frontiers | Frontiers in Robotics and AI

TYPE Original Research
PUBLISHED 07 November 2022
DOI 10.3389/frobt.2022.968305

Check for updates

OPEN ACCESS

EDITED BY
Marcello Romano,
Naval Postgraduate School,
United States

REVIEWED BY
Pitoyo Hartono,
Chukyo University, Japan
Muhammad Khan,
Atılım University, Turkey

*CORRESPONDENCE
Frances Zhu,
zhuf@hawaii.edu

SPECIALTY SECTION
This article was submitted to Space Robotics,
a section of the journal
Frontiers in Robotics and AI

RECEIVED 13 June 2022
ACCEPTED 05 August 2022
PUBLISHED 07 November 2022

CITATION
Zhu F, Jing D, Leve F and Ferrari S (2022), NN-Poly: Approximating common neural networks with Taylor polynomials to imbue dynamical system constraints.
Front. Robot. AI 9:968305.
doi: 10.3389/frobt.2022.968305

NN-Poly: Approximating common neural networks with Taylor polynomials to imbue dynamical system constraints

Frances Zhu[1]*, Dongheng Jing[2], Frederick Leve[3] and Silvia Ferrari[2]

[1]Hawaii Institute of Geophysics and Planetology, University of Hawaii, Honolulu, HI, United States, [2]Sibley School of Mechanical and Aerospace Engineering, Cornell University, Ithaca, NY, United States, [3]Air Force Office of Scientific Research, Arlington, VA, United States

Recent advances in deep learning have bolstered our ability to forecast the evolution of dynamical systems, but common neural networks do not adhere to physical laws, critical information that could lead to sounder state predictions. This contribution addresses this concern by proposing a neural network to polynomial (NN-Poly) approximation, a method that furnishes algorithmic guarantees of adhering to physics while retaining state prediction accuracy. To achieve these goals, this article shows how to represent a trained fully connected perceptron, convolution, and recurrent neural networks of various activation functions as Taylor polynomials of arbitrary order. This solution is not only analytic in nature but also least squares optimal. The NN-Poly system identification or state prediction method is evaluated against a single-layer neural network and a polynomial trained on data generated by dynamic systems. Across our test cases, the proposed method maintains minimal root mean-squared state error, requires few parameters to form, and enables model structure for verification and safety. Future work will incorporate safety constraints into state predictions, with this new model structure and test high-dimensional dynamical system data.

KEYWORDS

neural networks, safety, interpretability, polynomial, dynamic systems, prediction

1 Introduction

Neural networks have emerged as general-purpose regression models and revolutionized fields such as computer vision and machine translation, where occasional errors are less likely to jeopardize human lives. To extend deep neural networks (NNs) to dynamic system applications of high consequence while retaining predictive capabilities, neural networks should be verifiable and conform to physical laws. The original dynamical system adheres to physical laws but the exact form of the dynamical system's transition model is not known. A neural network excels in learning a representation of the transition model but this representation can disagree with physical laws, even within the bounds of training as the learned features are agnostic/uninformed of the physical laws but particularly outside bounds of training due to lack of generalization. Furthermore, measurement (input) uncertainty or disturbance gives rise to a state prediction that is inaccurate; garbage in—garbage out. For example, if a camera obscurant leads to an incorrect state estimate (not propagating position) while other sensors continue to propagate, a physically uninformed model will incorporate anomalous sensor reading in generating a physically infeasible state prediction. Restructuring a neural network or transforming the NN to a different abstraction could ensure that the model output adheres to physical laws as the original system that generated the data does.

An ideal, interpretable solution leverages the power of neural networks and incorporates physics through 1) trust: safety guarantees in the output prediction and algorithm behavior, 2) causality: deriving relationships between the input and the output, and 3) information: analyzing the abstraction learned from the neural network and inferring system characteristics from the learned parameters and relationships (Lipton, 2018). Trust, or safety, can come in the form of adhering to constraints set by the user to mitigate the impact of a disturbance spike in the input signal on the model prediction. Added domain information in the way of physical laws could counter undesirable behavior. Furthermore, adding constraints has the added benefit that the user imposing some user-defined information into the prediction; that is, there is some component of that system that the user can explain and guarantee in behavior. Causality may be captured by tracing the contribution of an input to output, for which causality in a set of linear equations is very straightforward to correlate contribution from the coefficient matrix in analysis, whereas nesting layers within a neural network is less obvious in drawing input/output casual relationships (saliency maps). Information can include the number of parameters/terms in a model, the familiarity of bases, and the existence of another abstraction: "Computing a tractable function model from the original model can also be viewed as a form of knowledge distillation from the verification perspective, as the function model should be able to produce comparable results or replicate the outputs of the target neural network on specific inputs" (Huang et al., 2019).

