses. The raw data from these experiments are shown in the SM.

Figure 3.6 shows the relative performance of OccamNet-CPU, OccamNet-GPU, and baselines according to several metrics. As shown in Figure 3.6a-c, overall, Eplex outperforms OccamNet-CPU in training and testing MSE loss, but OccamNet-CPU outperforms Eplex in validation loss. We speculate that OccamNet-CPU's performance drop between the validation and testing datasets being larger than Eplex's performance drop results from overfitting from the larger set of hyperparameter combinations used by OccamNet-CPU (details in the SM).

Additionally, OccamNet-CPU runs faster than Eplex in nearly all datasets tested, often by an order of magnitude (Figure 3.6d). Furthermore, OccamNet is highly parallel and can easily scale on a GPU. Thus, a major advantage of OccamNet is its speed and scalability (see Section 3.4.5 for a further discussion of OccamNet's scaling). Comparing OccamNet-GPU and Eplex demonstrates that OccamNet continues to improve when testing more functions. The testing MSE is where OccamNet-GPU performs worst in comparison to Eplex (see Figure 3.6e), but it still outperforms Eplex at 10 out of 15 of the datasets while running more than nine times faster on average. Thus OccamNet's speed and scalability can be exploited to greatly increase its accuracy at symbolic regression. This demonstrates that OccamNet is a powerful alternative to genetic algorithms for interpretable data modeling.

Additionally, OccamNet outperforms AIF for training, validation, and testing MSE, while running faster. OccamNet-CPU achieves a lower training and validation MSE than AIF for every dataset tested. For training loss, OccamNet-CPU performs better than AIF in 4 out of 7 datasets (Figure 3.6f). Additionally, unlike OccamNet, AIF performs polynomial fitting, giving it an additional advantage. However, the datasets we test are likely a worst-case for AIF; the datasets are small, have no known underlying formula, and we normalize the data prior to training, meaning that AIF will likely struggle not to overfit with its neural network and will also be unlikely to identify graph modularities.

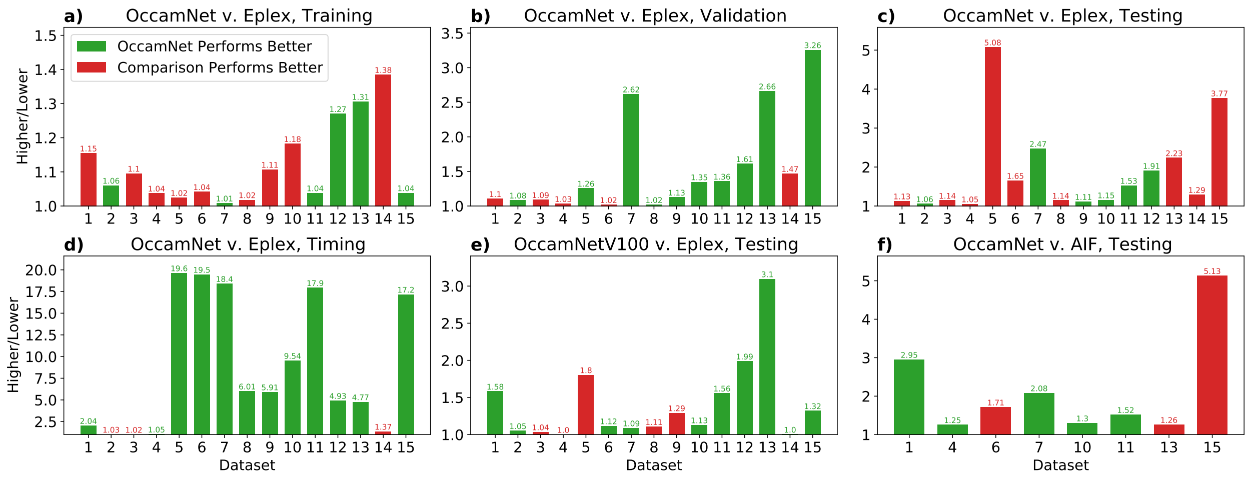


Figure 3.6: Bar charts showing the relative performance between OccamNet-CPU, OccamNet-GPU, and two baseline methods, Eplex and AIF. The x-axis is the dataset involved. The y-axis is the relative performance according to the given metric: the MSE on the training, validation, or testing set or the training time. To compute this relative performance, we divide the higher (worse) performance value by the lower (better) performance value for each dataset. The green bars represent datasets where OccamNet has a lower (better) performance value than the comparison baseline method, and the red bars represent the datasets where the comparison method has a better performance than OccamNet.

3.4.5 Scaling on real-world regression datasets

As discussed in Section 3.4.4, OccamNet-CPU runs far more quickly than Eplex on the same hardware, meaning that it can scale to testing far more functions per epoch than Eplex in the same runtime. To explore this advantage, we compare OccamNet running on an Nvidia V100 GPU (OccamNet-GPU) against Eplex while varying the number of functions sampled per epoch for each method. Since Eplex is not designed to scale on a GPU, we run Eplex on a CPU as before. We benchmark both methods on the same 15 PMLB datasets (see the methods section for more details).

We include and discuss the complete results of this experiment in Appendix B.3. In this section, we highlight key results. Figure 3.7 shows that OccamNet-GPU is often more than an order of magnitude faster than Eplex. Eplex scales quadratically with the number of functions, whereas OccamNet's runtime asymptotes to linear growth. However, the V100 GPU's extreme parallelism initially suppresses OccamNet-GPU's linear time complexity, demonstrating an advantage of OccamNet's ability to scale on a GPU.

In all of the 15 datasets, OccamNet-GPU's training loss decreases with larger runtimes, demonstrating that OccamNet can utilize the greater number of sampled functions that its efficient scaling allows. Additionally, for 11 of the training datasets, the OccamNet-GPU best fit has a MSE that is lower than or equal to the Eplex best fit MSE. Interestingly, OccamNet-

[4] AIF's regression algorithm examines all possible feature subsets, the number of which grows exponentially with the number of features. Accordingly, we only test the datasets with ten or fewer features. AI Feynman 2.0 failed to run on a few datasets. All remaining datasets are included in tables and figures.

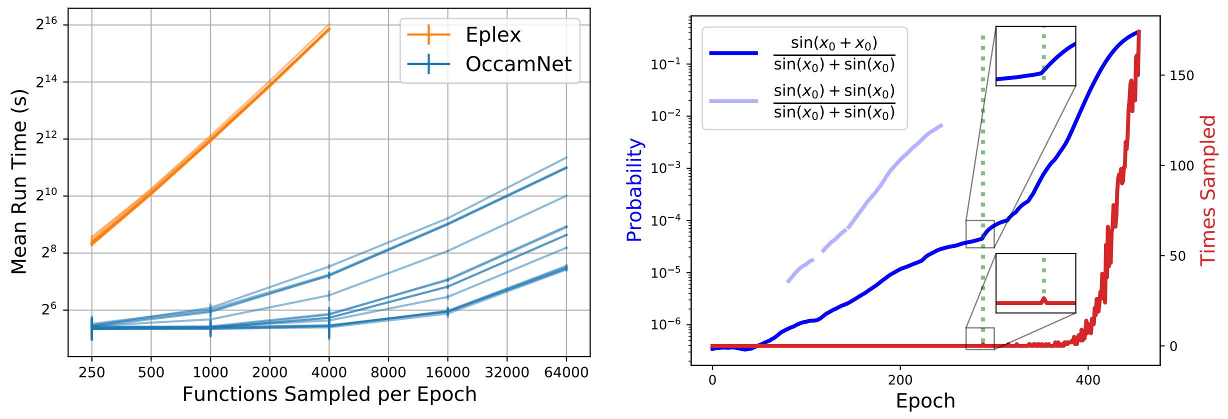


Figure 3.7: *Left:* The run time for OccamNet-GPU or Eplex as a function of the number of functions sampled per epoch. Each curve represents one of the 15 datasets. *Right:* Gradual modularity with training. Dark blue is the probability of the correct function. Light blue is the probability of a suboptimal fit with a high probability early in training. Red corresponds to the number of samples of the correct function. The insets zoom in on the curves around the epoch where the correct function is first sampled.

GPU's validation and testing loss do not always show such a clear trend of improvement with increasing sample size. Given that the training loss does improve, we suspect that this is a case of overfitting. OccamNet-GPU's validation loss does decrease with increasing number of functions sampled for most of the datasets.

3.5 Discussion

Since our experimental settings did not require very large depths, we have not tested the limits of OccamNet-GPU in terms of depth rigorously (preliminary results on increasing the depth for pattern recognition are in the SM). We expect increasing depth to yield significant complications as the search space grows exponentially. We recognize the need to create symbolic regression benchmarks that would require expressions that are large in depth. We believe that other contributions to symbolic regression would also benefit from such benchmarks. Another direction where OccamNet might be improved is low-level optimization that would make the method more efficient to train. For example, in our PMLB experiments, we estimate that OccamNet performs $>$8x as many computations as necessary. Eplex may also benefit from optimization. Finally, similarly to other symbolic regression methods, OccamNet requires a specified set of primitives to fit a dataset. While it is a notable advantage of OccamNet to have non-differentiable primitives, further work needs to be done to explore optimization at a meta level that identifies appropriate primitives for the datasets of interest without having them provided ahead of time.

OccamNet's learning procedure allows it to combine partial solutions into better results. For example in Figure 3.7, the correct function's probability increases monotonically by more than 100 times *before being sampled* because OccamNet samples similar approximate solutions.

OccamNet successfully fits many implicit functions that other neurosymbolic architectures struggle to fit because of the non-differentiable regularization terms required to avoid trivial solutions. Although Eureqa also fits many of these equations, we find that it sometimes requires the data to be ordered by some latent variable and struggles when the dataset is very small. This is likely because Eureqa numerically evaluates implicit derivatives from the dataset [102], which can be noisy when the data is sparse. While [102] propose methods for analyzing unordered data, it is unclear whether these methods have been implemented in Eureqa. Thus, OccamNet seems to shine in its ability to fit unordered and small datasets described by implicit equations (e.g., momentum conservation in line 5 in Table 3.3).

To our knowledge, a unique advantage of our method compared to other symbolic regression approaches is that OccamNet represents complete analytic expressions with a single forward pass. This allows sizable gains when using an AI accelerator, as demonstrated by our experiments on a V100 GPU (Figure 3.6). Furthermore, because of this property, OccamNet can be easily integrated with components from the standard deep learning toolkit. For example, lines 9-10 in Table 3.3 demonstrate integrating OccamNet with other neural networks and optimizing both together, which is not possible with Eureqa. We also conjecture that such integration with autoregressive approaches such as DSR [77] might be challenging as the memory and latency would increase.

An advantage of OccamNet over transformer-based approaches to symbolic regression is that OccamNet can find fits to data regardless of the primitive functions it is given, whereas transformer-based models [78] can only fit functions that contain a certain set of primitive functions chosen at pretraining time. Thus, although transformer-based approaches may outperform OccamNet for functions similar to their training distribution, OccamNet and other similar approaches are more flexible and broadly applicable than transformer-based models. As discussed above, this is the reason that we do not compare against transformer-based methods in our experiments.

3.6 Methods

We divide our methods section into two parts. In Section 3.6.1, we provide a more detailed description of OccamNet, and in 3.6.2 we fully describe the setup for all of our experiments.

3.6.1 Complete Model Description

We divide this section as follows:

1. In Section 3.6.1, we present additional materials that support the figures from the main text.

2. In Section 3.6.1, we describe of OccamNet's sampling process.

3. In Section 3.6.1, we describe OccamNet's probability distribution.

4. In Section 3.6.1, we describe OccamNet's initialization process.

5. In Section 3.6.1, we describe OccamNet's function selection.

6. In Section 3.6.1, we describe OccamNet's loss function.
7. In Section 3.6.1, we describe OccamNet's outer training loop.
8. In Section 3.6.1, we describe OccamNet's two-step training method for fitting constants.
9. In Section 3.6.1, we describe OccamNet's handling of recurrence.
10. In Section 3.6.1, we describe OccamNet's regularization for fitting implicit functions.
11. In Section 3.6.1, we describe OccamNet's procedure for handling functions with undefined outputs.
12. In Section 3.6.1, we describe OccamNet's method for regularizing to respect units.

Supporting Materials for the Main Figures

Tables 3.1, 3.2, and 3.3 present our experiments in a tabular format.

Table 3.1: Analytic Functions. The proportion of 10 trials that converge to the correct analytic function for OccamNet, Eureqa, Eplex, AI Feynman 2.0, and Deep Symbolic Regression (DSR). *sec.* is the average number of seconds for convergence. Eureqa almost always finishes much more quickly than the other methods, so we do not provide training times for Eureqa.

Analytic Functions										
#	Targets	OccamNet	sec.	Eureqa	Eplex	sec.	AI Feynman	sec.	DSR	sec.
1	$2x^2+3x$	1.0	5	1.0	1.0	16	1.0	35	1.0	3
2	$\sin(3x+2)$	0.8	56	1.0	0.9	16	1.0	620	1.0	3
3	$\sum_{n=1}^{3}\sin(nx)$	0.7	190	0.0	0.0	17	1.0	815	1.0	36
4	$(x^2+x)/(x+2)$	0.9	81	0.7	0.5	44	1.0	807	1.0	2
5	$x_0^2(x_0+1)/x_1^5$	0.3	305	1.0	0.9	53	0.0	1918	1.0	84
6	$x_0^2/2+(x_1+1)^2/2$	0.6	83	0.7	0.2	92	1.0	3237	0.0	3935

Sampling from OccamNet

In this section, we more carefully describe OccamNet's sampling process. As described in the main text, we start from a predefined collection of N primitive functions $\mathbf{\Phi} = \{\phi_i(\cdot)\}_{i=1}^{N}$. Each neural network layer is defined by two sublayers, the *arguments* and *image* sublayers. For a network of depth L, each of these sublayers is reproduced L times. Now let us introduce their corresponding hidden states: for $1 \le l \le L$, each l'th arguments sublayer defines a hidden state vector $\widetilde{\mathbf{h}}^{(l)}$, and each l'th image sublayer defines a hidden state $\mathbf{h}^{(l)}$, as follows:

$$\widetilde{\mathbf{h}}^{(l)} = \left[\widetilde{h}_1^{(l)}, \ldots, \widetilde{h}_M^{(l)}\right], \quad \mathbf{h}^{(l)} = \left[h_1^{(l)}, \ldots, h_N^{(l)}\right], \tag{3.2}$$

where

$$M = \sum_{0\le k\le N} \alpha(\phi_k)$$

Table 3.2: Non-analytic Functions. The proportion of 10 trials that converge to the correct function for OccamNet, Eureqa, and Eplex. *sec.* is the average number of seconds for convergence. Eureqa almost always finishes much more quickly than OccamNet and Eplex, so we do not provide training times for Eureqa. *For program #6, Eplex fits y_1 every time and never fits y_0 correctly, so we give it a score of 0.5.

Non-analytic Functions						
#	Targets	OccamNet	sec.	Eureqa	Eplex	sec.
1	$3x$ if $x > 0$, else x	0.7	26	1.0	0.0	52
2	x^2 if $x > 0$, else $-x$	1.0	10	1.0	0.0	46
3	x if $x > 0$, else $\sin(x)$	1.0	236	1.0	0.0	47
4	$\mathsf{SORT}(x_0, x_1, x_2)$	0.7	81	1.0	1.0	191
5	$\mathsf{4LFSR}(x_0, x_1, x_2, x_3)$	1.0	14	1.0	1.0	262
6	$y_0(\vec{x}) = x_1$ if $x_0 < 2$, else $-x_1$ $y_1(\vec{x}) = x_0$ if $x_1 < 0$, else x_1^2	0.3	157	0.1	*0.5	121
7	$g(x) = x^2$ if $x < 2$, else $x/2$ $y(x) = g^{\circ 4}(x)$	1.0	64	0.0	0.0	189
8	$g(x) = x + 2$ if $x < 2$, else $x - 1$ $y(x) = g^{\circ 2}(x)$	1.0	64	0.6	1.0	116

and $\alpha(\phi)$ is the arity of function $\phi(\cdot, \dots, \cdot)$. We also define $\mathbf{h}^{(0)}$ to be the input layer (an image sublayer) and $\widetilde{\mathbf{h}}^{(L+1)}$ to be the output layer (an arguments sublayer). These image and arguments sublayer vectors are related through the primitive functions

$$h_i^{(l)} = \phi_i \left(\widetilde{h}_{j+1}^{(l)}, \dots, \widetilde{h}_{j+\alpha(\phi_i)}^{(l)} \right), \quad j = \sum_{0 \le k < i} \alpha(\phi_k). \tag{3.3}$$

This formally expresses how the arguments connect to the images in any given layer, visualized as the bold edges between sublayers in Figure 1 in the main paper. To complete the architecture and connect the images from layer l to the arguments of layer $(l+1)$, we sample from the softmax of the weights[5]:

$$\widetilde{\mathbf{h}}^{(l+1)} = \begin{bmatrix} \widetilde{h}_1^{(l+1)} \\ \vdots \\ \widetilde{h}_{M_{l+1}}^{(l+1)} \end{bmatrix} \equiv \mathsf{SAMPLE} \left(\begin{bmatrix} \mathsf{softmax}(\mathbf{w}_1^{(l)}; T^{(l)}) \\ \vdots \\ \mathsf{softmax}(\mathbf{w}_{M_{l+1}}^{(l)}; T^{(l)}) \end{bmatrix} \right) \begin{bmatrix} h_1^{(l)} \\ \vdots \\ h_{N_l}^{(l)} \end{bmatrix} \tag{3.4}$$

[5]we define for any $\mathbf{z} = [z_1, \dots, z_{N_l}]$ the softmax function as follows $\mathsf{softmax}(\mathbf{z}; T) := \left[\frac{\exp(z_1/T)}{\sum_{i=1}^{N_l} \exp(z_i/T)}, \dots, \frac{\exp(z_{N_l}/T)}{\sum_{i=1}^{N_l} \exp(z_i/T)} \right]$

Table 3.3: Implicit Functions: The proportion of 10 trials that converge to the correct implicit function for OccamNet and Eureqa. Image Recognition: The best accuracy from 10 trials for both OccamNet and the baseline. The baseline above the mid-line is HeuristicLab [98], and the baseline below the mid-line is a feed-forward neural network with the same number of parameters as OccamNet. *sec.* is the average number of seconds for convergence. The baselines almost always finish much more quickly than OccamNet, so we do not provide baseline training times.

Implicit Functions					Image Recognition				
#	Target	OccamNet	sec.	Eureqa	#	Target	OccamNet	sec.	Baseline
1	$x_0x_1 = 1$	1.0	294	1.0	6	MNIST Binary	92.9	150	92.8
2	$x_0^2 + x_1^2 = 1$	1.0	153	0.6	7	MNIST Trinary	59.6	400	81.2
3	$x_0/\cos(x_1) = 1$	1.0	131	1.0	8	ImageNet Binary	70.7	400	78.0
4	$x_1/x_0 = 1$	0.9	232	1.0	9	Backprop OccamNet	98.1	37	97.7
5	$m_1v_1 - m_2v_2 = 0$	1.0	270	0.0	10	Finetune ResNet	97.3	200	95.4

where the SAMPLE function samples a one-hot row vector for each row based on the categorical probability distribution defined by $\mathsf{softmax}(\mathbf{w}; T)^\top$. Here the hidden states $\mathbf{h}^{(l)}$ and $\widetilde{\mathbf{h}}^{(l+1)}$ have N_l and M_{l+1} coordinates, respectively, and the vectors $\mathbf{w}_i^{(l)}$ represent the ith row of the weights for the lth layer. In practice, we set $T^{(l)}$ to a fixed, typically small, number. The last layer is usually set to a higher temperature to allow more compositionality. These sampled edges are encoded as sparse matrices, through which a forward pass evaluates $\vec{f}$.

It is also possible to implement OccamNet without the sampling part of the propagation. In this case, the softmax of the weight matrices is treated as the weights of linear layers, and we minimize the MSE loss between the outputs and the desired outputs. In practice, however, we find that this approach leads to solutions which are less sparse, which makes this approach less interpretable and often converge to suboptimal local minima.

As shown in Figure 3.8, we use skip connections similar to those in DenseNet [95] and ResNet [92], concatenating each image layer with prior image layers. In particular, such a network now has argument layers computed as

$$\widetilde{\mathbf{h}}^{(l+1)} = \begin{bmatrix} \widetilde{h}_1^{(l+1)} \\ \vdots \\ \widetilde{h}_{M_{l+1}}^{(l+1)} \end{bmatrix} \equiv \mathsf{SAMPLE}\left(\begin{bmatrix} \mathsf{softmax}(\mathbf{w}_1^{(l)}; T^{(l)}) \\ \vdots \\ \mathsf{softmax}(\mathbf{w}_{M_{l+1}}^{(l)}; T^{(l)}) \end{bmatrix} \right) \mathsf{CONCAT}(\mathbf{h}^{(0)}, \mathbf{h}^{(1)}, \ldots, \mathbf{h}^{(l)}), \quad (3.5)$$

where now each vector $\mathbf{w}_i^{(l+1)}$ has $\sum_{i=0}^{l} N_i$ components instead of N_l. Skip connections yield several desirable properties: (*i*) The depth of equations is not fixed, lifting the requirement that the number of layers of the solution be known in advance. (*ii*) The network can find compact solutions as it considers all levels of composition. This promotes solution sparsity and interpretability. (*iii*) Primitives in shallow layers can be reused, analogous to feature reuse in DenseNet. (*iv*) Subsequent layers may behave as higher-order corrections to the solutions found in early layers. Additionally, if we implement OccamNet without sampling, shallow layers are trained before or alongside the subsequent layers due to more direct supervision because gradients can propagate to shallow layers more easily to avoid exploding or vanishing gradients.