This article's contribution is to solve the system identification or state prediction problem by constructing a mapping from an NN function to a polynomial function from which users can more easily infer behavior (information and causality) and handle changes in constraints (trust). The proposed method approximates a trained NN function of various common architectures (fully connected perceptron, convolution, and recurrent) into a system of linear differential equations in polynomial basis space. As neural networks are universal approximators, the derivation assumes that the general NN structure approximates the system mapping well (Hornik et al., 1989; Pinkus, 1999). The final form of the representation used to approximate the NN is a set of matrix equations with polynomial entries. Constraints or invariant quantities derived from physical laws may then be applied to offer safety and viability to the system dynamics. The polynomial approximation of neural networks is a straightforward mapping, executes in real-time, requires minimal data storage, and limits overfitting by limiting the polynomial expression power. Both the neural network and polynomial abstraction increase the interpretability of the dynamic system that generated the data.

Linear models, the simplest polynomial, are a favored abstraction over neural networks by scientists for their familiarity and analytic traceability. A polynomial basis is capable of expressing nonlinear relationships with a linear model. Polynomials are more computationally tractable, which enables verification (Sidrane et al., 2022). Their analytic traceability enables theoretical guarantees, inference rules (Dutta et al., 2019), and analysis (Lyapunov stability and coefficient stability). In polynomial space, users may apply safety criteria or domain knowledge in the context of constraints, invariant physical quantities, input/output relations, and continuity or bounded sensitivity properties (Narasimhamurthy et al., 2019). The polynomial form appears in many physical contexts (energy, heat transfer, and friction), which may be applied to the approximation as domain knowledge or safety constraints, and could inform scientists of the underlying physical system characteristics.

There is value in obtaining both the universal (NN) approximation and a polynomial representation. Polynomials and NNs with polynomial activation functions are not universal approximators like NNs with sigmoid or tanh activation functions are (Hornik et al., 1989). A generic dataset generated by a dynamic system does not guarantee a sum of squares polynomial solution, which is why a neural network universal approximation is necessary in capturing the transition model from the dynamic data (Ahmadi and Parrilo, 2011). Deep NNs train a non-convex optimization problem, where the constraints do not usually make the solution more tractable as the training protocol typically uses a stochastic gradient search in the first place. Turning the mapping for the dynamics into a polynomial imposes the dynamics as a constraint to be semi-algebraic (i.e., $p(x) > = 0$ or $p(x) = 0$, including many other polynomial constraints). Semi-algebraic optimization has new powerful results (e.g., sum-of-squares and moment sequences) that allow the highly non-convex optimization program to be converted to an iterative set of semi-definite programs (SDPs) that converge to the global minima. This article's optimization solution provides the initial mapping for the dynamics represented as a polynomial constraint. Tools, like semi-definite programs, make such optimization problems tractable.

The proposed method advances upon previous methods (Ferrari et al., 2013; Ferrari and Stengel, 2005) by deriving a polynomial abstraction of a trained neural network that is capable of incorporating knowledge and constraints into state prediction by solving the end-to-end polynomial function and constraints simultaneously. Outputs, like a state prediction, can be constrained to adhere to guarantees if the function is polynomial form, whereas neural networks traditionally lack that ability (Rudy et al., 2017; Lagaris et al., 1998; Psichogios and Ungar, 1992). Recently, a class of neural networks ingrain physics directly into the neural network structure, like PINNs (Raissi et al., 2019; Wang et al., 2022), Lagrangian/Hamiltonian NNs (Cranmer et al., 2020; Greydanus et al., 2019), neural ODEs (Djeumou et al., 2022), and deep Markov models (Liu et al., 2022). A difference in our proposed work lies in when the imposition of physics occurs: physics-guided NNs during training and NN-Poly post-training. Our work also differs from strictly learning a polynomial directly from data as we learn a polynomial from a neural network. Furthermore, the polynomial that results from this approach does not have to adhere to the properties of a Lyapunov function as is the case in learning homogenous polynomial Lyapunov functions, in that $V(x)$ does not need to be strictly positive and $\dot{V}(x)$ does not need to be strictly negative, where V is traditionally a function of energy (Ahmadi and Parrilo, 2011). While for some state definitions and systems, this form may be the most convenient form that is ultimately used, we do not impose that form in our formulation. The constraint equations that are used may not use energy and may use momentum or distance.

The following sections detail the derivation to approximate a neural network model by transforming the trained NN parameters into coefficients of polynomial form. The derivation consists of four general steps detailed in Sections 2–5:

- Section 2 derives a general Taylor series expansion for a vector function in vector domain (i.e., tensor form) from a trained NN model; higher than second-order derivatives are tensor derivatives.
- Section 3 simplifies tensor derivatives and states to matrix and vector form. An important contribution of this article is unfolding the tensor derivatives into a matrix form and the tensor states into a vector form, which results in matrix manipulability and computation savings.
- Section 4 rewrites the general Taylor series expansion, containing tensor derivatives into Taylor series expansion with only matrix and vector form derivatives. The Taylor series expansion with only matrix and vector form derivatives is desirable because modern scientific programming languages are optimized for vectorized computations. The Taylor series expansion can be presented as an expression in a polynomial form, mapping inputs to outputs. Coefficients of each polynomial entry are derived, and the subsequent dynamic system that the NN approximates can be interpreted.
- Section 5 extends the single-layer polynomial approximation to a multi-layer network, resulting in a polynomial that represents an arbitrarily deep network.

The remaining article sections give context as to how to apply this methodology and show results for simulated dynamic systems.

- Section 6 relates physical constraints for Newtonian dynamics to semi-algebraic constraints that can be applied to the output of the polynomial approximated function.
- Section 7 demonstrates how to solve for state prediction simultaneously with the proposed constraints.
- Section 8 analyzes the proposed NN-extracted polynomial method under various cases. Results show high accuracy and efficiency when dealing with test cases; we discuss how to extend NN-Poly to various other cases.

2 Problem formulation: Taylor expansion of a neural network

Representing the NN model as a Taylor polynomial involves two steps. First, a Taylor expansion of vector function $\boldsymbol{f}$ must be derived in state vector domain $\boldsymbol{x}$. The input to the NN model is the state vector at some time step, $\boldsymbol{x}_k$. The output of the NN model is either the state vector at the next time step, $\boldsymbol{x}_{k+1}$, or the derivative of the state vector, $\dot{\boldsymbol{x}}_k$. The expansion includes various dimensions of tensors and redundant multinomial cross terms. Next, the general derivatives of a neural network are derived in tensor form. The two efforts together produce the final polynomial coefficients for the Taylor polynomial. To validate the method and offer context for other methods, the last section offers a comparison of model fidelity and computation to other numerical system identification methods.

Given a vector input $\boldsymbol{x} \in R^{m\times 1}$ and output $\boldsymbol{y} \in R^{n\times 1}$, a function $\boldsymbol{f}$ maps the input state to the output state $\boldsymbol{f}(\boldsymbol{x})$: $R^{m\times 1} \to R^{n\times 1}$. Assume the function $\boldsymbol{f}(\cdot)$ is a smooth, continuous vector function, where all derivatives with respect to $\boldsymbol{x}$ exist and are smooth.

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) \tag{1}$$

Given the training pairs $(\boldsymbol{x}, \boldsymbol{y})$, a neural network predicts $\hat{\boldsymbol{y}}_{NN}$ with a mapping $\boldsymbol{f}_{NN}(W, \boldsymbol{b}, \boldsymbol{x})$, given in Eq. 2 where the notation of a hat $\widehat{(\cdot)}$ signifies a prediction of the variable. The learned parameters are W and $\boldsymbol{b}$ in the neural network.

$$\hat{\boldsymbol{y}}_{NN} = \boldsymbol{f}_{NN}(W, \boldsymbol{b}, \boldsymbol{x}) \tag{2}$$

A polynomial $\boldsymbol{p}(\boldsymbol{x})$ in the form of a Taylor expansion approximates the neural network for which the order of the polynomial d that approximates the NN is defined by the NN structure (Rolnick and Tegmark, 2017). The polynomial output $\hat{y}_p$ is given in Eq. 3, where the expression consists of polynomial coefficients \{a_0, A^1, . . ., A^d\}.