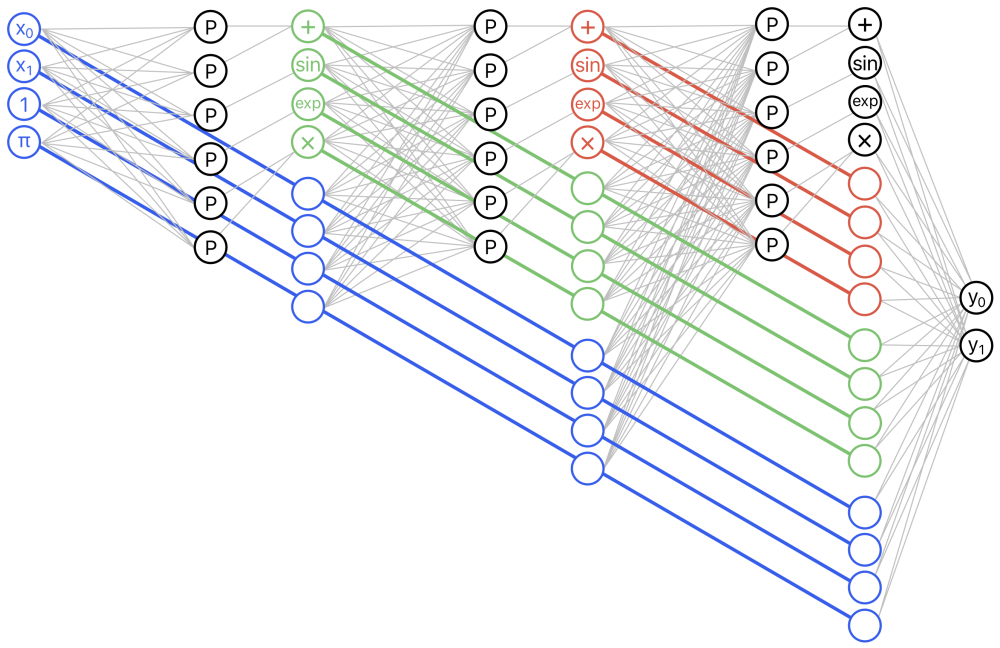


Figure 3.8: Skip connections. Nodes are color coded with lines indicating the origin of the reused neurons.

From Equation (3.3), we see that $M_{l+1} = M = \sum_{0\le k\le N} \alpha(\phi_k)$. If no skip connections are used, $N_l = N = |\mathbf{\Phi}|$. If skip connections are used, however, N_l grows as l increases. We demonstrate how the scaling grows as follows. Let u be the number of inputs and v be the number of outputs. When learning connections from images to arguments at layer l ($1 \le l \le L$), there will be skip connections from the images of the previous l layers $0, 1, \ldots, l-1$. Hence the ith layer has an image size of $u + iN$, as shown in Figure 3.8. We learn linear layers from these images to arguments, and the number of arguments is always M. Thus, in total, we have the following number of parameters:

$$v(u + (L+1)N) + M \sum_{i=0}^{L-1} (u + iN) \in O(NML^2).$$

Note that in the above discussion we assume that M remains constant. However, to be able to represent all functions up to a particular depth, we must repeat primitives in earlier layers, causing M to grow exponentially. For small numbers of layers, this is not problematic. If a larger expression depth is required, one can avoid primitives and increase the number of layers beyond what is necessary. This makes additional copies of each primitive available for use without requiring an exponential growth in the layer size.

OccamNet's Probability Distribution

OccamNet parametrizes not only the probability of sampling a given function $\vec{f} = (f_{(0)}, \ldots, f_{(v-1)})^\top$ but also the probability of sampling each $f_{(i)}$ independently of the other components of $\vec{f}$. As discussed in the main text, the probability of the model sampling $f_{(i)}$ as its ith output, $q_i(f_{(i)}|\mathbf{W})$, is the product of the probabilities of the edges of $f_{(i)}$'s DAG. Similarly,

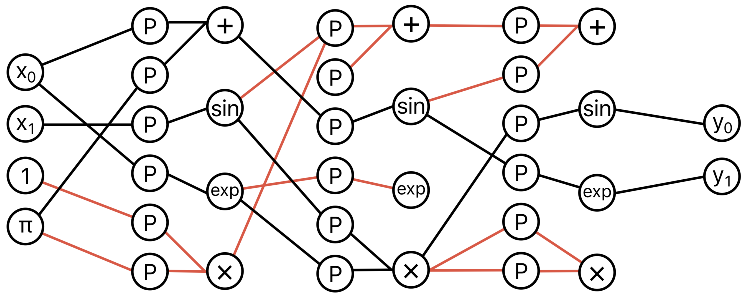


Figure 3.9: A demonstration of the dropped connections from sampled paths in OccamNet. All red paths are dropped from the final symbolic form of the sampled function because they are not directly connected to the outputs. These paths are unnecessarily computed during OccamNet's training process, leading to potential slowdowns in training.

$q(\vec{f}|\mathbf{W})$, the probability of the model sampling $\vec{f}$, is given by the product of $\vec{f}$'s edges, or $q(\vec{f}|\mathbf{W}) = \prod_{i=0}^{v-1} q_i(f_{(i)}|\mathbf{W})$.

Because $q(\cdot|\cdot)$ is a probability distribution, we have $\sum_{\vec{f}\in\mathcal{F}_{\mathbf{\Phi}}^L} q(\vec{f}|\mathbf{W}) = 1$ and $q(\vec{f}|\mathbf{W}) \geq 0$ for all $\vec{f}$ in $\mathcal{F}_{\mathbf{\Phi}}^L$. Similar results hold for the probability distributions of each component $f_{(i)}$.

OccamNet's sampling process involves independently sampling connections from each layer. Although each of OccamNet's layers represents an independent probability distribution, when sampling a function, the layers do not act independently. This is because the samples from layers closer to the outputs inform which of the sampled connections from previous layers are used. In particular, the full DAG that OccamNet samples has many disconnected components, and all components of the DAG which are not connected to any of the output nodes are effectively trimmed (See Figure 3.9). This is advantageous as it allows OccamNet to produce very different distributions of functions for different choices of connections in the final few layers, thereby allowing OccamNet to explore multiple classes of functions simultaneously.

As discussed in the main text, $q(\vec{f}|\mathbf{W})$ is the product of the probabilities of the sampled connections in $\vec{f}$'s DAG which are connected to the output nodes. However, in practice, we compute probabilities of functions in a feed-forward manner. This computation underestimates some probabilities; it actually computes an estimate $q_{apx}(\vec{f}|\mathbf{W})$ of $q(\vec{f}|\mathbf{W})$.

To compute the probability of a given function, we assign each image and argument node a probability given this function's DAG. We denote the probability of the i'th node of the l'th image layer with $p_i^{(l)}$ and the probability of the i'th node of the l'th argument layer with $\tilde{p}_i^{(l)}$.

We propagate probabilities as follows. If the i'th image node in layer l is connected to the j'th argument node in layer $l+1$, the probability of the j'th argument node in layer l is

$$\tilde{p}_j^{(l+1)} = p_i^{(l)} \cdot p_i^{(l,j)}(T^{(l)}). \tag{3.6}$$

The ith image node of the lth layer then has probability given by

$$p_i^{(l)} = \prod_{k=n+1}^{n+\alpha(\phi_i)} \tilde{p}_k^{(l)}, \quad n = \sum_{j=1}^{i-1} \alpha(\phi_j), \tag{3.7}$$

where $\alpha(f)$ denotes the number of inputs to f. Finally, to calculate the probability of a function, we multiply the probabilities of the output nodes.

This algorithm computes function probabilities correctly unless a function's DAG has multiple nodes connecting to the same earlier node in the DAG. In this case, the probability of the earlier node is included multiple times in the final function probability, producing an estimate that is below the true probability of sampling the function.

In practice, we find that this biased evaluation of probabilities does not substantially affect OccamNet training. Note that when we equalize all functions to have the same probability (Section 3.6.1) or sample the highest probability function (Section 3.6.1), we do so with respect to the probability estimate q_{apx}, not with respect to q. In this paper, we use q to mean q_{apx} unless otherwise specified.

Initialization

When beginning this project, we originally initialized all model weights to 0. However, this initializes complex functions, which have DAGs with many more edges than simple functions, to low probabilities. As a result, we found in practice that the network sometimes struggled to converge to complex functions with high fitness $K(\mathcal{M}, f)$ because their initial low probabilities meant that they were sampled far less often than simple functions. This is because even if complex functions have a higher probability increase than simple functions when they are sampled, the initial low probabilities caused the complex functions to be sampled far less and to have an overall lower expected probability increase.

To address this issue, we now use a second initialization algorithm, which initializes all functions to equal probability. This initialization algorithm iterates through the layers of the network. In practice, to balance effects discussed at the end of this section, we initialize to weights interpolated between 0 and the algorithm discussed below. More details are given at the end of this section.

The algorithm to initialize all functions with equal probability establishes as an invariant that, after assigning the weights up to the lth layer, all paths leading to a given node in the lth argument layer have equal probabilities. Then, each argument layer node has a unique corresponding probability, the probability of all paths up to that node. We denote the probability of the ith node in the lth argument sublayer as $\widetilde{p}_i^{(l)}$, because it is the probability of *any* path leading to the ith node in the lth argument sublayer. Because each argument layer node has a corresponding probability, each image layer node must also have a unique corresponding probability, which, for the ith node in the lth image sublayer, we denote as $p_i^{(l)}$. Again, we use the notation $p_i^{(l)}$ because this is the probability of *any* path leading to the ith node in the lth image sublayer. These image layer probabilities are given by

$$p_i^{(l)} = \prod_{k=n+1}^{n+\alpha(\phi_i)} \widetilde{p}_k^{(l)}, \quad n = \sum_{j=1}^{i-1} \alpha(\phi_j). \tag{3.8}$$

Our algorithm starts with input layer, or the 0th image layer. Paths to any node in the input layer have no edges so they all have probability 1. Thus, we initialize $p_i^{(0)} = 1$ for all i. As the algorithm iterates through all subsequent T-Softmax layers, the invariant established above provides a system of linear equations involving the desired connection probabilities, which the algorithm solves. The algorithm groups the previous image layer according to the node probabilities, obtaining a set of ordered pairs $\{(p'^{(l)}_a, n^{(l)}_a)\}_{i=a}^{k}$ representing $n^{(l)}_a$ nodes with probability $p'^{(l)}_a$ in the lth layer. Note that if two image nodes have the same probability $p^{(l)}_i = p^{(l)}_j$, then the edges between any argument node in the next layer and the two image nodes must have the same probability in order to satisfy the algorithm's invariant: $p^{(l,k)}_i = p^{(l,k)}_j$. Then, we define $p'^{(l,i)}_a$ as the probability of the edges between the image nodes with probability $p'^{(l)}_a$ and the ith argument P-node of the lth layer. The probabilities of the edges to a given P-node sum to 1, so for each j, we must have $\sum_a n^{(l)}_a p'^{(l,i)}_a = 1$. Further, the algorithm requires that the probability of a path to a P-node through a given connection is the same as the probability of a path to that P-node through any other connection. The probability of a path to the ith P-node through a connection with probability $p'^{(l,i)}_a$ is $p'^{(l)}_a p'^{(l,i)}_a$, so we obtain the equations $p'^{(l)}_0 p'^{(l,i)}_0 = p'^{(l)}_a p'^{(l,i)}_a$, for all a and i. These two constraints give the vector equation

$$\begin{bmatrix} n^{(l)}_0 & n^{(l)}_1 & n^{(l)}_2 & \cdots & n^{(l)}_k \\ p'^{(l)}_0 & -p'^{(l)}_1 & 0 & \cdots & 0 \\ p'^{(l)}_0 & 0 & -p'^{(l)}_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p'^{(l)}_0 & 0 & 0 & \cdots & -p'^{(l)}_k \end{bmatrix} \begin{bmatrix} p'^{(l,j)}_0 \\ p'^{(l,j)}_1 \\ p'^{(l,j)}_2 \\ \vdots \\ p'^{(l,j)}_k \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \tag{3.9}$$

for all $1 \le j \le M$. The algorithm then solves for each $p'^{(l,j)}_a$.

After determining the desired probability of each connection of the lth layer, the algorithm computes the SPL weights $\mathbf{w}'^{(l,j)}$ that produce the probabilities $p'^{(l,j)}_a$. Since there are infinitely many possible weights that produce the correct probabilities, the algorithm sets $w'^{(l,j)}_0 = 0$. Then, the algorithm uses the softmax definition of the edge probabilities to determine the required value of $\sum_{n=1}^{k} \exp\left(w'^{(l,j)}_n / T^{(l)}\right)$:

$$\begin{aligned} p'^{(l,j)}_0 &= \frac{\exp\left(w'^{(l,j)}_0 / T^{(l)}\right)}{\sum_{n=1}^{k} \exp\left(w'^{(l,j)}_n / T^{(l)}\right)} \\ &= \frac{1}{\sum_{n=1}^{k} \exp\left(w'^{(l,j)}_n / T^{(l)}\right)} \end{aligned}$$

so

$$\sum_{a=1}^{k} \exp\left(w'^{(l,j)}_a / T\right) = 1/p'^{(l,j)}_0.$$

Substituting this equation into the expression for the other probabilities gives

$$\begin{aligned} p'^{(l,j)}_a &= \exp\left(w'^{(l,j)}_a / T^{(l)}\right) / \left(\sum_{n=1}^{k} \exp\left(w'^{(l,j)}_n / T^{(l)}\right)\right) \\ &= p'^{(l,j)}_0 \exp\left(w'^{(l,j)}_a / T^{(l)}\right). \end{aligned}$$

Solving for ${w'}_i^{(l,j)}$ gives

$$ {w'}_a^{(l,j)} = T^{(l)} \log \left({p'}_a^{(l,j)} / {p'}_0^{(l,j)} \right), \tag{3.10} $$

which the algorithm uses to compute ${w'}_i^{(l,j)}$.

After determining the weights ${w'}_a^{(l,j)}$ the algorithm assigns them to the corresponding $w_a^{(l,j)}$. In particular, if the ith image node has probability ${p'}_a^{(l)}$, the weights of edges to the ith node are given by $w_i^{(l,j)} = {w'}_k^{(l,j)}$, for all j. The algorithm then determines the values of $\widetilde{p}_j^{(l+1)}$, given by $\widetilde{p}_j^{(l+1)} = p_i^{(l)} p_i^{(l,j)}$. Finally, the algorithm determines $p_i^{(l+1)}$ using Equation 3.8 and repeats the above process for subsequent layers until it reaches the end of the network.

In summary, the algorithm involves the following steps:

1. Set $l = 0$.
2. Set $p_i^{(l)} = 1$.
3. Increment l by 1.
4. Compute $\{({p'}_a^{(l)}, n_a^{(l)})\}_{i=a}^{k}$ and use Equation 3.9 to compute ${p'}_a^{(l,j)}$.
5. Set ${w'}_a^{(l,j)}$ according to Equation 3.10.
6. If $l < L + 1$, Compute $\widetilde{p}_i^{(l+1)}$ and $p_i^{(l+1)}$.
7. Return to step 3 until $l = L + 1$.

This algorithm efficiently equalizes the probabilities of all functions in the network. In practice, however, we find that perfect equalization of functions causes activation functions with two inputs to be highly explored. This is because there are many more possible functions containing activation functions with two inputs than with one input. Additionally, as mentioned in Section 3.6.1, in this section we have implicitly been using the approximate probability $\tilde{q}$. This probability underestimates many functions that include activation functions with two or more inputs because these functions are those which can use a node multiple times in their DAG. As a result, although all functions will have an equal $\tilde{q}$, some functions with multiple inputs will have larger q than other functions, and q is what determines the probability of being sampled. In practice, therefore, we find that a balance between initializing all weights to one and initializing all functions to equal probability is most effective for exploring all types of functions.

To implement this balance, we create an equalization hyperparameter, E. If $E = 0$, we initialize all weights to 1 as in the original OccamNet architecture. If $E \neq 0$, we use the algorithm presented above to initialize the weights and then divide all of the weights by E. For $E > 1$, this has the effect of initializing weights between the two initialization approaches. In practice, we find that values of $E = 1$ and $E = 5$ are most effective for exploring all types of functions (See Section 3.6.2).

Function Selection

As discussed in the main text, after training using a sampling strategy, the network selects the function $\vec{f}$ with the highest probability $q(\vec{f}|\mathbf{W})$.

We develop a dynamic programming algorithm that determines the DAG with the highest probability. The algorithm steps sequentially through each argument layer, and at each argument layer it determines the maximum probability path to each argument node. Knowing the maximum probability paths to the previous argument layer nodes allows the algorithm to easily determine the maximum probability paths to the next argument layer.

As with the network initialization algorithm, the function selection algorithm associates the ith P-node of the lth argument sublayer with a probability, $\widetilde{p}_i^{(l)}$, which represents the highest probability path to that node. Similarly, we let $p_i^{(l)}$ represent the assigned probability of the ith node of the lth image sublayer, defined as the highest probability path to a given image node. $p_i^{(l)}$ can once again be determined from $\widetilde{p}_i^{(l)}$ using Equation 3.8. Further, the algorithm associates each node with a function, $\widetilde{f}_i^{(l)}$ for argument nodes and $f_i^{(l)}$ for image nodes, which represents the highest probability function to the corresponding node. Thus, $\widetilde{f}_i^{(l)}$ has probability $\widetilde{p}_i^{(l)}$, and $f_i^{(l)}$ has probability $p_i^{(l)}$. Further, $f_i^{(l)}$ is determined from $\widetilde{f}_i^{(l)}$ using

$$f_i^{(l)}(\vec{x}) = \phi_i\left(\widetilde{f}_{n+1}^{(l)}(\vec{x}), \ldots, \widetilde{f}_{n+\alpha(\phi_j)}^{(l)}(\vec{x})\right), \quad n = \sum_{j=1}^{i-1} \alpha(\phi_j). \tag{3.11}$$

The algorithm iterates through the networks layers. At the lth layer, it determines the maximum probability path to each argument node, computing

$$\widetilde{p}_i^{(l+1)} = \mathrm{MAX}\left(p_0^{(l)} p_0^{(l,i)}, \ldots, p_N^{(l)} p_N^{(l,i)}\right)$$
$$\widetilde{f}_i^{(l+1)} = \begin{cases} f_0^{(l)} & \text{if } \widetilde{p}_i^{(l+1)} = p_0^{(l)} p_0^{(l,i)} \\ f_1^{(l)} & \text{if } \widetilde{p}_i^{(l+1)} = p_1^{(l)} p_1^{(l,i)} \\ \vdots & \qquad \vdots \\ f_N^{(l)} & \text{if } \widetilde{p}_i^{(l+1)} = p_N^{(l)} p_N^{(l,i)} \end{cases}.$$

Next, it determines the maximum probability path up to each image node, computing $p_i^{(l+1)}$ and $f_i^{(l+1)}$ using Equations 3.8 and 3.11, respectively. The algorithm repeats this process until it reaches the output layer, at which point it returns $\vec{f}_{\max} = [\widetilde{f}_1^{(L)}, \ldots, \widetilde{f}_N^{(L)}]^\top$ and $p_{\max} = \prod_{i=1}^{N} \widetilde{p}_i^{(L)}$.

An advantage of this process is that identifying the highest probability function has the same computational complexity as sampling functions. In particular, the complexity at each layer is $O(MN_i)$, leading to an overall complexity of $O(NML^2)$ if skip connections are included.

Loss Function and its Gradient

We train our network on mini-batches of data to provide flexibility for devices with various memory constraints. Consider a mini-batch $\mathcal{M} = (X, Y)$, and a sampled function from the network $\vec{f}(\cdot) \sim q(\cdot|\mathbf{W})$. We compute the *fitness* of each $f_{(i)}(\cdot)$ with respect to a training pair $(\vec{x}, \vec{y})$ by evaluating the likelihood

$$k_i\left(f_{(i)}(\vec{x}), \vec{y}\right) = (2\pi\sigma^2)^{-1/2} \exp\Big(-\left[f_{(i)}(\vec{x}) - (\vec{y})_i\right]^2/(2\sigma^2)\Big),$$

which is a Normal distribution with mean $(\vec{y})_i$ and variance σ^2, and measures how close $f_{(i)}(\vec{x})$ is to the target $(\vec{y})_i$. The fitness can be also viewed as a Bayesian posterior with a noninformative prior. The total fitness is determined by summing over the entire mini-batch: $K_i\left(\mathcal{M}, f_{(i)}\right) = \sum_{(\vec{x},\vec{y})\in\mathcal{M}} k_i\left(f_{(i)}(\vec{x}), \vec{y}\right)$.

The variance σ^2 of $k_i\left(f_{(i)}(\vec{x}), \vec{y}\right)$ characterizes the fitness function's smoothness. As $\sigma^2 \to 0$, the fitness is a delta function with nonzero fitness for some $(\vec{x}, \vec{y})$ only if $f_{(i)}(\vec{x}) = (\vec{y})_i$. Similarly, a large variance characterizes a fitness in which potentially many solutions provide accurate approximations, increasing the risk of convergence to local minima. In the former case, learning becomes harder as few $f_{(i)}(\cdot)$ out of exponentially many sampleable functions result in any signal, whereas in the latter case learning might not converge to the optimal solution. We let σ^2 be a network hyperparameter, tuned for the tradeoff between ease of learning and solution optimality for different tasks.

Similar to [77], we use a loss function for backpropagating on the weights of $q(\cdot|\mathbf{W})$:

$$H_{q_i}[f_{(i)}, \mathbf{W}, \mathcal{M}] = -K_i\left(\mathcal{M}, f_{(i)}\right) \cdot \log\left[q_i(f_{(i)}|\mathbf{W})\right]. \tag{3.12}$$

We can interpret (3.12) as the cross-entropy of the posterior for the target and the probability of the sampled function $f_{(i)}$. If the sampled function $f_{(i)}$ is close to $f^*_{(i)}$, then $K_i(\mathcal{M}, f_{(i)})$ will be large, and the gradient update below will also be large:

$$\nabla_{\mathbf{W}} H_{q_i}\left[f_{(i)}, \mathbf{W}, \mathcal{M}\right] = -\frac{\nabla_{\mathbf{W}} q_i(f_{(i)}|\mathbf{W})}{q_i(f_{(i)}|\mathbf{W})} K_i\left(\mathcal{M}, f_{(i)}\right). \tag{3.13}$$

The first term on the right-hand side (RHS) of update (3.13) increases the probability of the function $f_{(i)}$. The second term on the RHS is maximal when $f_{(i)}(\vec{x}) = f^*_{(i)}(\vec{x})$. Importantly, the second term approaches zero as $f_{(i)}$ deviates from $f^*_{(i)}$. If the sampled function is far from the target, then the probability update is suppressed by $K_i(\mathcal{M}, f_{(i)})$. Therefore, we only optimize the probability for functions close to the target. Note that in (3.13) we backpropagate only through the probability of the function $f_{(i)}$ given by $q_i\left(f_{(i)}|\mathbf{W}\right)$, whose value *does not* depend on the primitives in $\mathbf{\Phi}$, implying that the primitives can be non-differentiable. This is particularly useful for applications requiring non-differentiable primitive functions. Furthermore, this loss function allows non-differentiable regularization terms, which greatly expands the regularization possibilities.

Sample-based Training

We use a sampling-based strategy to update our model, explained below without constant fitting for simplicity. This training procedure was first proposed in Risk-Seeking Policy Gradients [77]. We denote $\mathbf{W}^{(t)}$ as the set of weights at training step t, and we fix two hyperparameters: R, the number of functions to sample at each training step, and λ, or the *truncation parameter*, which defines the number of the R paths chosen for optimization via (3.13). We initialize $\mathbf{W}^{(0)}$ as described in Section 3.2.2. We then proceed as follows:

1. Sample R functions $\vec{f}_1, \ldots, \vec{f}_R \sim q(\cdot|\mathbf{W}^{(t)})$. We denote the jth output of $\vec{f}_i$ as $f_{i(j)}$.

2. For each output j, sort $f_{i(j)}$ from greatest to least value of $K_j\left(\mathcal{M}, f_{i(j)}\right)$ and select the top λ functions, yielding a total of $v\lambda$ selected functions $g_{1,j}, \ldots, g_{\lambda,j}$. The total loss is then given by $\sum_{i=1}^{\lambda}\sum_{j=0}^{v-1} H_{q_j}[g_{i,j}, \mathbf{W}, \mathcal{M}]$, which yields the training step gradient update:
$$-\sum_{i=1}^{\lambda}\sum_{j=0}^{v-1} \frac{\nabla_{\mathbf{W}} q_j(g_{i,j}|\mathbf{W})}{q_j(g_{i,j}|\mathbf{W})}\, K_j(\mathcal{M}, g_{i,j}). \tag{3.14}$$
Notice that through (3.14) we have arrived at a modified REINFORCE update [103], where the policy is $q_i(\cdot|\cdot)$ and the regret is the fitness $K_i(\cdot,\cdot)$.

3. Perform the gradient step (3.14) on $\mathbf{W}^{(t)}$ for all selected paths to obtain $\mathbf{W}^{(t+1)}$. In practice, we find that the Adam algorithm [104] works well for this step.

4. Set $t = t + 1$ and repeat from Step 1 until a stop criterion is met.

Note that Equations (3.13) and (3.14) represent different objective functions – we use (3.14). The benefit of using Equation (3.14) is that accumulating over the top $v\lambda$ best fits to the target allows for explorations of function compositions that contain desired components but are not fully developed. In practice, we find that reweighting the importance of the top-$v\lambda$ routes, substituting $K_j'(\mathcal{M}, g_{i,j}) = K_j(\mathcal{M}, g_{i,j})/i$, improves convergence speed by biasing updates towards the best routes.

Constant Fitting

Thus far, our method works for functions with constants known *a priori*. Examples of such functions include x^2 or $x + \pi$ if $(\cdot)^2$ and π are provided respectively as primitives and constants ahead of time. In some cases, however, we may wish to fit functions that involve constants that are not known *a priori*. To fit such undetermined constants, we use activation functions with unspecified constants, such as x^c and $c \cdot x$ (c is undefined). We then combine the training process described in Section 3.6.1 with a constant fitting training process.

The two-step training process works as follows: We first sample a batch $\mathcal{M}$ and a function batch $(\vec{f}_1, \ldots, \vec{f}_R)$. Next, for each function $\vec{f}_i$, we fit the unspecified constants to $\mathcal{M}$ in $\vec{f}_i$ using gradient descent. Any other constant optimization method would also work. Finally, we update the network weights according to Section 3.6.1, using the fitness K of the constant-fitted function batch. To increase training speed, we store each function's fitted constants for reuse.

Recurrence

OccamNet can also be trained to find recurrence relations, which is of particular interest for programs that rely on FOR or WHILE loops. To find such recurrence relations, we assume a *maximal* recursion D. We use the following notation for recurring functions: $f^{\circ(n+1)}(x) \equiv f^{\circ n}(f(x))$, with base case $f^{\circ 1}(x) \equiv f(x)$.

To augment the training algorithm, we first sample $(\vec{f}_1, \ldots, \vec{f}_R) \sim q(\cdot|\mathbf{W}^{(t)})$. For each $\vec{f}_i$, we compute its recurrence to depth D as follows $\left(\vec{f}_i^{\circ 1}, \vec{f}_i^{\circ 2}, \ldots, \vec{f}_i^{\circ D}\right)$, obtaining a collection of RD functions. Training then continues as usual; we compute the corresponding $K_j(\mathcal{M}, \vec{f}_{i(j)}^{\circ n})$, select the best $v\lambda$, and update the weights. It is important to note that we consider all depths up to D since our maximal recurrence depth might be larger than the one for the target function.

Note that we do not change the network architecture to accommodate for recurrence depth $D > 1$. As described in the main text, we can efficiently use the network architecture to evaluate a sampled function $\vec{f}(\vec{x})$ for a given batch of $\vec{x}$. To incorporate recurrence, we take the output of this forward pass and feed it again to the network D times, similar to what is typically done for recurrent neural networks. The resulting outputs are evaluations $\left(\vec{f}_i^{\circ 1}(\vec{x}), \vec{f}_i^{\circ 2}(\vec{x}), \ldots, \vec{f}_i^{\circ D}(\vec{x})\right)$ for a given batch of $\vec{x}$.

Regularization

As discussed in the main text, to improve implicit function fitting, we implement a regularized loss function,

$$K_i'(\mathcal{M}, f) = K_i(\mathcal{M}, f) - s \cdot r[f],$$

for some regularization function r, where $s = n(\mathcal{M})/\sqrt{2\pi\sigma^2}$ is the maximum possible value of $K_i(\mathcal{M}, f)$. We define

$$r[f] = w_\phi \cdot \phi[f] + w_\psi \cdot \psi[f] + w_\xi \cdot \xi[f] + w_\gamma \cdot \gamma[f],$$

where $\phi[f]$ measures trivial operations, $\psi[f]$ measures trivial approximations, $\xi[f]$ measures the number of constants in f, $\gamma[f]$ measures the number of activation functions in f, and w_ϕ, w_ψ, w_ξ, and w_γ, are weights for their respective regularization terms. We now discuss each of these regularization terms in more detail.

The $\phi[f]$ Regularization Term: The $\phi[f]$ term measures whether the unsimplified form of f contains trivial operations, by which we mean operations that simplify to 0, 1, or the identity. For example, division is a trivial operation in x/x, because the expression simplifies to 1. Similarly, $1 \cdot x$, x^1, and x^0 are all trivial operations. We punish these trivial operations because they produce constant outputs without adding meaning to an expression.

To detect trivial operations, we employ two procedures. The first uses the `SymPy` package [105] to simplify f. If the simplified expression is different from the original expression, then there are trivial operations in f, and this procedure returns 1. Otherwise the first procedure returns 0. Unfortunately, the `SymPy` `==` function to test if functions are equal often incorrectly indicates that nontrivial functions are trivial. For example, `SymPy`'s `simplify` function, which we use to test if a function can be simplified, converts $x+x$ to $2\cdot x$, and the `==` function states that $x + x \neq 2 \cdot x$. To combat this, we develop a new function, `sympyEquals` which corrects for these issues with `==`. The `sympyEquals` is equivalent to `==`, except that it does not take the order of terms into account, and it does not mark expressions such as $x + x$ and $x \cdot x$ as unsimplified. We find that this greatly improves implicit function fitting.

The constant fitting procedure often produces functions that only differ from a trivial operation because of imperfect constant fitting, such as $f(x_0) = x_0^{0.0001}$, which is likely meant to represent x_0^0. `SymPy`, however, will not mark this function as trivial. The second procedure addresses this issue by counting the constant activations, such as $x_0^{0.0001}$, $1.001 \cdot x_0$, and $x_0 + 0.001$, which approximate trivial operations. For the activation function $f(x) = x + c$, if the fitted c satisfies $-0.1 < c < 0.1$, the procedure adds 1 to its counter. Similarly, for the activation functions $f(x) = cx$ and $f(x) = x^c$, if the fitted c satisfies $-0.1 < c < 0.1$ or $0.9 < c < 1.1$, the procedure adds 1 to its counter. We select these ranges to capture instances of imperfect constant fitting without labeling legitimate solutions as trivial. After checking all activation functions used, the procedure returns the counter.

The $\phi[f]$ term returns the sum of the outputs of the first and second procedures. We find that a weight of $w_\phi \approx 0.7$ for $\phi[f]$ is most effective in our loss function. This value of w_ϕ ensures that most trivial f have $K_i(\mathcal{M}, f) - s \cdot w_\phi \cdot \phi[f] < 0$, thus actively reducing the weights corresponding to functions with trivial operations, without over punishing functions and hindering learning.

The $\psi[f]$ Regularization Term: When punishing trivial operations using the ϕ term, we find that the network discovers many nontrivial operations which very closely approximate trivial operations by exploiting portions of functions with near-zero derivatives, which can be used to artificially compress data. For example, $\cos(x/2)$ closely approximates 1 if $-1 < x < 1$. Unfortunately, it is often difficult to determine if a function approximates a trivial function simply from its symbolic representation. This issue is also identified in [102].

To detect these trivial function approximations, we develop an approach that analyzes the activation functions' outputs *during* the forward pass. The $\psi[f]$ term counts the number of activation functions which, during a forward pass, the network identifies as possibly approximating trivial solutions, as well as a metric for how close to trivial these functions are. For each primitive function, the network stores values around which outputs of that function often cluster artificially. Table 3.4 lists the primitives which the network tests for clustering.

The procedure for determining ψ is as follows. The algorithm begins with a counter of 0. During the forward pass, if the network reaches a primitive function ϕ listed in Table 3.4, the algorithm tests each ordered tuple (ϕ, a, δ) from Table 3.4, where a is the point tested for clustering and δ is the cluster tolerance. If the mean of all the outputs of the primitive function, $\bar{y}$, for a given batch satisfies $|\bar{y} - a| < \delta$, the algorithm adds $\min(5, 0.1/\,|\bar{y} - a|)$ to the counter. These expressions increase with the severity of clustered data; the more closely the outputs are clustered, the higher the punishment term. The minimum term ensures that $\psi[f]$ is never infinite.

We also test for the approximation $\sin(x) \approx x$ by testing the inputs and outputs of $\overline{\text{the sine primitive function. If the inputs and outputs}}$ x and y of the sine primitive satisfy $|y - x| < 0.1$, the algorithm adds $\min(5, 0.05/|y - x|)$ to the counter. In the future, we plan to consider more approximations similar to the small angle approximation.

$\psi[f]$ should not artificially punish functions involving the primitives listed in Table 3.4 that are not trivial approximations because no proper use of these primitive functions will always produce outputs very close to the clustering points. Because $\psi[f]$ flags functions based on their batch outputs, each batch will likely give different outcomes. This allows $\psi[f]$ to better discriminate between trivial function approximations and nontrivial operations: $\psi[f]$ should flag trivial function approximations often, but it should only flag nontrivial operations rarely when the inputs statistically fluctuate to produce clustered outputs. In

Table 3.4: Primitive functions tested for clustering

Primitive Function	Cluster Points	Cluster Tolerance
$(\cdot)^2$	$\{0\}$	0.25
$(\cdot)^3$	$\{0\}$	0.25
$\sin(\cdot)$	$\{1, -1\}$	0.25
$\cos(\cdot)$	$\{1, -1\}$	0.25
$(\cdot)^c$	$\{1\}$	0.5

practice, we find that a weight of $w_\psi \approx 0.3$ for $\psi[f]$ is most effective in our loss function.

The $\xi[f]$ Regularization Term: When our network converges to the correct solution, it may converge to a more complicated expression equivalent to the desired expression. To promote simpler expressions, we slightly punish functions based on their complexity. The $\xi[f]$ term counts the number of activation functions used to produce f, which serves as a measure of f's complexity. We find that a small weight of $w_\xi \approx 0.1$ for $\xi[f]$ is most effective in our loss function. This small value has little significance when distinguishing between a function that fits a dataset well and a function that does not, but it is enough to promote simpler functions over complex functions when they have approximately the same loss otherwise.

The $\gamma[f]$ Regularization Term: The $\gamma[f]$ term also punishes functions for their complexity. The $\gamma[f]$ term counts the number of constants in f, which, like the number of activation functions, serves as a metric for f's complexity. We find that a weight of $w_\gamma \approx 0.15$ for $\gamma[f]$ is most effective in our loss function. Just as with $\xi[f]$, this small value has little significance when distinguishing between a function that fits a dataset well and a function that does not, but it is enough to slightly promote simpler functions over complex functions when they are otherwise equivalent. We weight $\gamma[f]$ slightly higher than $\xi[f]$ because many functions with constants can be simplified.

Functions with Undefined Outputs

One difficulty that may arise when training OccamNet is that many sampled functions are undefined on the input data range. Two cases of undefined functions are: 1) the function is undefined on part of the input data range for all values of a set of constants, or 2) the function is only undefined when the function's constants take on certain values. An example function satisfying case 1 is $f_1(x_0) = c_0/(x_0 - x_0)$, which divides by 0 regardless of the value of c_0. An example function satisfying case 2 is $f_2(x_0) = x_0^{c_0}$, which is undefined whenever x_0 is negative and c_0 is not an integer.

In the first case, the network should abandon the function. In the second case, the network should try other values for the constants. However, the network cannot easily determine which case an undefined function satisfies. To balance both cases, if the network obtains an undefined result, such as `NaN` or `inf`, for the forward pass, the network tests up to 100 more randomized sets of constants. If none of these attempts produce defined results, the network returns the array of undefined outputs. For example, with $c_0/(x_0 - x_0)$, the network tests a first set of constants, determines that they produce an undefined output, and tests

100 more constants. None of these functions are defined on all inputs, so the network returns the undefined outputs.

In contrast, with $x_0^{c_0}$, the network might find that the first set of constants produces undefined outputs, but after 20 retries, the network might discover that $c_0 = 2$ produces a function defined on all inputs. The network will then perform gradient descent and return the fitted value of c_0. Further, if at any point in the gradient descent, the forward pass yields undefined results, the network returns the well-defined constants and associated output from the previous forward pass. For example, for $x_0^{c_0}$, after the network discovers that $c_0 = 2$ works, the gradient for the constants will be undefined because c_0 can only be an integer. Thus, the network will return the outputs of $x_0^{c_0}$, for $c_0 = 2$, before the undefined gradients.

We find that if the network simply ignores functions with undefined outputs, these functions will increase in probability because our network regularization punishes many other functions. Since these punished functions decrease in probability during training, the functions with undefined outputs begin to increase in probability. To combat this, instead of ignoring undefined functions, we use a modified fitness for undefined functions, $K_i'(\mathcal{M}, f) = -w_{\text{undef}}s$, where w_{undef} is a hyperparameter that can be tuned. This punishes undefined functions, causing their weights to decrease. In practice, we find that a value of w_{undef} between 0 and 1 is most effective, depending on the application. Larger penalties may overly decrease probabilities of valid functions which are similar to an undefined function.

OccamNet with Units

Although we do not use this feature in our experiments, we also allow users to provide units for inputs and outputs. OccamNet will then regularize its functions so that they preserve the desired units.

To determine if a function f preserves units, we first encode the units of each input and output. We encode an input parameter's units as a NumPy array in which each entry represents the power of a given base unit. For example, if for a problem the relevant units are kg, m, and s, and we have an input F with units $\text{kg} \cdot \text{m/s}^2$, we would represent F's units as $[1, 1, -2]$.

We then feed these units through the sampled function. Each primitive function receives a set of variables with units, may have requirements on those units for them to be consistent, and returns a new set of units. For example, $\sin(\cdot)$ receives one variable which it requires to have units $[0, \ldots, 0]$ and returns the units $[0, \ldots, 0]$. Similarly, $+(\cdot, \cdot)$ takes two variables with units that it requires to be equal and returns the same units. Using these rules, we propagate units through the function until we obtain units for the output. If at any point the input units for a primitive function do not meet that primitive's requirements, that primitive returns $[\infty, \ldots, \infty]$. Any primitive functions that receive $[\infty, \ldots, \infty]$ also return $[\infty, \ldots, \infty]$. Finally, if the output units of f do not match the desired output units (including if f outputs $[\infty, \ldots, \infty]$), we mark f as not preserving units.

For the multiplication by a constant primitive function, $\cdot c$, we have to be careful. Because the units of c are unspecified, this primitive function can produce any output units. As such, it returns $[\text{NaN}, \ldots, \text{NaN}]$. If any primitive function receives $[\text{NaN}, \ldots, \text{NaN}]$, it will either return $[\text{NaN}, \ldots, \text{NaN}]$ if it has no constraints on the input units, or it will treat the $[\text{NaN}, \ldots, \text{NaN}]$ as being the units required to meet the primitive function's consistency conditions. For example, if the $\sin(\cdot)$ function receives $[\mathrm{NaN}, \dots, \mathrm{NaN}]$, it will treat the input as $[0, \dots, 0]$, and if the $+(\cdot, \cdot)$ function receives $[1, 2, 3]$ and $[\mathrm{NaN}, \mathrm{NaN}, \mathrm{NaN}]$, it will return $[1, 2, 3]$.

After sampling functions from OccamNet, we determine which functions do not preserve units. Because we wish to avoid these functions entirely, we bypass evaluating their normal fitness (thereby saving compute time) and instead assign a fitness of $K_i'(\mathcal{M}, f) = -w_{\text{units}} s$, where $s = n(\mathcal{M})/\sqrt{2\pi\sigma^2}$ is the maximum possible value of $K_i(\mathcal{M}, f)$ and w_{units} is a hyper-parameter that can be tuned (set to 1 by default).

3.6.2 Experimental Setup

We divide this section as follows:

1. Section 3.6.2 describes the hyperparameters used for the experiments described in the main text testing OccamNet on Analytic Functions, Non-Analytic Functions, Implicit Functions, and Image Recognition.

2. Section 3.6.2 describes the experimental setup for our tests with the PMLB datasets.

Experimental Setup and Hyperparameters for Non-PMLB Experiments

For the non-PMLB experiments, we terminate learning when the top-$v\lambda$ sampled functions all return the same fitness $K(\cdot, f)$ for 30 consecutive epochs. If this happens, these samples are equivalent function expressions.

Computing the most likely DAG allows retrieval of the final expression. If this final expression matches the correct function, we determine that the network has converged. For pattern recognition, there is no correct target composition, so we measure the accuracy of the classification rule on a test split, as is conventional. Note that in the experiments where $E = 0$, we instead take an approximate of the highest probability function by taking the argmax of the weights into each argument node.

In all experiments, if termination is not met in a set number of steps, we consider it as not converged. We also keep a constant temperature for all the layers except for the last one. An increased last layer temperature allows the network to explore higher function compositionality, as shallow layers can be further trained before the last layer probabilities become concentrated; this is particularly useful for learning functions with high degrees of nesting. More details on hyperparameters for experiments are in the SM. Our network converges rapidly, often in only a few seconds and at most a few minutes.

In Tables 3.5 and 3.6, we present and detail the hyperparameters we used for our experiments in the main paper. Note that detail about the setup for each experiment is provided in the open source repositories available at https://github.com/druidowm/OccamNet_Versions.

In Tables 3.5 and 3.6, $+$ is addition (2 arguments); $-$ is subtraction (2 arguments) $\cdot$ is multiplication (2 arguments); $/$ is division (2 arguments); $\sin(\cdot)$ is sine, $+c$ is addition of a constant, $\cdot c$ is multiplication of a constant, $(\cdot)^c$ is raising to the power of a constant, $\leq$ is an if-statement (4 arguments: comparing two numbers, one return for a true statement, and one for a false statement); $-(\cdot)$ is negation. MIN, MAX, and XOR all have two arguments. Here,

$\mathsf{SIGMOID}'$ is a sigmoid layer, and $\tanh'$ is a tanh layer where the inputs to both functions are scaled by a factor of 10, $+_4$, and $+_9$ are the operations of adding 4 and 9 numbers respectively, and MAX_4, MIN_4, MAX_9 and MIN_9 are defined likewise. The primitives for pattern recognition experiments are given as follows: $\mathbf{\Phi}_A$ consists of $\mathsf{SIGMOID}'$, $\mathsf{SIGMOID}'$, $\tanh'$, $\tanh'$, $+_4$, $+_4$, $+_9$, $+_9$, $+$, $+$, MIN, MIN, MAX and MAX; $\mathbf{\Phi}_B$ consists of id, id, id, id, $+$, $+$, $+$, $+_4$, $+_4$, $+_9$, $+_9$, $+_9$, tanh,, tanh, $\mathsf{SIGMOID}$, and $\mathsf{SIGMOID}$. Additionally, the constants used for pattern recognition are $\mathbf{C} = \{-1, -1, 0, 0, 1, 1, 1\}$.