$$\hat{y}_p \coloneqq \boldsymbol{p}(\boldsymbol{x}) = a_0 + A^1 \odot \boldsymbol{x} + \frac{1}{2!}A^2 \odot \boldsymbol{x}^{2\otimes} + \cdots + \frac{1}{d!}A^d \odot \boldsymbol{x}^{d\otimes} \tag{3}$$

The goal is to find coefficients a of a polynomial expression that minimizes the error relative to the neural network model, defined by the cost function, C, given in Eq. 4 where the coefficients are the set $a = \{a_0, A^1, \ldots, A^d\}$.

$$C^* = \underset{a}{\operatorname{argmin}} \|\boldsymbol{f}_{NN}(W, \boldsymbol{b}, \boldsymbol{x}) - \boldsymbol{p}(\boldsymbol{x}, a, d)\| \tag{4}$$

The polynomial form is a terminal form of the Taylor expansion, given in Eq. 5, where the kth partial derivative of function f is given by $\frac{\partial^k f}{\partial x^k}$, analogously the Jacobian term, J_f^k. The polynomial terminates at order d, and the remaining higher order terms are captured in $\boldsymbol{R}(\boldsymbol{x})$.

$$\begin{aligned}\boldsymbol{f}(\boldsymbol{x}) &= \boldsymbol{f}(\boldsymbol{0}) + \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} \odot \boldsymbol{x} + \frac{\partial^2 \boldsymbol{f}}{\partial \boldsymbol{x}^2} \odot \boldsymbol{x}^{2\otimes} + \cdots + \boldsymbol{R}(\boldsymbol{x}) \\ &= \boldsymbol{f}(\boldsymbol{0}) + J_f^1(\boldsymbol{0})\boldsymbol{x} + \frac{1}{2!}J_f^2(\boldsymbol{0}) \odot \boldsymbol{x}^{2\otimes} + \cdots + \boldsymbol{R}(\boldsymbol{x})\end{aligned} \tag{5}$$

The outer product $\otimes$ is used in this manuscript to exponentiate a vector, for which an exponentiated vector equation example is given in Eq. 6 and the index notation in Eq. 7, adopted from Granados (2015).

$$\boldsymbol{x}^{3\otimes} \coloneqq \boldsymbol{x} \otimes \boldsymbol{x} \otimes \boldsymbol{x} \tag{6}$$

$$(\boldsymbol{x} \otimes \boldsymbol{x} \otimes \boldsymbol{x})_{ijk} = x_i x_j x_k \tag{7}$$

The inner product $\odot$ is used in this manuscript to multiply the function derivatives with the exponentiated states, for which index notation is given in Eq. 8, adopted from Granados (2015).

$$J_f^2 \odot \boldsymbol{x}^{2\otimes} = \sum_i \sum_j J_{ijk}^2 \boldsymbol{x}_{ij}^{2\otimes} \tag{8}$$

The polynomial expression is a vectorial series defined with Jacobian terms and exponentiated state vector terms in Eq. 9, where the expansion is expressed in a summation over $d + 1$ Jacobian terms. For simplicity, the expansion is about the zero state, assuming $\boldsymbol{x}_0 = 0$, which simplifies the lower dimension terms. Note the equivalency of Eqs 3, 9.

$$\begin{aligned}\boldsymbol{p}(\boldsymbol{x}) &= \sum_{k=0}^{d} \frac{J_f^k(\boldsymbol{x}_0)}{k!} \odot (\boldsymbol{x} - \boldsymbol{x}_0)^{k\otimes} \\ &\coloneqq \sum_{k=0}^{d} \frac{1}{k!} A^k \odot (\boldsymbol{x} - \boldsymbol{x}_0)^{k\otimes}\end{aligned} \tag{9}$$

Although Eq. 5 is elegant and may be able to derive a closed-form expression for the original function, the subsequent derivative terms incrementally increase in dimension, seen in Eq. 10.