In Tables 3.5 and 3.6, L is the depth, T is the temperature, T_{last} is the temperature of the final layer, σ is the variance, R is the sample size, λ is the fraction of best fits, α is the learning rate, E is the initialization parameter described in Section 3.6.1, and w_ϕ, w_ψ, w_ξ, and w_γ are as defined in Appendix 3.6.1. Table 3.5 does not include E as a listed hyperparameter because for all experiments listed, $E = 0$. With * we denote the experiments for which the best model is without skip connections. We do not regularize for any experiments in Table 3.5. NA entries mean that the corresponding hyperparameter is not present in the experiment. Note that the first three equations in Table 3.6 are not discussed in the main text. Instead, they are smaller experiments that we performed and which we discuss in the SM.

For all experiments in Table 3.6, we use a learning rate of 0.01 and, when applicable, a constant-learning rate of 0.05. We also set the temperature to 1 and the final layer temperature to 10 for all experiments in the table. For the equation $m_1v_1 - m_2v_2 = 0$, we sample m_1, v_1, and m_2 from $[-10, 10]$ and compute v_2 using the implicit function.

All our experiments in Table 3.5 use a batch size of 1000, except for *Backprop OccamNet* and *Finetune ResNet*, for which we use batch size 128. All our experiments in Table 3.6 use a batch size of 200. For each of our pattern recognition experiments, we use a 90%/10% train/test random split for the corresponding datasets. The input pixels are normalized to be in the range [0, 1]. During validation, for *MNIST Binary*, *MNIST Trinary* and *ImageNet Binary* the outputs of OccamNet are thresholded at 0.5. If the output matches the one-hot label, then the prediction is accurate; otherwise, it is inaccurate. For *Backprop OccamNet* and *Finetune ResNet* the outputs of OccamNet are viewed as the logits of a negative log likelihood loss function, so the prediction is the argmax of the logits. Backprop OccamNet and Finetune ResNet use an exponential decay of the learning rate with decay factor 0.999.

PMLB Experiments Setup

As described in the main text, we test OccamNet on 15 datasets from the Penn Machine Learning Benchmarks (PMLB) repository [101]. The 15 datasets chosen and the corresponding numbers we use to reference them, are shown in Table 3.7. We chose these datasets by selecting the first 15 regression datasets with fewer than 1667 datapoints. These 15 datasets are the only datasets from PMLB we examine.

We test four methods on these datasets. OccamNet-CPU, OccamNet-GPU, Eplex, AIF, and Extreme Gradient Boosting (XGB) [106]. We have described all of these methods except for XGB in the main text. XGB is a tree-based method that was identified by [107] as the best machine learning method based on validation MSE for modeling the PMLB datasets. However, XGB is not interpretable and thus cannot be used as a one-to-one comparison with OccamNet. Hence, although we provide the raw data for XGB's performance, we

Table 3.5: Hyperparameters for Experiments Where $E = 0$

Target	Primitives	Constants	Range	$L/\ T/\ T_{\text{last}}/\ \sigma$	$R/\ \lambda/\ \alpha$
Analytic Functions					
$2x^2 + 3x$	$(\cdot, \cdot, +, +)$	$\emptyset$	$[-10, 10]$	2/1/1/0.01	50/5/0.05
$\sin(3x + 2)$	$(\cdot, \sin, \sin, +, +)$	1, 2	$[-10, 10]$	3/1/1/0.001	50/5/0.005
$\sum_{n=1}^{3} \sin(nx)$	$(\sin, \sin, +, +, +)$	1, 2	$[-20, 20]$	5/1/1/0.001	50/5/0.005
$(x^2 + x)/(x + 2)$	$(\cdot, \cdot, +, +, /, /)$	1	$[-6, 6]$	2/1/2/0.0001	100/5/0.005
$x_0^2(x_0 + 1)/x_1^5$	$(\cdot, \cdot, +, +, /, /)$	1	$[[-10, 10], [0.1, 3]]$	4/1/3/0.0001	100/10/0.002
$x_0^2/2 + (x_1 + 1)^2/2$	$(\cdot, \cdot, +, +, /)$	1, 2	$[[-20, -2], [2, 20]]$	3/1/2/0.1	150/5/0.005
Program Functions					
$3x$ if $x > 0$, else x	$(\leq, \leq, \cdot, +, +, /)$	1	$[-20, 20]$	2/1/1.5/0.1	100/5/0.005
x^2 if $x > 0$, else $-x$	$(\leq, \leq, -(\cdot), +, +, -, \cdot)$	1	$[-20, 20]$	2/1/1.5/0.1	100/5/0.005
x if $x > 0$, else $\sin(x)$	$(\leq, \leq, +, +, \sin, \sin)$	1	$[-20, 20]$	3/1/1.5/0.01	100/5/0.005
$\mathsf{SORT}(x_0, x_1, x_2)$	$(\leq, +, \mathsf{MIN}, \mathsf{MAX}, \mathsf{MAX}/, \cdot, -)$	1, 2	$[-50, 50]^4$	3/1/4/0.01	100/5/0.004
$\mathsf{4LFSR}(x_0, x_1, x_2, x_3)$	$(+, +, \mathsf{XOR}, \mathsf{XOR})$	$\emptyset$	$\{0, 1\}^4$	2/1/1/0.1	100/5/0.005
$y_0(\vec{x}) = x_1$ if $x_0 < 2$, else $-x_1$ $y_1(\vec{x}) = x_0$ if $x_1 < 0$, else x_1^2	$(\leq, \leq, -(\cdot), \cdot)$	1, 2	$[-5, 5]^2$	3/1/3/0.01	100/5/0.002
$g(x) = x^2$ if $x < 2$, else $x/2$ $y(x) = g^{\circ 4}(x)$	$(\leq, \leq, +, \cdot, \cdot, /, /)$	1, 2	$[-8, 8]$	2/1/2/0.01	100/5/0.005
$g(x) = x + 2$ if $x < 2$, else $x - 1$ $y(x) = g^{\circ 2}(x)$	$(\leq, \leq, +, +, +, -, -)$	1, 2	$[-3, 6]$	2/1/1.5/0.01	100/5/0.005
Pattern Recognition					
MNIST Binary	$\mathbf{\Phi}_A$	$\mathbf{C}$	$[0, 1]^{784}$	2/1/10/0.01	150/ 10/0.05
MNIST Trinary	$\mathbf{\Phi}_A$	$\mathbf{C}$	$[0, 1]^{784}$	2/1/10/0.01	150/ 10/0.05
ImageNet Binary*	$\mathbf{\Phi}_A$	$\mathbf{C}$	$[0, 1]^{2048}$	4/1/10/10	150/10/0.0005
Backprop OccamNet*	$\mathbf{\Phi}_B$	$\mathbf{C}$	$[0, 1]^{2048}$	4/1/10/NA	NA/NA/0.1
Finetune ResNet*	$\mathbf{\Phi}_B$	$\mathbf{C}$	$[0, 1]^{3\times 224\times 224}$	4/1/10/NA	NA/NA/0.1

do not analyze it further. We train all methods except OccamNet-GPU on a single core of an Intel Xeon E5-2603 v4 @ 1.70GHz. For all methods, we use the primitive set $\Phi = (+(\cdot,\cdot), -(\cdot,\cdot), \times(\cdot,\cdot), \div(\cdot,\cdot), \sin(\cdot), \cos(\cdot), \exp(\cdot), \log|\cdot|)$.

For each dataset, we perform grid search to identify the best hyperparameters. The hyperparameters searched for the two OccamNet runs are shown in Table 3.8. The other hyperparameters not used in the grid search are set as follows: $T = 10$, $T_{\text{last}} = 10$, $w_\phi = w_\psi = w_\xi = w_\gamma = 0$, and the dataset batch size is the size of the training data. For OccamNet-GPU, we set R to be approximately as large as can fit on the V100 GPU, which varies between datasets. See Table 3.9 for the exact number of functions tested for each dataset for OccamNet-GPU. For XGBoost, we use exactly the same hyperparameter grid as used in

Table 3.6: Hyperparameters for Experiments Where $E = 1$

Target	Primitives	Constants	Range	L	σ	R	λ	$w_\phi/w_\psi/w_\xi/w_\gamma$
Analytic Functions								
$10.5x^3.1$	$(+, -, \cdot, /, \sin, \cos, +c, \cdot c, (\cdot)^c)$	$\emptyset$	$[0, 1]$	2	0.0005	200	10	0/0/0/0
$\cos(x)$	$(+, /, \sin)$	$2, \pi$	$[-100, 100]$	3	0.01	400	50	0/0/0/0
e^x	$(+, \cdot c, (\cdot)^c)$	10	$[0, 1]$	3	0.05	200	1	0.7/0.3/0.05/0.03
Implicit Functions								
$x_0 x_1 = 1$	$(+, -, \cdot, /, \sin, \cos)$	$\emptyset$	$[-1, 1]$	2	0.01	400	1	0.7/0.3/0.15/0.1
$x_0/x_1 = 1$	$(+, -, \cdot, /, \sin, \cos)$	$\emptyset$	$[-1, 1]$	2	0.01	400	1	0.7/0.3/0.15/0.1
$x_0^2 + x_1^2 = 1$	$(+, -, \cdot, /, \sin, \cos)$	$\emptyset$	$[-1, 1]$	2	0.01	200	10	0.7/0.3/0.15/0.1
$x_0/\cos(x_1) = 1$	$(+, -, \cdot, /, \sin, \cos)$	$\emptyset$	$[-1, 1]$	2	0.01	200	10	0.7/0.3/0.15/0.1
$m_1 v_1 - m_2 v_2 = 0$	$(+, -, \cdot, /, \sin, \cos)$	$\emptyset$	$[-10, 10]^3$	2	0.01	200	10	0.7/0.3/0.15/0.1

Table 3.7: Datasets Tested

#	Dataset	Size	# Features
1	1027_ESL	488	4
2	1028_SWD	1000	10
3	1029_LEV	1000	4
4	1030_ERA	1000	4
5	1089_USCrime	47	13
6	1096_FacultySalaries	50	4
7	192_vineyard	52	2
8	195_auto_price	159	15
9	207_autoPrice	159	15
10	210_cloud	108	5
11	228_elusage	55	2
12	229_pwLinear	200	10
13	230_machine_cpu	209	6
14	4544_GeographicalOriginalofMusic	1059	117
15	485_analcatdata_vehicle	48	4

[107]. For Eplex, we use the same hyperparameter grid as used in [107], with the exception that we use a depth of 4 to match that of OccamNet.

Table 3.8: OccamNet Hyperparameters

Hyperparameter	OccamNet-CPU	OccamNet-GPU
α	$\{0.5, 1\}$	$\{0.5, 1\}$
σ	$\{0.5, 1\}$	$\{0.1, 0.5, 1\}$
E	$\{1, 5\}$	$\{0, 1, 5\}$
λ/R	$\{0.1, 0.5, 0.9\}$	$\{0.1, 0.5, 0.9\}$
R	$\{500, 1000, 2000\}$	max
N	$1000000/R$	1000

We select the best run from the grid search as follows. For each hyperparameter combination, we first identify the models with the lowest training MSE and the lowest validation MSE:

- For OccamNet-CPU and OccamNet-GPU, we examine the highest probability function after each epoch. From these functions, we select the function with the lowest testing MSE and the function with the lowest validation MSE.

- For Eplex, we examine the highest-fitness individual from each generation. From these individuals, we select the individual with the lowest training MSE and the individual with the lowest validation MSE.

- For XGBoost, we train the model until the validation loss has not decreased for 100 epochs. We then return this model as the model with the best training MSE and validation MSE.

Table 3.9: Number of Functions Sampled Per Epoch

#	R
1	17123
2	8333
3	8333
4	8333
5	178571
6	166666
7	161290
8	52631
9	52631
10	78125
11	151515
12	41666
13	40000
14	7874
15	178571

Once we have the models with the lowest training and validation MSE for each hyperparameter combination, we identify the overall model with the lowest training MSE from the set of lowest training MSE models, and we identify the overall model with the lowest validation MSE from the set of lowest validation MSE models. We then record these models' training MSE and validation MSE as the best training MSE and validation MSE, respectively. Finally, we test the model with the overall lowest validation MSE on the testing dataset and record the result as the grid search testing MSE.

For our test of OccamNet and Eplex's scalability on the PMLB datasets, we use the same hyperparameter combinations as those listed described above, except that, as described in the main text, we run OccamNet-GPU with 250, 1000, 4000, 16000, and 64000 functions sampled per epoch and Eplex with 250, 500, 1000, 2000 and 4000 functions sampled per epoch. Our evaluation of training, validation, and testing loss is exactly the same as described above, except that we evaluate the lowest losses for each value of N instead of grouping N with all of the other hyperparameters.

Chapter 4

Conclusion

Given ML's success, it is natural to ask in what ways it may be a valuable tool for physics. This thesis explores two approaches that can enable more effective ML techniques for physics: 1) fast and memory-efficient simulation and 2) discovering new physics. These two directions reflect problems where classical techniques display shortcomings and machine-learning models have the potential to overcome these shortcomings. We develop a physics-optimized ML model for each approach to illustrate the potential of ML to advance physics through each.

Regarding our first approach, existing simulation techniques often struggle to scale to large systems because of poor scaling of memory and compute [11]. For a quantum system, this is especially problematic because the size of the state of the system grows exponentially with the number of components (particles, potential wells, etc.) [11]. The inability to simulate large quantum systems limits the progress in fields such as quantum computing and quantum engineering [11]. We discussed Q-Flow, a technique for bosonic quantum simulation using normalizing flows and a novel time evolution algorithm to simulate a compressed representation of a quantum state [11]. By time evolving the compressed neural representation of the state, we avoid the need to store and process the full quantum state and enable simulating higher-dimensional quantum systems than is possible using standard finite-difference or finite-element solvers [11].

Regarding our second approach, scientific discovery is difficult to perform algorithmically because of its open-ended nature and the enormous search-space of possible theories [13]. Instead, it is usually performed by hand, relying on human intuition and tedious trial and error [13]. We beleive this problem naturally lends itself to ML techniques, because they have proven successful in high-dimensional search problems [12] and can be instilled with human-like intuition [3]. We discussed OccamNet, a novel architecture and set of algorithms for scientific discovery using efficient and parallelizable symbolic regression [13]. By using reinforcement learning and symbolic inductive biases, OccamNet intelligently searches through the space of possible equations describing data, a step toward automated physics discovery [13].

Our methods demonstrate the potential for ML as a valuable tool for physics research. In the future, we hope to extend this research by incorporating further inductive biases from math and physics and applying our methods to real-world problems in physics.

Appendix A

Q-Flow Appendices

The below is taken from the appendices of [11].

A.1 Q Function Conversions

A.1.1 Quantum Preliminaries

Here, we provide a brief and intuitive introduction to the theory of bosonic systems. We intentionally simplify most of the definitions and focus on the important concepts to our study. For an in-depth discussion, please refer to [108].

We will discuss a few important terms that we use throughout the main text.

Braket notation. Such notation is used throughout the text to denote quantum states. Quantum states are elements of a complex vector space V, equipped with a Hermitian form. In our work we use the standard Hermitian inner product, which in math notation is $(\mathbf{v}, \mathbf{w}) = \mathbf{v}^\dagger \mathbf{w}$. Here $\dagger$ denotes the complex conjugate, for any two vectors $\mathbf{v}, \mathbf{w} \in V$. In physics notation, we write $\mathbf{v}$ as $|\mathbf{v}\rangle$ (known as a *ket*) and likewise for $\mathbf{w}$. We also use the notation $\langle \mathbf{v}| \equiv \mathbf{v}^\dagger$, and call this a *bra*. Then, $\mathbf{v}^\dagger \mathbf{w}$ can be written $\langle \mathbf{v}| |\mathbf{w}\rangle$, or more concisely as $\langle \mathbf{v}|\mathbf{w}\rangle$. Furthermore, $|\mathbf{v}\rangle \langle \mathbf{w}|$ denotes the outer product of $\mathbf{v}$ and $\mathbf{w}^\dagger$.

Vacuum states, the Fock space. In addition to using $|\cdot\rangle$ and $\langle\cdot|$ to denote arbitrary members of the Hilbert space and it's dual. We also have special notation for a few members of the Hilbert space. It turns out that the Hilbert space for a single Well is spanned by a countably infinite set of basis vectors, which we label $|0\rangle, |1\rangle, \ldots$. For multiple Wells, the total Hilbert space will be the tensor product of each particle's Hilbert space. To represent a general element of $|0\rangle, |1\rangle, \ldots$, we will use a Roman letter inside the ket or bra.

Creation, annihilation operators and Coherent state. The creation operator $a^\dagger$ satisfies $a^\dagger |n\rangle = \sqrt{n+1}\,|n+1\rangle$ and the annihilation operator a satisfies $a|n\rangle = \sqrt{n}\,|n-1\rangle$ with $a|0\rangle = 0$. The coherent state $|\alpha\rangle$ with a complex number α is defined as $|\alpha\rangle = e^{\alpha a^\dagger - \alpha^* a}|0\rangle$, where e should be interpreted as matrix exponential function.

Compute observables. In quantum mechanics, a density matrix ρ can be expressed as $\rho = \sum_{n,m} \rho_{n,m} \left|n\right\rangle \left\langle m\right|$ and an observable O can be expressed as $O = \sum_{n,m} O_{n,m} \left|n\right\rangle \left\langle m\right|$, where both ρ and O can be viewed as Hermitian matrices. It follows that the expectation value of the observable $\langle O\rangle = \mathrm{tr}(\rho O) = \sum_{n,m} O_{n,m}\rho_{m,n}$.

A.1.2 Q Function to ρ

In this section, we show that for a given $Q(\alpha, \alpha^*)$, the density matrix ρ corresponding to it is given by

$$\left\langle m\right| \rho \left|n\right\rangle = \pi\sqrt{m!n!} \sum_{k=0}^{\min(m,n)} \frac{Q_{m-k,n-k}}{k!}, \tag{A.1}$$

where

$$Q_{a,b}(\alpha,\alpha^*) = \frac{1}{a!b!}\,\frac{\partial^{a+b}}{\partial^a\alpha\,\partial^b\alpha^*} Q(\alpha,\alpha^*)\bigg|_{\alpha=\alpha^*=0}.$$

From expressing $\left\langle\alpha\right|$ and $\left|\alpha\right\rangle$ in terms of Harmonic Oscillator eigenstates, we have that

$$\begin{aligned}
Q(\alpha,\alpha^*) &= \frac{1}{\pi}e^{-\alpha\alpha^*}\sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\frac{\left\langle m\right|\rho\left|n\right\rangle}{\sqrt{m!n!}}\alpha^{*m}\alpha^n \\
&= \frac{1}{\pi}\left(\sum_{s=0}^{\infty}\frac{(-1)^s}{s!}(\alpha\alpha^*)^s\right)\sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\frac{\left\langle m\right|\rho\left|n\right\rangle}{\sqrt{m!n!}}\alpha^{*m}\alpha^n \\
&= \frac{1}{\pi}\sum_{s=0}^{\infty}\sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\frac{(-1)^s}{s!}\frac{\left\langle m\right|\rho\left|n\right\rangle}{\sqrt{m!n!}}\left(\alpha^*\right)^{m+s}\alpha^{n+s}
\end{aligned}$$

To determine $\left\langle m\right|\rho\left|n\right\rangle$, we must thus invert this series. However, since we know the correct form, we can simply substitute Equation A.1 into the expression above and show that it correctly gives $Q(\alpha,\alpha^*)$:

$$\begin{aligned}
&\frac{1}{\pi}\sum_{s=0}^{\infty}\sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\frac{(-1)^s}{s!}\frac{\left\langle m\right|\rho\left|n\right\rangle}{\sqrt{m!n!}}\left(\alpha^*\right)^{m+s}\alpha^{n+s} \\
=&\frac{1}{\pi}\sum_{s=0}^{\infty}\sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\frac{(-1)^s}{s!}\frac{\pi\sqrt{m!n!}\sum_{k=0}^{\min(m,n)}\frac{Q_{m-k,n-k}}{k!}}{\sqrt{m!n!}}\left(\alpha^*\right)^{m+s}\alpha^{n+s} \\
=&\sum_{s=0}^{\infty}\sum_{m=0}^{\infty}\sum_{n=0}^{\infty}\sum_{k=0}^{\min(m,n)}\frac{(-1)^s}{s!}\frac{Q_{m-k,n-k}}{k!}\left(\alpha^*\right)^{m+s}\alpha^{n+s}.
\end{aligned}$$

Setting $a = m+s$ and $b = n+s$ gives

$$\sum_{a=0}^{\infty}\sum_{b=0}^{\infty}\sum_{s=0}^{\min(a,b)}\sum_{k=0}^{\min(a,b)-s}\frac{(-1)^s}{s!}\frac{Q_{a-k-s,b-k-s}}{k!}\left(\alpha^*\right)^a\alpha^b.$$

Then, setting $d = s + k$ gives

$$\begin{aligned}
&\sum_{a=0}^{\infty}\sum_{b=0}^{\infty}\sum_{d=0}^{\min(a,b)}\sum_{s=0}^{d}\frac{(-1)^s}{s!}\frac{Q_{a-d,b-d}}{(d-s)!}\left(\alpha^*\right)^a\alpha^b \\
=&\sum_{a=0}^{\infty}\sum_{b=0}^{\infty}\left(\alpha^*\right)^a\alpha^b\sum_{d=0}^{\min(a,b)}Q_{a-d,b-d}\sum_{s=0}^{d}(-1)^s\binom{d}{s}.
\end{aligned}$$

Now, by the Binomial Theorem, $\sum_{s=0}^{d}(-1)^s\binom{d}{s} = (1-1)^d = 0^d$, which is 0 unless $d = 1$. So, we get

$$\begin{aligned}
&\sum_{a=0}^{\infty}\sum_{b=0}^{\infty}Q_{a,b}(\alpha,\alpha^*)\cdot\left(\alpha^*\right)^a\alpha^b \\
=&\sum_{a=0}^{\infty}\sum_{b=0}^{\infty}\frac{\left(\alpha^*\right)^a\alpha^b}{a!b!}\frac{\partial^{a+b}}{\partial^a\alpha\partial^b\alpha^*}Q(\alpha,\alpha^*) \\
=&Q(\alpha,\alpha^*),
\end{aligned}$$

as desired. The last step comes from the Taylor series representation of Q, which is only valid if Q is analytic. So as long as Q is analytic, this result holds.

Example. To convert a Liouvillian to an equation of motion for Q, we use the fact that

$$\dot{\rho} = (a^\dagger)^j a^k \rho (a^\dagger)^l a^m, \tag{A.2}$$

corresponds to

$$\dot{Q} = (\alpha^*)^j\left(\alpha + \frac{\partial}{\partial\alpha^*}\right)^k\alpha^m\left(\alpha^* + \frac{\partial}{\partial\alpha}\right)^l Q. \tag{A.3}$$

A linear combination of terms of the form in Equation A.2 corresponds to the same linear combination of the corresponding terms in Equation A.3.

A.1.3 Coherent State Identities

$$\begin{aligned}
a^\dagger |\alpha\rangle &= a^\dagger e^{-|\alpha|^2/2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \\
&= e^{-|\alpha|^2/2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} \sqrt{n+1}\, |n+1\rangle \\
&= e^{-|\alpha|^2/2} \frac{\partial}{\partial\alpha} \sum_{n=0}^{\infty} \frac{\alpha^{n+1}}{\sqrt{(n+1)!}} |n+1\rangle \\
&= e^{-|\alpha|^2/2} \frac{\partial}{\partial\alpha} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \\
&= e^{-|\alpha|^2/2} \frac{\partial}{\partial\alpha} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \\
&= \frac{\partial}{\partial\alpha} e^{-|\alpha|^2/2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle - \left(\frac{\partial}{\partial\alpha} e^{-|\alpha|^2/2}\right) \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \\
&= \frac{\partial}{\partial\alpha} e^{-|\alpha|^2/2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle - \left(\frac{\partial}{\partial\alpha} e^{-|\alpha|^2/2}\right) \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \\
&= \frac{\partial}{\partial\alpha} e^{-|\alpha|^2/2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle + \frac{\alpha^*}{2} e^{-|\alpha|^2/2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \\
&= \left(\frac{\alpha^*}{2} + \frac{\partial}{\partial\alpha}\right) e^{-|\alpha|^2/2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle \\
&= \left(\frac{\alpha^*}{2} + \frac{\partial}{\partial\alpha}\right) |\alpha\rangle .
\end{aligned}$$

Similarly,

$$\langle\alpha| a = \left(\frac{\alpha}{2} + \frac{\partial}{\partial\alpha^*}\right) \langle\alpha| .$$

Also,

$$\frac{\partial}{\partial\alpha^*} |\alpha\rangle = \frac{\partial}{\partial\alpha^*} e^{-|\alpha|^2/2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle = -\frac{\alpha}{2} e^{-|\alpha|^2/2} \sum_{n=0}^{\infty} \frac{\alpha^n}{\sqrt{n!}} |n\rangle = -\frac{\alpha}{2} |\alpha\rangle \tag{A.4}$$

and

$$\frac{\partial}{\partial\alpha} \langle\alpha| = -\frac{\alpha^*}{2} \langle\alpha| .$$

A.1.4 $\boldsymbol{\rho}$ Evolution to Q Function Evolution

Note that

$$\begin{aligned}\langle\alpha| a^\dagger \hat{O}_1\rho\hat{O}_2 |\alpha\rangle &= \alpha^* \langle\alpha| \hat{O}_1\rho\hat{O}_2 |\alpha\rangle ,\\ \langle\alpha| \hat{O}_1\rho\hat{O}_2 a |\alpha\rangle &= \alpha \langle\alpha| \hat{O}_1\rho\hat{O}_2 |\alpha\rangle .\end{aligned}$$

Also,

$$\begin{aligned}\langle\alpha| a\hat{O}_1\rho\hat{O}_2 |\alpha\rangle &= \left[\left(\frac{\alpha}{2}+\frac{\partial}{\partial\alpha^*}\right)\langle\alpha|\right]\hat{O}_1\rho\hat{O}_2 |\alpha\rangle\\ &= \left(\frac{\alpha}{2}+\frac{\partial}{\partial\alpha^*}\right)\langle\alpha| \hat{O}_1\rho\hat{O}_2 |\alpha\rangle - \langle\alpha| \hat{O}_1\rho\hat{O}_2\frac{\partial}{\partial\alpha^*} |\alpha\rangle\\ &= \left(\frac{\alpha}{2}+\frac{\partial}{\partial\alpha^*}\right)\langle\alpha| \hat{O}_1\rho\hat{O}_2 |\alpha\rangle + \frac{\alpha}{2}\langle\alpha| \hat{O}_1\rho\hat{O}_2 |\alpha\rangle\\ &= \left(\alpha+\frac{\partial}{\partial\alpha^*}\right)\langle\alpha| \hat{O}_1\rho\hat{O}_2 |\alpha\rangle ,\end{aligned}$$

and

$$\begin{aligned}\langle\alpha| \hat{O}_1\rho\hat{O}_2 a^\dagger |\alpha\rangle &= \langle\alpha| \hat{O}_1\rho\hat{O}_2\left(\frac{\alpha^*}{2}+\frac{\partial}{\partial\alpha}\right)|\alpha\rangle\\ &= \left(\frac{\alpha^*}{2}+\frac{\partial}{\partial\alpha}\right)\langle\alpha| \hat{O}_1\rho\hat{O}_2 |\alpha\rangle - \left[\frac{\partial}{\partial\alpha}\langle\alpha|\right]\hat{O}_1\rho\hat{O}_2 |\alpha\rangle\\ &= \left(\frac{\alpha^*}{2}+\frac{\partial}{\partial\alpha}\right)\langle\alpha| \hat{O}_1\rho\hat{O}_2 |\alpha\rangle + \frac{\alpha^*}{2}\langle\alpha| \hat{O}_1\rho\hat{O}_2 |\alpha\rangle\\ &= \left(\alpha^*+\frac{\partial}{\partial\alpha}\right)\langle\alpha| \hat{O}_1\rho\hat{O}_2 |\alpha\rangle .\end{aligned}$$

With these results, we now have that for an equation of the form

$$\dot{\rho} = \sum_{j,k,l,m} c_{j,k,l,m}(a^\dagger)^j a^k \rho (a^\dagger)^l a^m,$$

we can convert the the Q function equation of motion by inserting $\frac{1}{\pi}\langle\alpha|\ \ |\alpha\rangle$ to get

$$\begin{aligned}
&\frac{1}{\pi}\langle\alpha|\,\dot{\rho}\,|\alpha\rangle = \frac{1}{\pi}\sum_{j,k,l,m} c_{j,k,l,m}\langle\alpha|\,(a^\dagger)^j a^k \rho (a^\dagger)^l a^m\,|\alpha\rangle \\
\implies &\dot{Q}(\alpha,\alpha^*) = \frac{1}{\pi}\sum_{j,k,l,m} c_{j,k,l,m}(\alpha^*)^j\langle\alpha|\,a^k\rho(a^\dagger)^l a^m\,|\alpha\rangle \\
\implies &\dot{Q}(\alpha,\alpha^*) = \frac{1}{\pi}\sum_{j,k,l,m} c_{j,k,l,m}(\alpha^*)^j\left(\alpha+\frac{\partial}{\partial\alpha^*}\right)^k\langle\alpha|\,\rho(a^\dagger)^l a^m\,|\alpha\rangle \\
\implies &\dot{Q}(\alpha,\alpha^*) = \frac{1}{\pi}\sum_{j,k,l,m} c_{j,k,l,m}(\alpha^*)^j\left(\alpha+\frac{\partial}{\partial\alpha^*}\right)^k\alpha^m\langle\alpha|\,\rho(a^\dagger)^l\,|\alpha\rangle \\
\implies &\dot{Q}(\alpha,\alpha^*) = \frac{1}{\pi}\sum_{j,k,l,m} c_{j,k,l,m}(\alpha^*)^j\left(\alpha+\frac{\partial}{\partial\alpha^*}\right)^k\alpha^m\left(\alpha^*+\frac{\partial}{\partial\alpha}\right)^l\langle\alpha|\,\rho\,|\alpha\rangle \\
\implies &\dot{Q}(\alpha,\alpha^*) = \sum_{j,k,l,m} c_{j,k,l,m}(\alpha^*)^j\left(\alpha+\frac{\partial}{\partial\alpha^*}\right)^k\alpha^m\left(\alpha^*+\frac{\partial}{\partial\alpha}\right)^l Q(\alpha,\alpha^*).
\end{aligned}$$

A.1.5 Observable calculation with respect to Q function

Consider a general observable $\hat{O}$. Its expected value given a density matrix ρ is

$$\langle\hat{O}\rangle = \mathrm{Tr}\Big(\hat{O}\rho\Big). \tag{A.5}$$

Inserting the coherent state resolution of the identity, we get that

$$\begin{aligned}
\mathrm{Tr}\Big(\hat{O}\rho\Big) &= \int\frac{d\alpha d\alpha^*}{\pi}\,\mathrm{Tr}\Big(\hat{O}\rho\,|\alpha\rangle\langle\alpha|\Big) \\
&= \int\frac{d\alpha d\alpha^*}{\pi}\,\langle\alpha|\,\hat{O}\rho\,|\alpha\rangle\,.
\end{aligned}$$

Depending on the operator, it may be most useful to insert the resolution of the identity elsewhere.

Example The expected value of $a^m(a^\dagger)^n$ given a density matrix ρ is

$$\begin{aligned}
\langle a^m\left(a^\dagger\right)^n\rangle &= \mathrm{Tr}\left(a^m\left(a^\dagger\right)^n\rho\right) \\
&= \int\frac{d\alpha d\alpha^*}{\pi}\,\mathrm{Tr}\big(a^m\,|\alpha\rangle\langle\alpha|\left(a^\dagger\right)^n\rho\big) \\
&= \int\frac{d\alpha d\alpha^*}{\pi}\,\langle\alpha|\left(a^\dagger\right)^n\rho a^m\,|\alpha\rangle \\
&= \int\frac{d\alpha d\alpha^*}{\pi}\,\alpha^m(\alpha^*)^n\langle\alpha|\,\rho\,|\alpha\rangle \\
&= \int dq dp\ (q+ip)^m(q-ip)^n Q(q,p).
\end{aligned}$$

If $m \neq n$, this is not an observable, but could be made an observable by adding its Hermitian conjugate.

A.2 Stochastic Euler-KL Method

Here we derive Equation 2.5 for the control-variance gradient of the KL-Divergence. We start with

$$KL(Q_\theta^{t+dt}||Q_\mathcal{L}^t) = \int Q_\theta^{t+dt} \ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t}.$$

Taking the gradient gives

$$\begin{aligned}
\nabla_\theta KL(Q_\theta^{t+dt}||Q_\mathcal{L}^t) &= \nabla_\theta \int Q_\theta^{t+dt} \ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t} \\
&= \int \left(\nabla_\theta Q_\theta^{t+dt}\right) \ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t} + \int Q_\theta^{t+dt} \nabla_\theta \ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t} \\
&= \int Q_\theta^{t+dt} \left(\nabla_\theta \ln Q_\theta^{t+dt}\right) \ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t} + \int Q_\theta^{t+dt} \nabla_\theta \ln Q_\theta^{t+dt} \\
&= \int Q_\theta^{t+dt} \left[\ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t} + 1\right] \nabla_\theta \ln Q_\theta^{t+dt}.
\end{aligned}$$

Now, note that

$$\int Q_\theta^{t+dt}(x) \nabla_\theta \ln Q_\theta^{t+dt}(x) = \int Q_\theta^{t+dt}(x) \frac{\nabla_\theta Q_\theta^{t+dt}(x)}{Q_\theta^{t+dt}(x)} = \int \nabla_\theta Q_\theta^{t+dt}(x) = \nabla_\theta \int Q_\theta^{t+dt}(x) = \nabla_\theta 1 = 0. \tag{A.6}$$

So, letting

$$b = \int Q_\theta^{t+dt} \ln \frac{Q_\theta^{t+dt}(x)}{Q_\mathcal{L}^t(x)} \approx \frac{1}{N} \sum_{x \sim Q_\theta^{t+dt}} \ln \frac{Q_\theta^{t+dt}(x)}{Q_\mathcal{L}^t(x)}, \tag{A.7}$$

we can subtract a control variance to get

$$\begin{aligned}
\nabla_\theta KL(Q_\theta^{t+dt}||Q_\mathcal{L}^t) &= \int Q_\theta^{t+dt} \left[\ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t} + 1\right] \nabla_\theta \ln Q_\theta^{t+dt} \\
&= \int Q_\theta^{t+dt} \left[\ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t} + 1\right] \nabla_\theta \ln Q_\theta^{t+dt} - (b+1) \int Q_\theta^{t+dt}(x) \nabla_\theta \ln Q_\theta^{t+dt}(x) \\
&= \int Q_\theta^{t+dt} \left[\ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t} - b\right] \nabla_\theta \ln Q_\theta^{t+dt} \\
&= \int Q_\theta^{t+dt} \left[\ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t} - b\right] \nabla_\theta \ln Q_\theta^{t+dt}
\end{aligned}$$

Finally, approximating the integral gives

$$\begin{aligned}
\nabla_\theta KL(Q_\theta^{t+dt}||Q_\mathcal{L}^t) &= \int Q_\theta^{t+dt} \left[\ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t} - b\right] \nabla_\theta \ln Q_\theta^{t+dt} \\
&\approx \frac{1}{N} \sum_{x \sim Q_\theta^{t+dt}} \left[\ln \frac{Q_\theta^{t+dt}}{Q_\mathcal{L}^t} - b\right] \nabla_\theta \ln Q_\theta^{t+dt},
\end{aligned}$$

as desired.

A.3 Additional Experimental Details

A.3.1 Pseudo-spectral and finite difference baseline details

As a baseline approach to solving the Q function evolution PDE (Eq. 2.2), we implement a pseudo-spectral and finite difference discretization of the PDE [42] in a square domain with $-10 < q_j < 10$ and $-10 < p_j < 10$ for each Well j, set $Q = 0$ at the boundaries, and integrate using an adaptive Tsitouras 5/4 Runge-Kutta solver (Tsit5) [61] while projecting at each time step to ensure the probability density Q remains positive and normalized. The psuedo-spectral method uses periodic boundary conditions and computes spatial derivatives using a fast Fourier transform. The finite difference method uses Dirichlet boundary conditions set at zero and computes spatial derivatives using the standard second-order finite difference stencil. We use a grid size of 256 grid points per dimension for 1-Well and 32 grid points per dimension for 2-Wells, resulting in a state size of $256^2 = 65{,}536$ for 1-Well and $32^4 = 1{,}048{,}576$ for 2-Wells. Note that for a fixed grid size, the state grows exponentially with the number of Wells—i.e. the curse of dimensionality. This limits our ability to perform more fine-grained simulations on larger domains and makes this baseline approach intractable for more than a few Wells.

A.3.2 PINN baseline details

We also use Physics Informed Neural Networks (PINNs) as a baseline. To implement this, we use the PINA library which is built on top of PyTorch. For each problem, we have two loss terms. The first computes the L_2 loss between the predicted initial distribution and the actual initial distribution for points sampled uniformly from within the domain of the solver at $t = 0$. The second computes the L_2 loss between the PINN time derivative and the predicted time derivative $\tilde{\mathcal{L}}Q$ at points sampled uniformly from within the spacial and temporal domain of the solver. The total loss is the sum of these two losses. We then optimize using gradient decent. Every 500 epochs we re-sample the points with which to compute the loss. For each experiment, we use a fully connected network with skip connections. The layer sizes are [input size, 40, 40, 40, 1]. The following are the hyperparameters used for each of the experiments:

- **1-Well Harmonic Oscillator:** We use 1000 samples at a time for the initial condition and 50000 samples at a time for the derivative condition. We train for 25000 epochs with a learning rate of 0.001.
- **2-Well Harmonic Oscillator:** We use 1000 samples at a time for the initial condition and 30000 samples at a time for the derivative condition. We train for 25000 epochs with a learning rate of 0.001.
- **20-Well Harmonic Oscillator:** We use 3000 samples at a time for the initial condition and 3000 samples at a time for the derivative condition. We train for 50000 epochs with a learning rate of 0.001. Here, we have to decrease the number of samples for the derivative condition because of memory limits.

- **2-Well Dissipative Bosonic Model:** We use 1000 samples at a time for the initial condition and 30000 samples at a time for the derivative condition. We train for 25000 epochs with a learning rate of 0.001.

A.3.3 Euler experiment details

Below are the hyperparameters we use for the Euler method. For the Harmonic Oscillator results we use a 3 layer RealNVP where each affine transformation is a 2 hidden layer feed forward neural network with hidden layers of size 5. For the Dissipative Bosonic Model result, we use a Convex Potential Flow with a 5 hidden layer input-convex neural network with hidden layers of size 20 and augmented layers of size 4, see [52].

- **1-Well Harmonic Oscillator:** We train for 1500 steps with a step size of 0.01. For each step, we use the KL control variance loss to fit for 150 epochs with a learning rate of 0.001. We use 1000 samples per fitting epoch.
- **2-Well Harmonic Oscillator:** We train for 1500 steps with a step size of 0.01. For each step, we use the KL control variance loss to fit for 150 epochs with a learning rate of 0.001. We use 1000 samples per fitting epoch.
- **20-Well Harmonic Oscillator:** We train for 1500 steps with a step size of 0.01. For each step, we use the KL control variance loss to fit for 150 epochs with a learning rate of 0.001. We use 10000 samples per fitting epoch.
- **2-Well Dissipative Bosonic Model:** We train for 400 steps with a step size of 0.02. For each step, we use the KL control variance loss to fit for 200 epochs with a learning rate of 0.002. We use 10000 samples per fitting epoch.

A.3.4 TDVP experiment details

Below are the hyperparameters we use for the TDVP method. For the Harmonic Oscillator results we use a 3 layer RealNVP where each affine transformation is a 2 hidden layer feed forward neural network with hidden layers of size 5. For the Dissipative Bosonic Model result, we use a Convex Potential Flow with a 5 hidden layer input-convex neural network with hidden layers of size 20 and augmented layers of size 4, see [52].

- **1-Well Harmonic Oscillator:** We train for 1500 steps with a step size of 0.01. We use 1000 samples per step. We use a diagonal shift of 0.01.
- **2-Well Harmonic Oscillator:** We train for 1500 steps with a step size of 0.01. We use 1000 samples per step. We use a diagonal shift of 0.01.
- **20-Well Harmonic Oscillator:** We train for 1500 steps with a step size of 0.01. We use 10000 samples per step. We use a diagonal shift of 0.01.
- **2-Well Dissipative Bosonic Model:** We train for 2000 steps with a step size of 0.004. We use 10000 samples per step. We use a diagonal shift of 0.01.

Appendix B

OccamNet Appendices

The below is taken from the appendices of [13].

We have organized the Supplemental Material as follows:

- In Section B.1 we provide the results of our experiments on the PMLB datasets.
- In Section B.2 we examine the fits each method provides for the PMLB Datasets.
- In Section B.3 analyze the results of the experiment scaling OccamNet-GPU on the PMLB datasets.
- In Section B.4 we present a series of ablation studies.
- In Section B.5 we discuss neural models for sorting and pattern recognition.
- In Section B.6 we discuss a few small experiments we tested.
- In Section B.7 we discuss research related to the various applications of OccamNet.
- In Section B.8 we discuss the evolutionary strategies for fitting functions and programs that we use as benchmarks.
- In Section B.9, we catalog our code and video files.

B.1 PMLB Experiment Results

The raw data for the PMLB experiments are shown in Table B.1. To improve readability, we use red highlighting and bold text to illustrate the best-performing model for each dataset and metric. We compare OccamNet-CPU, OccamNet-GPU, Eplex, and AIF, marking the method with the lowest MSE or training time in red. We also compare OccamNet-GPU, Eplex, and AIF, marking the method with the lowest MSE or training time in bold. Plots of the full results for the PMLB scaling experiment are shown in Figures B.1 and B.2. As discussed in Section B.3, Figure B.2 shows OccamNet-GPU's performance when only considering a restricted set of hyperparameters.