$$\begin{aligned}\boldsymbol{f}(\boldsymbol{x}) = &\begin{bmatrix} f_1(\boldsymbol{x_0}) \\ f_2(\boldsymbol{x_0}) \\ \vdots \\ f_n(\boldsymbol{x_0}) \end{bmatrix} + \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \odot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \\ &+ \frac{1}{2!} \left[\begin{bmatrix} \frac{\partial^2 f_1}{\partial x_1^2} & \frac{\partial^2 f_1}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f_1}{\partial x_m \partial x_1} \\ \frac{\partial^2 f_2}{\partial x_1^2} & \frac{\partial^2 f_2}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f_2}{\partial x_m \partial x_1} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f_n}{\partial x_1^2} & \frac{\partial^2 f_n}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f_n}{\partial x_m \partial x_1} \end{bmatrix}, \ldots, \begin{bmatrix} \frac{\partial^2 f_1}{\partial x_1 \partial x_m} & \frac{\partial^2 f_1}{\partial x_2 \partial x_m} & \cdots & \frac{\partial^2 f_1}{\partial x_m^2} \\ \frac{\partial^2 f_2}{\partial x_1 \partial x_m} & \frac{\partial^2 f_2}{\partial x_2 \partial x_m} & \cdots & \frac{\partial^2 f_2}{\partial x_m^2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f_n}{\partial x_1 \partial x_m} & \frac{\partial^2 f_n}{\partial x_2 \partial x_m} & \cdots & \frac{\partial^2 f_n}{\partial x_m^2} \end{bmatrix} \right] \odot \begin{bmatrix} x_1^2 & x_1 x_2 & \cdots & x_1 x_m \\ x_2 x_1 & x_2^2 & \cdots & x_2 x_m \\ \vdots & \vdots & \ddots & \vdots \\ x_m x_1 & x_m x_2 & \cdots & x_m^2 \end{bmatrix} . \\ &+ \cdots + \boldsymbol{R}(\boldsymbol{x})\end{aligned} \tag{10}$$

Instead, we would like a set of linear equations that simply solves for the polynomial coefficients with a single matrix operation, given in Eq. 11. To transform the various tensors into a set of linear equations, the derivative tensor terms must be unfolded and compressed into matrices and vectors to achieve the desired form, given in Eq. 11, described in the following sections.

$$\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} f_1(\boldsymbol{x_0}) & \left(\frac{\partial f_1}{\partial x_1}\right) & \cdots & \left(\frac{\partial f_1}{\partial x_m}\right) & \left(\frac{\partial^2 f_1}{\partial x_1^2}\right) & \cdots & \left(\frac{\partial^2 f_1}{\partial x_1 \partial x_m}\right) & \cdots & \left(\frac{\partial^2 f_1}{\partial x_m^2}\right) & \cdots & \left(R_1\left(\boldsymbol{x}^{k\otimes}\right)\right) \\ & & & & & \vdots & & & & & \\ f_n(\boldsymbol{x_0}) & \left(\frac{\partial f_n}{\partial x_1}\right) & \cdots & \left(\frac{\partial f_n}{\partial x_m}\right) & \left(\frac{\partial^2 f_n}{\partial x_1^2}\right) & \cdots & \left(\frac{\partial^2 f_n}{\partial x_1 \partial x_m}\right) & \cdots & \left(\frac{\partial^2 f_n}{\partial x_m^2}\right) & \cdots & \left(R_n\left(\boldsymbol{x}^{k\otimes}\right)\right) \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_m \\ \frac{1}{2!}x_1^2 \\ \vdots \\ \frac{2}{2!}x_1 x_m \\ \vdots \\ \frac{1}{2!}x_m^2 \\ \vdots \\ 1 \end{bmatrix} \tag{11}$$

3 Unfolding and compressing tensors into matrices and vectors

The tensor equation that approximates the original function may be collapsed into a set of linear equations, consisting of a coefficient matrix and state vector, a useful form for a linear least square solution or semi-algebraic optimization. Exponentiated states of order higher than two are in the form of tensors and contain redundant multinomial terms as they are symmetric. By analogy, the upper triangular part of a symmetric matrix contains all of its unique values. To unfold and reshape the high dimensional tensors to matrices and vectors, the tensor indices of each higher dimension has a relationship to a single index in the vector and the matrix. The derivation may be intuited visually. The columns of the upper