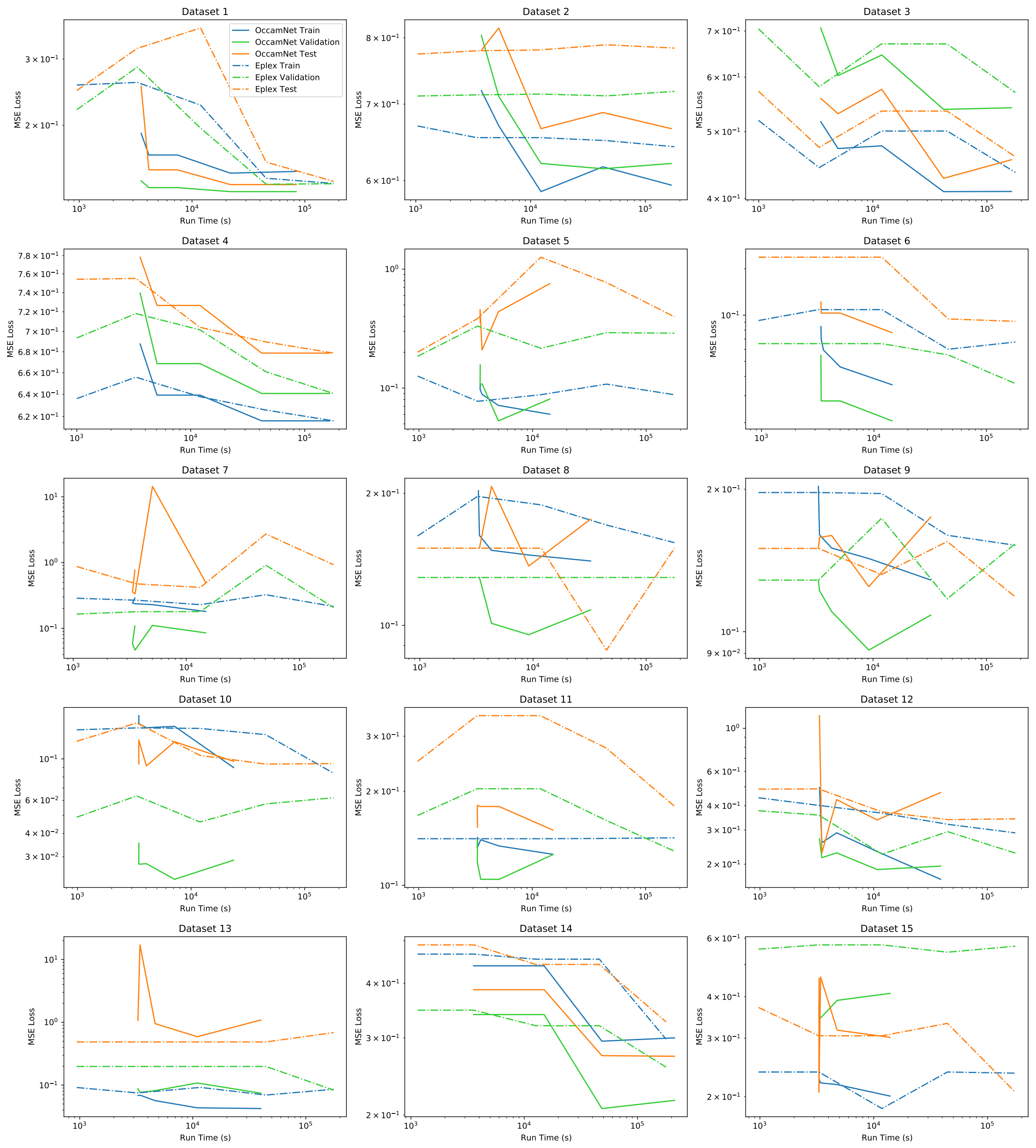


Figure B.1: OccamNet-GPU and Eplex's Training, Validation, and Testing MSE as a function of run time for the 15 PMLB datasets discussed above.

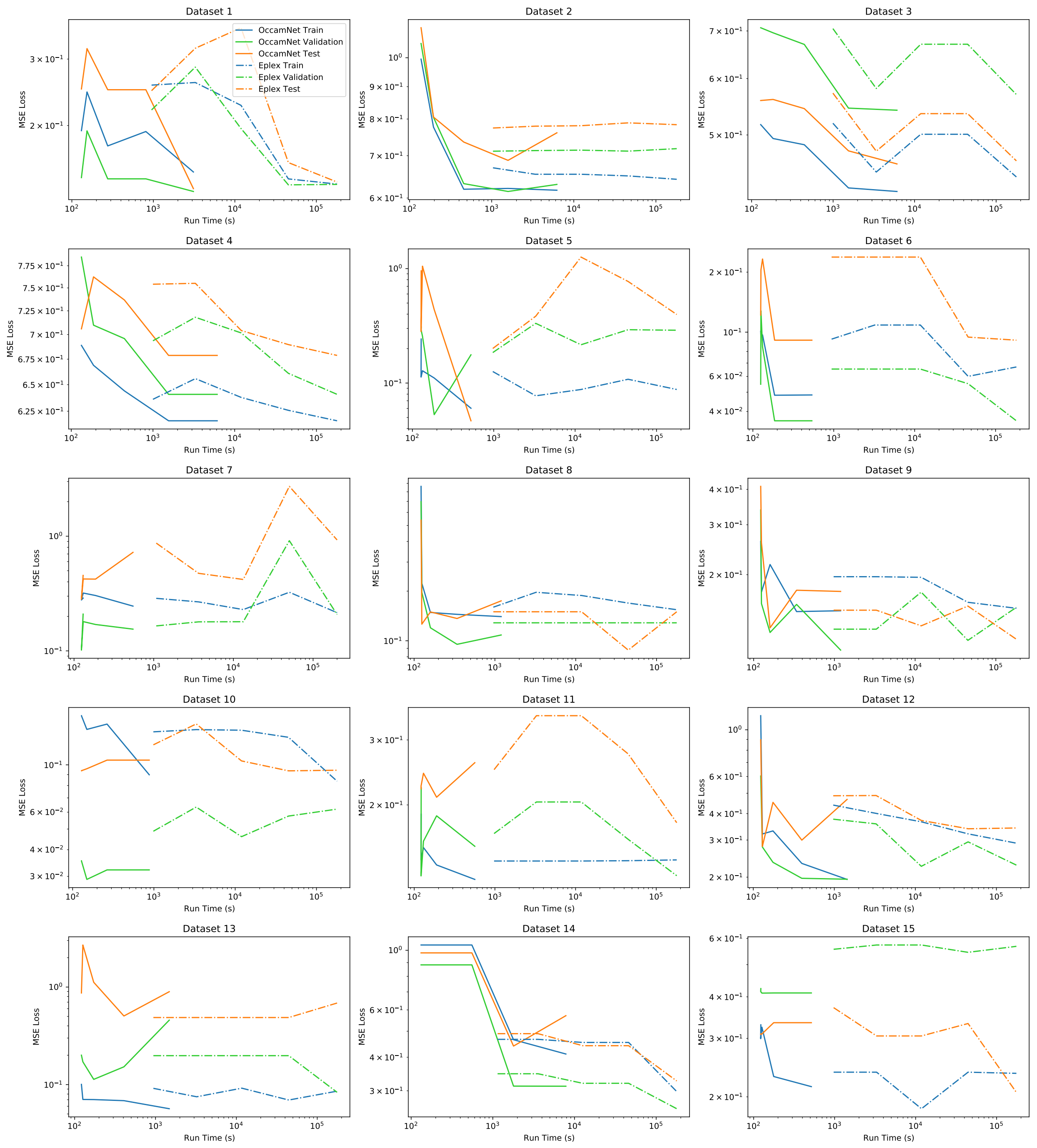


Figure B.2: OccamNet-GPU and Eplex's Training, Validation, and Testing MSE as a function of run time for the 15 PMLB datasets discussed above. For this figure, we only consider the losses for a restricted subset of hyperparameter combinations.

Table B.1: Raw data from the PMLB experiments. Hyperparameters and best fits are in the following path in our code (see Section B.9): `pmbl-experiments/pmlb-results`.

	Training Loss (MSE)					Validation Loss (MSE)				
#	OccamNet-CPU	OccamNet-GPU	Eplex	AIF	XGB	OccamNet-CPU	OccamNet-GPU	Eplex	AIF	XGB
1	0.177	**0.139**	0.153	0.465	0.056	0.141	0.137	**0.128**	0.449	0.133
2	0.607	**0.605**	0.643		0.443	0.647	**0.640**	0.702		0.567
3	0.486	**0.432**	0.443		0.326	0.634	0.597	**0.581**		0.556
4	0.639	0.616	**0.616**	0.886	0.547	0.662	0.641	**0.641**	1.040	0.649
5	0.107	**0.054**	0.105		0.000	0.145	**0.108**	0.182		0.134
6	0.070	**0.035**	0.067	0.162	0.000	0.037	**0.017**	0.036	0.066	0.114
7	0.228	**0.161**	0.230	0.713	0.039	0.047	**0.099**	0.122	0.802	0.175
8	0.155	**0.145**	0.152		0.000	0.095	0.115	**0.097**		0.105
9	0.168	**0.141**	0.152		0.000	0.114	**0.097**	0.129		0.105
10	0.154	**0.101**	0.130	0.171	0.000	0.027	**0.021**	0.036	0.044	0.162
11	0.136	**0.129**	0.141	0.177	0.029	0.119	0.162	**0.161**	0.178	0.106
12	0.255	**0.167**	0.324		0.000	0.193	**0.177**	0.310		0.083
13	0.062	**0.042**	0.082	0.103	0.004	0.074	**0.076**	0.198	0.289	0.163
14	0.573	0.438	**0.414**		0.000	0.470	**0.312**	0.320		0.196
15	0.208	**0.183**	0.216	0.456	0.000	0.174	**0.411**	0.567	0.524	0.175

	Testing Loss (MSE)					Average Run Time (s)				
#	OccamNet-CPU	OccamNet-GPU	Eplex	AIF	XGB	OccamNet-CPU	OccamNet-GPU	Eplex	AIF	XGB
1	0.251	**0.141**	0.223	0.741	0.147	2246	319	4574	4498	5
2	0.704	**0.706**	0.742		0.648	4789	302	4651		7
3	0.542	0.491	**0.474**		0.464	4767	300	4696		6
4	0.713	0.679	**0.679**	0.895	0.659	4511	308	4729	4222	5
5	0.589	0.209	**0.116**		0.113	239	504	4684		3
6	0.151	**0.082**	0.091	0.088	0.234	244	479	4755	5076	3
7	0.335	0.761	0.829	**0.698**	0.316	249	468	4583	1222	1
8	0.137	0.132	**0.119**		0.094	751	359	4516		5
9	0.135	0.194	**0.150**		0.094	764	354	4517		5
10	0.084	**0.086**	0.097	0.109	0.080	483	373	4605	6196	2
11	0.180	**0.177**	0.275	0.274	0.150	259	455	4639	1223	2
12	0.223	**0.214**	0.426		0.165	944	335	4653		3
13	1.088	**0.157**	0.487	0.864	0.622	956	364	4561	6899	5
14	0.570	**0.440**	0.442		0.228	6562	354	4785		139
15	1.664	**0.334**	0.441	0.324	0.188	264	487	4533	586	29

B.2 Analysis of Fits to PMLB Datasets

In this section, we analyze the fits that the methods discussed in Section 3.6.2 provide for the PMLB dataset.

OccamNet-CPU and OccamNet-GPU provide solutions which are all short, easy to comprehend fits to the data. We find that OccamNet uses addition, subtraction, multiplication, and division most extensively, exploiting $\sin(\cdot)$ and $\cos(\cdot)$ for more nonlinearity. Interestingly, OccamNet uses $\exp(\cdot)$ and $\log|\cdot|$ less frequently, perhaps because both functions can vary widely with small changes in input, making functions with these primitives more likely to represent poor fits.

OccamNet's solutions demonstrate its ability to exploit modularity and reuse components. These solutions often have repeated components, for example in dataset #1, 1027_ESL, where OccamNet-CPU's best fit to the training data is

$$y_0 = \frac{(\sin(x_2) + x_3 + x_1) \cdot (\sin(x_2) + x_3 + x_1)}{(\sin(x_2) + x_3 + x_1) + (x_3 + x_1) + (x_1 + x_3)}.$$

In this fit, OccamNet-CPU builds $\sin(x_2) + x_3 + x_1$ in the first two layers of the network and then reuses it three times. Solutions like the above demonstrate OccamNet's ability to identify successful subcomponents of a solution and then to rearrange the subcomponents into a more useful form. Examples like the above, however, also demonstrate that OccamNet often overuses modularity, potentially restricting the domain of functions it can search. We suspect that the main reason that OccamNet may rely too heavily on modularity in some fits is that OccamNet uses an extremely high learning rate of 1 for its training. We used such a large learning rate to allow OccamNet to converge even when faced with 10^{30} or more functions. However, we suspect that this may also cause OccamNet to converge to certain paths before exploring sufficiently. For example, with the function above, OccamNet may have identified that $\sin(x_2) + x_3 + x_1$ is a useful component and, because of its high learning rate, used this pattern several times instead of the one time needed. This hypothesis is supported by the fact that OccamNet-GPU, which samples many more functions before taking a training step, repeats patterns less frequently than OccamNet-CPU. For example OccamNet-GPU's best-fit solution for the training dataset of dataset #1 is

$$y_0 = \cos(x_1/x_1) \cdot \cos(x_1/x_1) \cdot (x_2 + x_3 + \sin(x_0) + x_3 + x_1 - \sin(x_3)),$$

which contains almost no repetition.

Remarkably, for dataset #4, 1030_ERA, both Eplex and OccamNet-GPU discover equivalent functions for both training and validation: OccamNet-GPU discovers

$$y_0 = \cos(\sin(x_1 - x_2)) \cdot (\sin(x_2) + x_0 + x_1) \cdot \cos(x_2/x_2),$$

and Eplex discovers

$$y_0 = \cos(x_1/x_1) \cdot (\sin(x_2) + x_0 + x_1) \cdot \cos(\sin(x_2 - x_1)).$$

As a result, the two methods' losses are identical up to 7 decimal places. Still, we mark Eplex as performing better on this dataset because after the seventh decimal place it has a slightly lower loss, likely due to differences in rounding or precision between the two approaches. Two different methods identifying the same function is extremely unlikely; OccamNet's search space includes $2 \cdot 10^{30}$ paths for this dataset, meaning that the probability of both methods identifying this function purely by chance is minuscule. In combination with the fact that this function was the best fit to both the training and validation datasets for both methods, this suggests that the identified function is a nearly optimal fit to the data for the given search space. Given the size of the search space, this result thus provides further evidence that OccamNet and Eplex perform far better than brute-force search. Interestingly, although OccamNet-CPU did not discover this function, it's best fit for the validation,

$$y_0 = \sin(x_2/x_2) \cdot (\sin(x_2) + x_0 + x_1) \cdot \cos(\cos(x_3)) \cdot \cos(\sin(x_3)),$$

does include several features present in the fits found by OccamNet-GPU and Eplex, such as the $\sin(x_2) + x_0 + x_1$ term, the $\cos(\sin(\cdot))$ term, and the x_2/x_2 inside of the trigonometric function. This suggests that OccamNet-CPU may also have been close to converging to the function discovered by Eplex and OccamNet-GPU. OccamNet-CPU's loss was also always within 5% of Eplex's loss on this dataset, again suggesting that OccamNet-CPU had identified a function close to that of Eplex and OccamNet-GPU.

Interestingly, AI Feynman 2.0's fits generally tend to be very simple compared to those of OccamNet-CPU and OccamNet-GPU. For example, AIF's fit for the training dataset #11 is

$$y_0 = -0.050638447726 + \log(x_0/\sin(x_0)) - x_0,$$

whereas OccamNet-CPU's fit is

$$y_0 = \sin(x_0) \cdot x_1 \cdot x_0 \cdot \sin(x_0) \cdot \log|x_0| - \cos(x_1 \cdot x_0 - x_1).$$

AI Feynman's fit is slightly simpler and easier to interpret, but it comes at the cost of having a 35% higher loss. We suspect that because the PMLB datasets likely do not have modular representations, AI Feynman must rely mainly on its brute-force search, which ultimately produces shorter expressions. AI Feynman can also produce constants because of its polynomial fits, and it uses constants in nearly every solution it proposes. We did not allow the other symbolic methods to fit constants, but they still consistently performed better than AI Feynman, suggesting that fitting constants may not be essential to accurately modeling the PMLB datasets.

As discussed in the main text, OccamNet-CPU is considerably faster than Eplex, often running faster by more than an order of magnitude. This may be in part because we train Eplex with the DEAP evolutionary computation framework [109], which is implemented in Python and utilizes NumPy arrays for computation. Thus, our implementation of Eplex may be somewhat slower than an implementation written in C. However, because of its selection based on many fitness cases, Eplex is also by nature considerably slower than many other genetic algorithms, running in $O(TN^2)$, where T is the number of fitness cases and N is the population size [110]. This suggests that even a pure C implementation of Eplex may not be as fast as OccamNet-CPU. More recent selection algorithms perform comparably to Eplex but run significantly faster, for example Batch Tournament Selection [110]. However, because these methods did not exist at the time of [107], they have not been compared to other methods on the PMLB datasets. Thus, we have not tested these methods here. On the other hand, our current implementation of sampling and the forward pass work with DAGs in which an edge leads to each argument node, regardless of whether the argument node is connected to the outputs. The result is that our implementation of OccamNet evaluates more than $|\Phi|$ times more primitive functions than is necessary, where $|\Phi|$ is the number of primitive functions. In the case of these experiments, this amounts to more than eight times the number of calculations necessary. It may be possible to optimize OccamNet by not evaluating such unused connections, thereby obtaining a much faster runtime.

B.3 Analysis of PMLB Scaling Tests

As can be seen in Figure B.1, OccamNet-GPU's training loss decreases with increasing sample size for every dataset – the training loss for 64000 functions sampled is always less than the training loss for 250 functions sampled. For some datasets, OccamNet-GPU's training loss is not monotonically decreasing, but this is to be expected given OccamNet-GPU's inherent randomness and the size of the search space.

For datasets 1, 2, 3, 4, 13, and 14, the training loss does not drop noticeably when increasing the sample size beyond a certain point. There are two possible explanations for this. (i) For all of these datasets, OccamNet-GPU's training loss is very close to or lower than Eplex's best training loss, suggesting that OccamNet-GPU may be approaching an optimal fit and that there is little room to further decrease the loss. (ii) There may be critical sample sizes before which OccamNet-GPU's training loss is stagnant and beyond which its training loss begins to decrease. This is apparent in datasets 1, 3, 4, 10, 12, and 14, where the OccamNet training loss temporarily stops decreasing at 1000 or 4000 functions sampled. It is possible that for some datasets another such critical sample size exists beyond 64000 functions.

For datasets 1, 2, 3, 4, 6, 12, and 14, OccamNet-GPU's validation loss also decreases with the number of functions sampled. However, for datasets 5, 7, 8, 9, 10, and 11, it initially decreases and then begins to increase, and for datasets 13 and 15 it does not decrease. Interestingly, the datasets for which OccamNet-GPU's testing loss does not decrease are generally the same as the datasets for which OccamNet-GPU's validation loss does not decrease. The datasets where the validation and testing loss do not decrease are generally very small, with around 200 or fewer datapoints. This suggests that OccamNet-GPU is overfitting. Such overfitting is to be expected given the small number of samples in the PMLB datasets. On the other hand, Eplex only seems to overfit in datasets 5 and 9. Because overfitting results from fitting a training or validation dataset too well, this is further evidence that OccamNet-GPU is fitting the training datasets better than Eplex.

Note that for each number of functions sampled, we tested 81 different hyperparameter combinations for OccamNet-GPU and only 3 for Eplex. This is largely because, as a new architecture, OccamNet-GPU's optimal hyperparameters are not known. For a fair comparison, the runtimes we report in Figure B.1 are the times required to run all hyperparameter combinations. Thus, because OccamNet-GPU uses 27 times more hyperparameter combinations, its speed advantage is lessened, although still significant.

After examining the results for all hyperparameter combinations, however, we noted that all hyperparameters but the learning rate had an "optimal" value, listed in Appendix 3.6.2. Restricting to only the remaining three hyperparameter combinations produces a best training loss that is often the same as, and is never more than 40% greater than, the lowest loss among all 81 hyperparameters. Figure B.2 shows the results when considering only OccamNet-GPU's restricted hyperparameter combinations for three datasets.

When OccamNet-GPU and Eplex are restricted to the same number of hyperparameter combinations, OccamNet-GPU always runs faster when sampling 64000 functions per epoch than Eplex does when sampling 1000 functions per epoch. OccamNet-GPU's training loss consistently decreases with increasing sample size, although its validation and testing losses do not always follow such a clear trend. Further, OccamNet-GPU almost always converges to training and validation losses that are close to or less than Eplex's training and validation losses. OccamNet-GPU's best training loss is less than or approximately the same as Eplex's best training loss for all but datasets 1, 14, and 15.

Dataset 14 consists of over 1000 datapoints with 117 features, so it is likely one of the most difficult datasets which we test. The fact that OccamNet-GPU does perform comparatively to Eplex when it tests additional hyperparameter combinations suggests that for such difficult problems OccamNet-GPU benefits from additional hyperparameter exploration, particularly involving weight initialization.

For dataset 15, because Eplex only identifies a better function than OccamNet-GPU when it samples 1000 functions and not when it samples 2000 or 4000 functions, Eplex's better performance appears to be somewhat of an outlier.

With the restricted set of hyperparameters, OccamNet-GPU still overfits on every dataset it overfitted on when using the full set of hyperparameters, suggesting that the overfitting is not due to the large number of hyperparameters. Interestingly, for both the full and restricted hyperparameter versions of OccamNet-GPU, OccamNet-GPU and Eplex again identify the same fit for Dataset 4,

$$y_0 = \cos(\sin(x_1 - x_2)) \cdot (\sin(x_2) + x_0 + x_1) \cdot \cos(x_3/x_3).$$

B.4 Ablation Studies

We test the performance of various hyperparameters in a collection of ablation studies, as shown in Table B.2. Here, we focus on what our experiments demonstrate to be the most critical parameters to be tuned: the collection of primitives and constants, the network depth, the variance of our interpolating function, the overall network temperature (as well as the last layer temperature), and, finally, the learning rate of our optimizer. As before, we set the stop criterion and terminate learning when the top-λ sampled functions all return the same fitness $K(\cdot, f)$ for 30 consecutive epochs. If this does not occur in a predefined, fixed number of iterations, or if the network training terminates and the final expression does not match the correct function we aim to fit, we say that the network has not converged. All hyperparameters for baselines are specified in Section 3.6.2, except for the sampling size, which is set to $R = 100$.

Our benchmarks use a sampling size large enough for convergence in most experiments. It is worth noting, however, that deeper networks sometimes failed to converge (with a convergence fraction of $\eta = 8/10$) for the analytic function we tested. Deeper networks allow for more function composition and let approximations emerge as local minima: in practice, we find that increasing the last layer temperature or reducing the variance is often needed to allow for a larger depth L. For pattern recognition, we found that *MNIST Binary* and *Trinary* require depth 2 for successful convergence, while the rest of the experiments require depth 4. Shallower or deeper networks either yield subpar accuracy or fail to converge. We also find that for OccamNet without skip connections, larger learning rates usually work best, i.e., 0.05 works best, while OccamNet with skip connections requires a smaller learning rate, usually around 0.0005. We also tested different temperature and variance schedulers in the spirit of simulated annealing. In particular, we tested increasing or decreasing these parameters over training epochs, as well as sinusoidally varying them with different frequencies. Despite the increased convergence time, however, we did not find any additional benefits of using schedulers. As we test OccamNet in larger problem spaces, we will revisit these early scheduling studies and investigate their effects in those domains.

B.5 Neural Approaches to Benchmarks

Since OccamNet is a neural model that is constructed on top of a fully connected neural architecture, below we consider a limitation of the standard fully connected architectures for sorting and then a simple application of our temperature-controlled connectivity.

E-1. Exploring the limits of fully connected neural architectures for sorting

We made a fully connected neural network with residual connections. We used the mean squared error (MSE) as the loss function. The output size was equal to the input size and

Table B.2: Ablation studies on representative experiments

Modification	Convergence fraction η	Convergence epochs T_c
Experiment $\sin(3x+2)$		
baseline	10/10	390
added constants (2) and primitives $(\cdot, (\cdot)^2, -(\cdot))$	10/10	710
lower last layer temperature (0.5)	10/10	300
higher last layer temperature (3)	10/10	450
lower learning rate (0.001)	10/10	2500
higher learning rate (0.01)	10/10	170
deeper network (6)	8/10	3100
lower variance (0.0001)	10/10	390
higher variance (0.1)	10/10	450
lower sampling (50)	10/10	680
higher sampling (250)	10/10	200
Experiment x^2 if $x > 0$, else $-x$		
baseline	10/10	100
added constants (1, 2) and primitives $(-, -(\cdot))$	10/10	290
lower last layer temperature (0.5)	10/10	160
higher last layer temperature (3)	10/10	150
lower learning rate (0.001)	10/10	780
higher learning rate (0.01)	10/10	90
deeper network (6)	10/10	180
shallower network (2)	10/10	160
lower variance (0.001)	10/10	160
higher variance (1)	10/10	180
lower sampling (50)	10/10	290
higher sampling (250)	10/10	140

represented the original numbers in sorted order. We used L_2 regularization along with Adam optimization. We tested weight decay ranging from 1e-2 to 1e-6 and found that 1e-5 provided the best training and testing accuracy. Finally, we found that the optimal learning rate was around 1e-3. We used $30,000$ data points to train the model with batch size of 200. Each of the data points is a list of numbers between 0 and 100. For a particular value of input size x (representing the number of points to be sorted), we varied the number of hidden units from 2 to 20 and the number of hidden layers from 2 (just an input and an output layer) to $x! + 2$. Then, the test loss was calculated on 20,000 points, chosen from same distribution. Finally, for each input size, Table B.3 records the combination (hidden_layer, hidden_unit) for which the loss is less than 5 and (hidden_ layer * hidden_units) is minimized. As seen from the table, the system failed to find any optimal combination for any input size greater than or equal to 5. For example, for input size 5, the hidden units were upper capped at 20 and hidden layers at 120 and thus 2400 parameters were insufficient to sort 5 numbers.

E-2. Generalization

The model developed above generalizes poorly on data outside the training domain. For example, consider the model with 18 hidden units and four hidden layers, which is successfully trained to sort four numbers chosen from the range 0 to 100. It was first tested on numbers from 0 to 100 and then on 100 to 200. The error in the first case was around

Table B.3: Minimal configurations to sort list of length "input size."

Input Size	Hidden units	Hidden Layers	Parameters
2	6	2	12
3	8	4	32
4	18	4	72
5	-	-	-

2 while the average error in the second case was between 6 and 8 (which is $(200/100)^2 = 4$ times the former loss). Finally, when tested on larger ranges such as $(9900, 10000)$, the error exploded to around 0.1 million (which is an order greater than $(10000/100)^2 = 10000$ times the original loss). This gives a hint that the error might be scaling proportionally to the square of the test domain with respect to the train domain. A possible explanation for this comes from the use of the MSE loss function. Scaling test data by ρ scales the absolute error by approximately the same factor and then taking a square of the error to calculate the MSE scales the total loss by the square of that factor, i.e., ρ^2.

E-3. Applying temperature-controlled connectivity to standard neural networks for MNIST classification

We would like to demonstrate the promise of temperature-controlled connectivity as a regularization method for the classification heads of models with a very simple experiment. We used the ResNet50 model to train on the standard MNIST image classification benchmark. We studied two variants of the model: the standard ResNet model and ResNet augmented with our temperature-controlled connectivity (with $T = 1$) between the flattened layer and the last fully connected layer (on the lines discussed in the main paper). Then we trained both models with a learning rate fixed at 0.05 and a batch size of 64 and ran it for 10 epochs. The model with regularization performed slightly better than the one without it. The regularized model achieved the maximum accuracy among all methods, 99.18%, while the same figure for the standard one was 98.43%. Another interesting observation is that the regularized model produces much more stable and consistent results across iterations than the unregularized model. These results encourage us to study the above regularization method in larger experiments.

B.6 Small Experiments

To demonstrate its ability to fit functions with constants, we also tested OccamNet on the function $10.5x^{3.1}$ without providing either 10.5 or 3.1 beforehand. OccamNet identified the correct function 10 times out of 10, taking an average of 553s.

We also investigated whether OccamNet could discover a formula for cosine using only the primitives $\sin(\cdot)$, $+(\cdot,\cdot)$, and $\div(\cdot,\cdot)$ and the constants 2 and π. We expected OccamNet to discover $\cos(x) = \sin(x + \pi/2)$, but, interestingly, it instead always identified the double angle identity $\cos(x) = \sin(2x)/(2\sin(x))$. OccamNet successfully identified an identity for cosine 8 out of 10 times and in an average of 410s. A more optimized implementation of OccamNet takes only 7s for the same task, although its accuracy is somewhat lower, fluctuating between 2 and 6 out of 10 correct identifications.

Similarly, we tested whether OccamNet could discover Taylor polynomials of e^x. OccamNet identified $e^x \approx 1 + x + x^2/2$, but was unable to discover the subsequent $x^3/6$ term.

B.7 Related Work

A Symbolic regression

OccamNet was partially inspired by the EQL network [89, 90, 91], a neural network-based symbolic regression system that successfully finds simple analytic functions. Neural Arithmetic Logic Units (NALU) and related models [84, 85] provide a neural inductive bias for arithmetic in neural networks that can in principle fit some benchmarks in Table 3.1. NALU updates the weights by backpropagating through the activations, shaping the neural network towards a gating interpretation of the linear layers. However, generalizing those models to a diverse set of function primitives might be a formidable task: from our experiments, backpropagation through some activation functions (such as division or sine) makes training considerably harder. In a different computational paradigm, genetic programming (GP) has performed exceptionally well at symbolic regression [68, 69], and a number of evolution-inspired, probability-based models have been explored for this goal [76].

A concurrent work [77] explores deep symbolic regression by using an RNN to search the space of expressions using autoregressive expression generation. Interestingly, the authors observed that a risk-aware reinforcement learning approach is a necessary component in their optimization, which is similar to our approach of selecting the top λ function for optimization in Step 2 of our algorithm. A notable difference is that OccamNet does not generate the expressions autoregressively, although it still exhibits a gradual increase in modularity during training, as discussed in Section 3.5. This is also a benefit both for speed and scalability. Moreover, their entropy regularization is a potentially useful addition to our training algorithm. Marrying our approach with theirs is a promising direction for future work.

Transformer-based models can quickly and accurately identify functions given data by leveraging their extensive pretraining. However, these approaches are limited in that they are restricted to a set of primitive functions specified at training time. It may thus be fruitful to investigate combining OccamNet and such approaches in a way that increases convergence speed while maintaining OccamNet's flexibility.

B Program synthesis

A field related to symbolic regression is program synthesis. For programs, one option to fit programs is to use EQL-based models with logic activations (step functions, MIN, MAX, etc.) approximated by sigmoid activations. Another is probabilistic program induction using domain-specific languages [79, 80, 81]. Neural Turing Machines [86, 87] and their stable versions [88] are also able to discover interpretable programs based on observations of input-output pairs. They do so by simulating programs using neural networks connected to an external memory. [82] first train a machine learning model to predict a DSL based on input-output pairs and then use methods from satisfiability modulo theory [83] to search the space of programs built using the predicted DSL. In contrast, our DSL is lower level and can fit components like "sort" instead of including them in the DSL directly. [111] develop a neural model for simple algorithmic tasks by utilizing memory access for pointer manipulation and dereferencing. However, here we achieve similar results (for example, sorting) without external memory and in only minutes on a CPU.

C Integration with deep learning

We are not aware of classifiers that predict MNIST or ImageNet labels using symbolic rules. The closest baseline we found is using GP [112], which performs comparably well to our neural method, but cannot easily integrate with deep learning. In the reinforcement learning (RL) domain, [73] and [113] propose training deep models of millions of parameters on standard RL tasks using a gradient-free GP, which is competitive to gradient-based RL algorithms. Work such as [91] performs similar tasks with EQL, a less powerful symbolic regression model.

D SCGs and pruning

Treating the problem of finding the correct function or program as a stochastic computational graph is appealing due to efficient soft approximations to discrete distributions [114, 115, 116]. Our T-softmax layers offer such an approximation and could further benefit from an adaptive softmax methodology [117], which we leave for future work. Furthermore, the sparsity induced by T-softmax layers parallels the abundant work on pruning connections and weights in neural networks [118, 119] or using regularizations, encouraging sparse connectivity [120, 121].

B.8 Information about Symbolic Regression Benchmarks

H-1. Eureqa

Eureqa is a software package for symbolic regression where one can specify different target expressions, building block functions (analogous to the primitives in OccamNet), and loss functions [68]. For most functions, we use the absolute error as the optimization metric. We choose formula building blocks in Eureqa to match the primitive functions used in OccamNet.

For implicit functions, we use the implicit derivative error. We also order the data to improve the performance. For the implicit functions in lines 1, 3, and 4 in Table 2 of the main text, the data is ordered by x_0. For the equation $x_0^2 + x_1^2 = 1$, the data is generated by sampling $\theta \in [0, 2\pi)$ and calculating $x_0 = \sin(x)$ and $x_1 = \cos(x)$, and is ordered by θ. When the data is not ordered, the value of the implicit derivative error is much higher, resulting in the algorithm favoring incorrect equations. For equation $m_1v_1 - m_2v_2$, the ordering is more ambiguous because of the higher dimensionality. We tried ordering by both m_1 and the product m_1v_1 without success.

H-2. HeuristicLab

Due to limits on the number of data points and feature columns in Eureqa, we instead use HeuristicLab for the image recognition tasks described in Section 5.4 of the main text. HeuristicLab is a software package for optimization and evolutionary algorithms, including symbolic regression and symbolic classification. We use the Island Genetic Algorithm with default settings.

Similar to the building block functions in Eureqa, HeuristicLab can specify the primitive symbols for each task. However, HeuristicLab does not have the primitives MAX, SIGMOID, or tanh. Instead, we use the symbols IfThenElse, GreaterThan, LessThan, And, Or, and Not.

H-3. Eplex

As discussed in Section 3.6.2, Eplex [96], short for Epsilon-Lexicase selection, is a genetic programming population selection technique that we use as a symbolic regression benchmark in our experiments with PMLB datasets. We implement a genetic algorithm using Eplex with the DEAP [109] evolutionary framework, using Numpy arrays [122] for computation to increase speed.

Eplex selects individuals from a population by evaluating the individuals on subsets, or fitness cases, of the full data. For each fitness case, Eplex selects the top-performing individuals and proceeds to the next fitness case. This process is repeated until only one individual remains. This individual is then used as the parent for the next generation.

H-4. AI Feynman 2.0 We also benchmark OccamNet against AI Feynman 2.0 [97]. AI Feynman 2.0 is a mixed approach that combines brute-force symbolic regression, polynomial fits, and identification for modularity in the data using neural networks. To identify modularities in the data, AI Feynman first trains a neural network on it. This serves as an interpolating function for the true data and allows the network to search for symmetries and other forms of modularities.

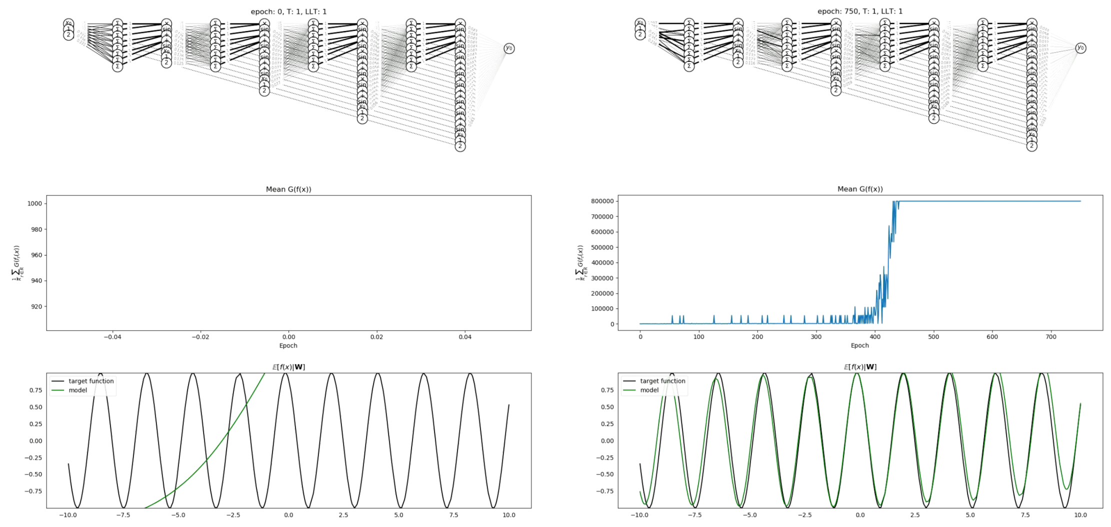


Figure B.3: In this figure, we present two video frames for the target $\sin(3x+2)$, which could be accessed via `videos/sin(3x + 2).mp4` in our code files. We show the beginning of the fitting (left) and the end, where OccamNet has almost converged (right).

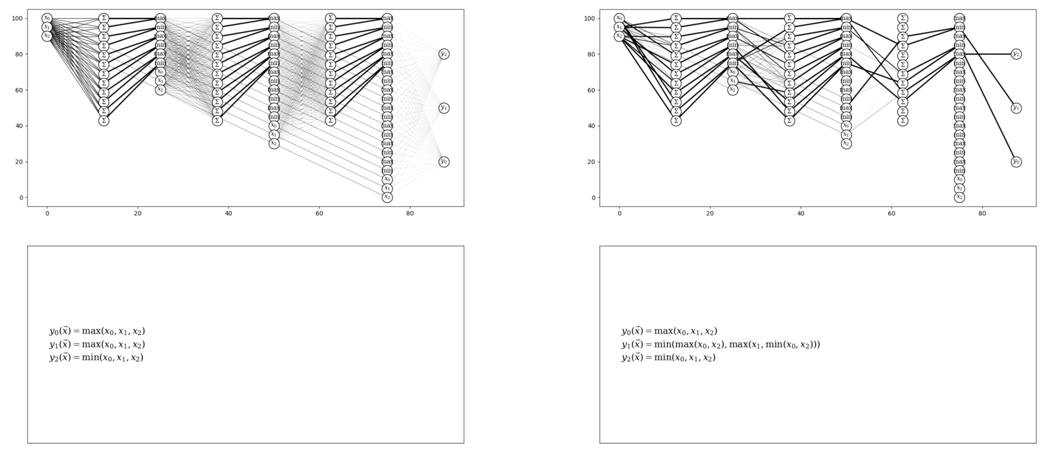

$y_0(\vec{x}) = \max(x_0, x_1, x_2)$
$y_1(\vec{x}) = \max(x_0, x_1, x_2)$
$y_2(\vec{x}) = \min(x_0, x_1, x_2)$
$y_0(\vec{x}) = \max(x_0, x_1, x_2)$
$y_1(\vec{x}) = \min(\max(x_0, x_2), \max(x_1, \min(x_0, x_2)))$
$y_2(\vec{x}) = \min(x_0, x_1, x_2)$

Figure B.4: In this figure we present two video frames for the target $\mathsf{SORT}(x_0, x_1, x_2)$, which could be accessed via `videos/sorting.mp4` in our code files. We show the beginning of the fitting (left) and the end, where OccamNet has almost converged (right).

B.9 Code, Videos, and Responsible Use

Code for all iterations of OccamNet is available at https://github.com/druidowm/OccamNet_Versions. The code used for this paper can be found at https://github.com/druidowm/OccamNet_Public. We have grouped our code into five main folders. `analytic-and-programs` stores our network and experiments for fitting analytic functions and programs. `implicit` stores our network and experiments for implicit functions, although it also includes the three analytic functions listed in Table 3.6. `constant-fitting` stores code very similar to `implicit` but optimized for constant fitting. `image-recognition` stores our network and experiments for image classification. `pmlb-experiment` stores our code for benchmarking against the PMLB regression datasets. Finally, `videos` stores several videos of our model converging to various functions. In Figures B.3 and B.4, we present snapshots of the videos.

Currently, our method is not explicitly designed against adversarial attacks. Thus, malicious stakeholders could exploit our method and manipulate the symbolic fits that OccamNet produces. A potential direction towards alleviating the problem would be to explore ways to robustify OccamNet by training it against an adversary. In the meantime, we ask that users of our code remain responsible and consider the repercussions of their use cases.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. “Attention Is All You Need”. In: *arXiv e-prints*, arXiv:1706.03762 (June 2017), arXiv:1706.03762. DOI: 10.48550/arXiv.1706.03762. arXiv: 1706.03762 [cs.CL].

[3] OpenAI et al. “GPT-4 Technical Report”. In: *arXiv e-prints*, arXiv:2303.08774 (Mar. 2023), arXiv:2303.08774. DOI: 10.48550/arXiv.2303.08774. arXiv: 2303.08774 [cs.CL].

[4] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. “End-to-End Training of Deep Visuomotor Policies”. In: *arXiv e-prints*, arXiv:1504.00702 (Apr. 2015), arXiv:1504.00702. DOI: 10.48550/arXiv.1504.00702. arXiv: 1504.00702 [cs.LG].

[5] Igor Melnyk, Aurelie Lozano, Payel Das, and Vijil Chenthamarakshan. “AlphaFold Distillation for Protein Design”. In: *arXiv e-prints*, arXiv:2210.03488 (Oct. 2022), arXiv:2210.03488. DOI: 10.48550/arXiv.2210.03488. arXiv: 2210.03488 [q-bio.BM].

[6] S. V. Chekanov and W. Hopkins. “Event-based anomaly detection for new physics searches at the LHC using machine learning”. In: *arXiv e-prints*, arXiv:2111.12119 (Nov. 2021), arXiv:2111.12119. DOI: 10.48550/arXiv.2111.12119. arXiv: 2111.12119 [hep-ph].

[7] Junze Liu, Jordan Ott, Julian Collado, Benjamin Jargowsky, Wenjie Wu, Jianming Bian, and Pierre Baldi. “Deep-Learning-Based Kinematic Reconstruction for DUNE”. In: *arXiv e-prints*, arXiv:2012.06181 (Dec. 2020), arXiv:2012.06181. DOI: 10.48550/arXiv.2012.06181. arXiv: 2012.06181 [physics.ins-det].

[8] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[9] Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J. Reddi, and Sanjiv Kumar. “Are Transformers universal approximators of sequence-to-sequence functions?” In: *arXiv e-prints*, arXiv:1912.10077 (Dec. 2019), arXiv:1912.10077. DOI: 10.48550/arXiv.1912.10077. arXiv: 1912.10077 [cs.LG].

[10] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pp. 533–536. URL: https://api.semanticscholar.org/CorpusID:205001834.

[11] Owen Dugan, Peter Y. Lu, Rumen Dangovski, Di Luo, and Marin Soljačić. "Q-Flow: Generative Modeling for Differential Equations of Open Quantum Dynamics with Normalizing Flows". In: *arXiv e-prints*, arXiv:2302.12235 (Feb. 2023), arXiv:2302.12235. DOI: 10.48550/arXiv.2302.12235. arXiv: 2302.12235 [quant-ph].

[12] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *arXiv e-prints*, arXiv:1712.01815 (Dec. 2017), arXiv:1712.01815. DOI: 10.48550/arXiv.1712.01815. arXiv: 1712.01815 [cs.AI].

[13] Owen Dugan, Rumen Dangovski, Allan Costa, Samuel Kim, Pawan Goyal, Joseph Jacobson, and Marin Soljačić. "OccamNet: A Fast Neural Model for Symbolic Regression at Scale". In: *arXiv e-prints*, arXiv:2007.10784 (July 2020), arXiv:2007.10784. DOI: 10.48550/arXiv.2007.10784. arXiv: 2007.10784 [cs.LG].

[14] Frank Verstraete, Michael M Wolf, and J Ignacio Cirac. "Quantum computation and quantum-state engineering driven by dissipation". In: *Nature physics* 5.9 (2009), pp. 633–636.

[15] Julio T Barreiro, Markus Müller, Philipp Schindler, Daniel Nigg, Thomas Monz, Michael Chwalla, Markus Hennrich, Christian F Roos, Peter Zoller, and Rainer Blatt. "An open-system quantum simulator with trapped ions". In: *Nature* 470.7335 (2011), pp. 486–491.

[16] Filippo Vicentini, Alberto Biella, Nicolas Regnault, and Cristiano Ciuti. "Variational Neural-Network Ansatz for Steady States in Open Quantum Systems". In: *Phys. Rev. Lett.* 122 (25 June 2019), p. 250503. DOI: 10.1103/PhysRevLett.122.250503. URL: https://link.aps.org/doi/10.1103/PhysRevLett.122.250503.

[17] Nobuyuki Yoshioka and Ryusuke Hamazaki. "Constructing neural stationary states for open quantum many-body systems". In: *Phys. Rev. B* 99 (21 June 2019), p. 214306. DOI: 10.1103/PhysRevB.99.214306. URL: https://link.aps.org/doi/10.1103/PhysRevB.99.214306.

[18] Michael J. Hartmann and Giuseppe Carleo. "Neural-Network Approach to Dissipative Quantum Many-Body Dynamics". In: *Phys. Rev. Lett.* 122 (25 June 2019), p. 250502. DOI: 10.1103/PhysRevLett.122.250502. URL: https://link.aps.org/doi/10.1103/PhysRevLett.122.250502.

[19] Alexandra Nagy and Vincenzo Savona. "Variational Quantum Monte Carlo Method with a Neural-Network Ansatz for Open Quantum Systems". In: *Phys. Rev. Lett.* 122 (25 June 2019), p. 250501. DOI: 10.1103/PhysRevLett.122.250501. URL: https://link.aps.org/doi/10.1103/PhysRevLett.122.250501.

[20] Di Luo, Zhuo Chen, Juan Carrasquilla, and Bryan K Clark. "Autoregressive neural network for simulating open quantum systems via a probabilistic formulation". In: *Physical review letters* 128.9 (2022), p. 090501.

[21] Moritz Reh, Markus Schmitt, and Martin Gärttner. “Time-dependent variational principle for open quantum systems with artificial neural networks”. In: *Physical Review Letters* 127.23 (2021), p. 230501.

[22] Maziar Raissi. “Deep Hidden Physics Models: Deep Learning of Nonlinear Partial Differential Equations”. In: *Journal of Machine Learning Research* 19.25 (2018), pp. 1–24. URL: http://jmlr.org/papers/v19/18-046.html.

[23] M. Raissi, P. Perdikaris, and G.E. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2018.10.045. URL: http://www.sciencedirect.com/science/article/pii/S0021999118307125.

[24] Howard J. Carmichael. “Quantum—Classical Correspondence for the Electromagnetic Field II: P, Q, and Wigner Representations”. In: *Statistical Methods in Quantum Optics 1: Master Equations and Fokker-Planck Equations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 101–145. ISBN: 978-3-662-03875-8. DOI: 10.1007/978-3-662-03875-8_4. URL: https://doi.org/10.1007/978-3-662-03875-8_4.

[25] Laurent Dinh, David Krueger, and Yoshua Bengio. “Nice: Non-linear independent components estimation”. In: *arXiv preprint arXiv:1410.8516* (2014).

[26] Danilo Rezende and Shakir Mohamed. “Variational inference with normalizing flows”. In: *International conference on machine learning*. PMLR. 2015, pp. 1530–1538.

[27] William Lauchlin McMillan. “Ground state of liquid He 4”. In: *Physical Review* 138.2A (1965), A442.

[28] Shun-ichi Amari. “Neural learning in structured parameter spaces-natural Riemannian gradient”. In: *Advances in neural information processing systems* 9 (1996).

[29] Shun-Ichi Amari. “Natural gradient works efficiently in learning”. In: *Neural computation* 10.2 (1998), pp. 251–276.

[30] Giuseppe Carleo and Matthias Troyer. “Solving the quantum many-body problem with artificial neural networks”. In: *Science* 355.6325 (2017), pp. 602–606. DOI: 10.1126/science.aag2302.

[31] Or Sharir, Yoav Levine, Noam Wies, Giuseppe Carleo, and Amnon Shashua. “Deep Autoregressive Models for the Efficient Variational Simulation of Many-Body Quantum Systems”. In: *Phys. Rev. Lett.* 124 (2 Jan. 2020), p. 020503. DOI: 10.1103/PhysRevLett.124.020503.

[32] Di Luo, Zhuo Chen, Juan Carrasquilla, and Bryan K. Clark. “Autoregressive Neural Network for Simulating Open Quantum Systems via a Probabilistic Formulation”. In: *Phys. Rev. Lett.* 128 (9 Feb. 2022), p. 090501. DOI: 10.1103/PhysRevLett.128.090501.

[33] Zhuo Chen, Di Luo, Kaiwen Hu, and Bryan K Clark. “Simulating 2+ 1D Lattice Quantum Electrodynamics at Finite Density with Neural Flow Wavefunctions”. In: *arXiv preprint arXiv:2212.06835* (2022).

[34] Di Luo, Zhuo Chen, Kaiwen Hu, Zhizhen Zhao, Vera Mikyoung Hur, and Bryan K Clark. "Gauge invariant autoregressive neural networks for quantum lattice models". In: *arXiv preprint arXiv:2101.07243* (2021).

[35] David Pfau, James S. Spencer, Alexander G. D. G. Matthews, and W. M. C. Foulkes. "Ab initio solution of the many-electron Schrödinger equation with deep neural networks". In: *Phys. Rev. Res.* 2 (3 Sept. 2020), p. 033429. DOI: 10.1103/PhysRevResearch.2.033429.

[36] Jan Hermann, Zeno Schätzle, and Frank Noé. "Deep-neural-network solution of the electronic Schrödinger equation". In: *Nature Chemistry* 12.10 (Oct. 2020), pp. 891–897. ISSN: 1755-4349. DOI: 10.1038/s41557-020-0544-y.

[37] Di Luo and Bryan K Clark. "Backflow transformations via neural networks for quantum many-body wave functions". In: *Physical review letters* 122.22 (2019), p. 226401.

[38] Gabriel Pescia, Jiequn Han, Alessandro Lovato, Jianfeng Lu, and Giuseppe Carleo. "Neural-network quantum states for periodic systems in continuous space". In: *Physical Review Research* 4.2 (2022), p. 023138.

[39] Matija Medvidović and Dries Sels. "Towards unitary dynamics of large two-dimensional quantum rotor models". In: *arXiv preprint arXiv:2212.11289* (2022).

[40] Di Luo, Shunyue Yuan, James Stokes, and Bryan K Clark. "Gauge equivariant neural networks for 2+ 1d u (1) gauge theory simulations in hamiltonian formulation". In: *arXiv preprint arXiv:2211.03198* (2022).

[41] John M Martyn, Khadijeh Najafi, and Di Luo. "Variational Neural-Network Ansatz for Continuum Quantum Field Theory". In: *arXiv preprint arXiv:2212.00782* (2022).

[42] Bengt Fornberg. *A practical guide to pseudospectral methods.* 1. Cambridge university press, 1998.

[43] E Weinan, Jiequn Han, and Arnulf Jentzen. "Algorithms for solving high dimensional PDEs: from nonlinear Monte Carlo to machine learning". In: *Nonlinearity* 35.1 (2021), p. 278.

[44] Jens Berg and Kaj Nyström. "Data-driven discovery of PDEs in complex datasets". In: *Journal of Computational Physics* 384 (2019), pp. 239–252. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2019.01.036. URL: http://www.sciencedirect.com/science/article/pii/S0021999119300944.

[45] Aditi Krishnapriyan, Amir Gholami, Shandian Zhe, Robert Kirby, and Michael W Mahoney. "Characterizing possible failure modes in physics-informed neural networks". In: *Advances in Neural Information Processing Systems*. Ed. by M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan. Vol. 34. Curran Associates, Inc., 2021, pp. 26548–26560. URL: https://proceedings.neurips.cc/paper/2021/file/df438e5206f31600e6ae4af72f2725f1-Paper.pdf.

[46] Miguel Angel Cazalilla, Roberta Citro, Thierry Giamarchi, Edmond Orignac, and Marcos Rigol. "One dimensional bosons: From condensed matter systems to ultracold gases". In: *Reviews of Modern Physics* 83.4 (2011), p. 1405.

[47] Gerardo Adesso, Sammy Ragy, and Antony R Lee. “Continuous variable quantum information: Gaussian states and beyond”. In: *Open Systems & Information Dynamics* 21.01n02 (2014), p. 1440001.

[48] Howard J. Carmichael. “Dissipation in Quantum Mechanics: The Master Equation Approach”. In: *Statistical Methods in Quantum Optics 1: Master Equations and Fokker-Planck Equations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 1–28. ISBN: 978-3-662-03875-8. DOI: 10.1007/978-3-662-03875-8_1. URL: https://doi.org/10.1007/978-3-662-03875-8_1.

[49] Tom Westerhout, Nikita Astrakhantsev, Konstantin S Tikhonov, Mikhail I Katsnelson, and Andrey A Bagrov. “Generalization properties of neural network approximations to frustrated magnet ground states”. In: *Nature communications* 11.1 (2020), p. 1593.

[50] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. “Density estimation using Real NVP”. In: *International Conference on Learning Representations*. 2017. URL: https://openreview.net/forum?id=HkpbnH9lx.

[51] Will Grathwohl, Ricky T. Q. Chen, Jesse Bettencourt, and David Duvenaud. “Scalable Reversible Generative Models with Free-form Continuous Dynamics”. In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=rJxgknCcK7.

[52] Chin-Wei Huang, Ricky T. Q. Chen, Christos Tsirigotis, and Aaron Courville. “Convex Potential Flows: Universal Probability Distributions with Optimal Transport and Convex Optimization”. In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=te7PVH1sPxJ.

[53] Moritz Reh and Martin Gärttner. “Variational Monte Carlo approach to partial differential equations with neural networks”. In: *Machine Learning: Science and Technology* 3.4 (2022), 04LT02.

[54] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/google/jax.

[55] Giuseppe Carleo et al. “NetKet: A Machine Learning Toolkit for Many-Body Quantum Systems”. In: *SoftwareX* (2019), p. 100311. DOI: 10.1016/j.softx.2019.100311. URL: http://www.sciencedirect.com/science/article/pii/S2352711019300974.

[56] Filippo Vicentini et al. “NetKet 3: Machine Learning Toolbox for Many-Body Quantum Systems”. In: *SciPost Phys. Codebases* (2022), p. 7. DOI: 10.21468/SciPostPhysCodeb.7. URL: https://scipost.org/10.21468/SciPostPhysCodeb.7.

[57] Sandro Sorella. “Green function Monte Carlo with stochastic reconfiguration”. In: *Physical review letters* 80.20 (1998), p. 4558.

[58] Sandro Sorella. “Generalized Lanczos algorithm for variational quantum Monte Carlo”. In: *Physical Review B* 64.2 (2001), p. 024512.

[59] Dion Häfner and Filippo Vicentini. “mpi4jax: Zero-copy MPI communication of JAX arrays”. In: *Journal of Open Source Software* 6.65 (2021), p. 3419. DOI: 10.21105/joss.03419. URL: https://doi.org/10.21105/joss.03419.

[60] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).

[61] Christopher Rackauckas and Qing Nie. “Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia”. In: *Journal of Open Research Software* 5.1 (2017), p. 15.

[62] G. Kordas, S. Wimberger, and D. Witthaut. “Decay and fragmentation in an open Bose-Hubbard chain”. In: *Phys. Rev. A* 87 (4 Apr. 2013), p. 043618. DOI: 10.1103/PhysRevA.87.043618. URL: https://link.aps.org/doi/10.1103/PhysRevA.87.043618.

[63] G. Kordas, D. Witthaut, P. Buonsante, A. Vezzani, R. Burioni, A. I. Karanikas, and S. Wimberger. “The dissipative Bose-Hubbard model”. In: *European Physical Journal Special Topics* 224.11 (Nov. 2015), pp. 2127–2171. DOI: 10.1140/epjst/e2015-02528-2. arXiv: 1510.00127 `[cond-mat.quant-gas]`.

[64] Julia Balla, Sihao Huang, Owen Dugan, Rumen Dangovski, and Marin Soljacic. “AI-Assisted Discovery of Quantitative and Formal Models in Social Science”. In: *arXiv e-prints*, arXiv:2210.00563 (Oct. 2022), arXiv:2210.00563. DOI: 10.48550/arXiv.2210.00563. arXiv: 2210.00563 `[cs.SC]`.

[65] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep Learning”. In: *Nature* 521.7553 (2015), pp. 436–444.

[66] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. “The Loss Surfaces of Multilayer Networks”. In: *AISTATS*. 2015.

[67] Guillaume Lample and François Charton. “Deep Learning For Symbolic Mathematics”. In: *ICLR*. 2020.

[68] Michael Schmidt and Hod Lipson. “Distilling free-form natural laws from experimental data”. In: *Science* 324.5923 (2009), pp. 81–85.

[69] Silviu-Marian Udrescu and Max Tegmark. “AI Feynman: A physics-inspired method for symbolic regression”. In: *Science Advances* 6.16 (2020).

[70] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming.* lulu.com, 2008.

[71] P. J. Angeline, G. M. Saunders, and J. B. Pollack. “An evolutionary algorithm that constructs recurrent neural networks”. In: *IEEE Transactions on Neural Networks* 5.1 (1994), pp. 54–65.

[72] Dirk V. Arnold and Nikolaus Hansen. “A (1+1)-CMA-ES for Constrained Optimisation”. In: *GECCO*. 2012.

[73] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. “Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning”. In: *arXiv preprint arXiv:1712.06567* (2017).

[74] Nikolaus Hansen. "The CMA Evolution Strategy: A Tutorial". In: *arXiv preprint arXiv:1604.00772* (2016).

[75] Ilya Loshchilov and Frank Hutter. "CMA-ES for Hyperparameter Optimization of Deep Neural Networks". In: *arXiv preprint arXiv:1604.07269* (2016).

[76] Robert I Mckay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'neill. "Grammar-based genetic programming: a survey". In: *Genetic Programming and Evolvable Machines* 11.3-4 (2010), pp. 365–396.

[77] Brenden K Petersen, Mikel Landajuela Larma, Terrell N. Mundhenk, Claudio Prata Santiago, Soo Kyung Kim, and Joanne Taery Kim. "Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients". In: *ICLR*. 2021.

[78] Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. "Neural Symbolic Regression that Scales". In: *arXiv e-prints*, arXiv:2106.06427 (June 2021), arXiv:2106.06427. DOI: 10.48550/arXiv.2106.06427. arXiv: 2106.06427 [cs.LG].

[79] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. "Learning to Infer Graphics Programs from Hand-Drawn Images". In: *NIPS*. 2018.

[80] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. "Learning Libraries of Subroutines for Neurally–Guided Bayesian Program Induction". In: *NIPS*. 2018.

[81] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. "Write, Execute, Assess: Program Synthesis with a REPL". In: *NeurIPS*. 2019.

[82] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. "DeepCoder: Learning to Write Programs". In: *ICLR*. 2016.

[83] Armando Solar Lezama. "Program Synthesis By Sketching". PhD thesis. EECS Department, University of California, Berkeley, 2008.

[84] Andrew Trask, Felix Hill, Scott E Reed, Jack Rae, Chris Dyer, and Phil Blunsom. "Neural Arithmetic Logic Units". In: *NIPS*. 2018.

[85] Andreas Madsen and Alexander Rosenberg Johansen. "Neural Arithmetic Units". In: *ICLR*. 2020.

[86] Alex Graves, Greg Wayne, and Ivo Danihelka. "Neural Turing Machines". In: *arXiv preprint arXiv:1410.5401* (2014).

[87] Alex Graves et al. "Hybrid computing using a neural network with dynamic external memory". In: *Nature* 538 (2016), pp. 471–476.

[88] Mark Collier and Joeran Beel. "Implementing Neural Turing Machines". In: *ICANN*. 2018, pp. 94–104.

[89] Georg Martius and Christoph H. Lampert. "Extrapolation and learning equations". In: *arXiv e-prints*, arXiv:1610.02995 (Oct. 2016), arXiv:1610.02995. arXiv: 1610.02995 [cs.LG].

[90] Subham Sahoo, Christoph Lampert, and Georg Martius. "Learning Equations for Extrapolation and Control". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR, Oct. 2018, pp. 4442–4450. URL: http://proceedings.mlr.press/v80/sahoo18a.html.

[91] Samuel Kim, Peter Y. Lu, Srijon Mukherjee, Michael Gilbert, Li Jing, Vladimir Čeperić, and Marin Soljačić. "Integration of Neural Network-Based Symbolic Regression in Deep Learning for Scientific Discovery". In: *IEEE Transactions on Neural Networks and Learning Systems* (2020), pp. 1–12.

[92] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". In: *CVPR*. 2016, pp. 770–778.

[93] Julia Balla, Sihao Huang, Owen Dugan, Rumen Dangovski, and Marin Soljacic. "AI-Assisted Discovery of Quantitative and Formal Models in Social Science". In: *arXiv preprint arXiv:2210.00563* (2022).

[94] Martin Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010.

[95] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. "Densely connected convolutional networks". In: *CVPR*. 2017.

[96] William La Cava, Lee Spector, and Kourosh Danai. "Epsilon-Lexicase Selection for Regression". In: *GECCO*. 2016.

[97] Silviu-Marian Udrescu, Andrew Tan, Jiahai Feng, Orisvaldo Neto, Tailin Wu, and Max Tegmark. "AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity". In: 2020.

[98] Stefan Wagner et al. "Advanced Methods and Applications in Computational Intelligence". In: vol. 6. Springer, 2014. Chap. Architecture and Design of the HeuristicLab Optimization Environment, pp. 197–261.

[99] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-Based Learning Applied to Document Recognition". In: *IEEE*. 1998.

[100] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR*. 2009.

[101] Randal S. Olson, William La Cava, Patryk Orzechowski, Ryan J. Urbanowicz, and Jason H. Moore. "PMLB: a large benchmark suite for machine learning evaluation and comparison". In: *BioData Mining* 10.1 (2017), p. 36.

[102] Michael Schmidt and Hod Lipson. "Symbolic Regression of Implicit Equations". In: *Genetic Programming Theory and Practice VII*. Ed. by Rick Riolo, Una-May O'Reilly, and Trent McConaghy. Springer US, 2010, pp. 73–85.

[103] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Mach. Learn.* 8.3–4 (May 1992), pp. 229–256.

[104] Diederik P. Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *ICLR*. 2015.

[105] Aaron Meurer et al. “SymPy: symbolic computing in Python”. In: *Peer J Computer Science* 3 (2017), p. 103.

[106] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *arXiv e-prints*, arXiv:1603.02754 (Mar. 2016), arXiv:1603.02754. arXiv: 1603.02754 [cs.LG].

[107] Patryk Orzechowski, William La Cava, and Jason H. Moore. “Where are we now? A large benchmark study of recent symbolic regression methods”. In: *GECCO*. 2018.

[108] Ramamurti Shankar. *Principles of quantum mechanics*. Springer Science & Business Media, 2012.

[109] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. “DEAP: Evolutionary Algorithms Made Easy”. In: *Journal of Machine Learning Research* 13 (2012), pp. 2171–2175.

[110] Vinicius V. Melo, Danilo Vasconcellos Vargas, and Wolfgang Banzhaf. “Batch Tournament Selection for Genetic Programming”. In: *GECCO*. 2019, arXiv:1904.08658.

[111] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. “Neural Random-Access Machines”. In: *ICLR*. 2016.

[112] David J. Montana and Lawrence Davis. “Training Feedforward Neural Networks Using Genetic Algorithms”. In: *IJCAI*. Morgan Kaufmann Publishers Inc., 1989.

[113] Tim Salimans, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. “Evolution Strategies as a Scalable Alternative to Reinforcement Learning”. In: *arXiv preprint arXiv:1703.03864* (2017).

[114] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. “The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables”. In: *ICLR*. 2017.

[115] Eric Jang, Shixiang Gu, and Ben Poole. “Categorical Reparameterization with Gumbel-Softmax”. In: *ICLR*. 2017.

[116] George Tucker, Andriy Mnih, Chris J. Maddison, and Jascha Sohl-Dickstein. “REBAR: Low-variance, unbiased gradient estimates for discrete latent variable models”. In: *NIPS*. 2017.

[117] Edouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou. “Efficient softmax approximation for GPUs”. In: *ICML*. 2017.

[118] Song Han, Jeff Pool, John Tran, and William J. Dally. “Learning both Weights and Connections for Efficient Neural Networks”. In: (2015).

[119] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. “Pruning Filters for Efficient ConvNets”. In: *ICLR*. 2017.

[120] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. “Variational Dropout Sparsifies Deep Neural Networks”. In: *ICML*. 2017.

[121] Christos Louizos, Max Welling, and Diederik P. Kingma. “Learning Sparse Neural Networks through L_0 Regularization”. In: *ICLR*. 2018.

[122] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362.