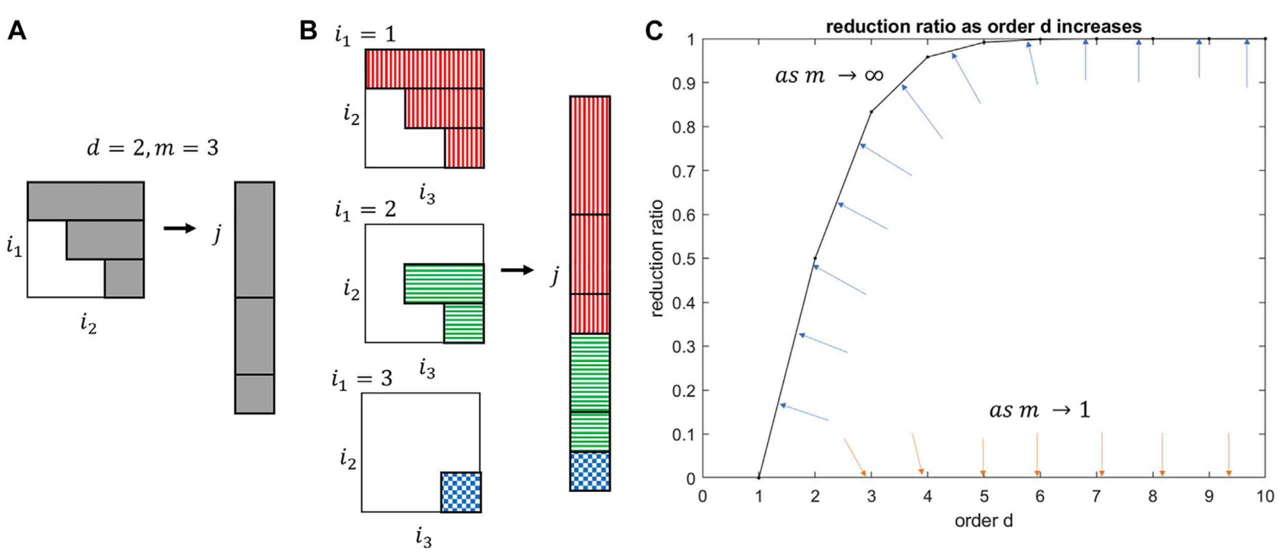

FIGURE 1
(A) Unfolding an upper triangular matrix into a vector. **(B)** Unfolding a tensor into a vector. **(C)** Reduction ratio between number of reductions and number of full terms. The blue and orange arrows represent the movement of the reduction ratio curve at two extremes: blue as the state vector dimension approaches infinity; orange as the state vector dimension approaches one.

triangular matrix are sequentially appended to a vector, seen in Figure 1A. This process is extended to symmetric tensors, Figure 1B, to yield the augmented state vector.

By grouping redundant multinomial cross terms together into a coefficient vector, the input vector contains only the unique states. The first step is gathering redundant state terms in the same equation, highlighted (Eq. 12). The desired matrix equation, (Eq. 11), of the tensor equation yields the same set of linear equations, yet reduces the number of total state terms and is in a more useful representation. The augmented state vector in Eq. 11 contains scalar coefficients, which represent the number of redundant terms in the multinomial expansion of the state. These scalar coefficients are separated into a new coefficient vector. Defining this new coefficient vector is necessary to reform the exponentiated states into a state vector of only unique states and in the process reveals the reduction in computation by using only unique states. For intuition, the unique multinomial state vector of third order is $\tilde{\boldsymbol{x}}^{\otimes 3}$, given in Eq. 13, where the addition of the tilde annotates that the uniqueness operation has been performed on the original state vector $\boldsymbol{x}^{\otimes 3}$. The corresponding multinomial coefficient vector, $\boldsymbol{a}^3$, contains respective coefficients, representing the combinatorial number of redundant multinomial states, given in Eq. 14.

$$
\boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix}
f_1(\boldsymbol{x_0}) + \left(\frac{\partial f_1}{\partial x_1}x_1 + \frac{\partial f_1}{\partial x_2}x_2 + \cdots + \frac{\partial f_1}{\partial x_m}x_m\right) + \cdots \\
+ \frac{1}{2!}\left[\left(\frac{\partial^2 f_1}{\partial x_1^2}x_1^2 + \frac{\partial^2 f_1}{\partial x_2 \partial x_1}x_2 x_1 + \cdots + \frac{\partial^2 f_1}{\partial x_m \partial x_1}x_m x_1\right) + \cdots \right. \\
\left. + \left(\frac{\partial^2 f_1}{\partial x_1 \partial x_m}x_1 x_m + \frac{\partial^2 f_1}{\partial x_2 \partial x_m}x_2 x_m + \cdots + \frac{\partial^2 f_1}{\partial x_m^2}x_m^2\right) + \cdots + R_1\left(\boldsymbol{x}^{k\otimes}\right)\right. \\
f_2(\boldsymbol{x_0}) + \left(\frac{\partial f_2}{\partial x_1}x_1 + \frac{\partial f_2}{\partial x_2}x_2 + \cdots + \left(\frac{\partial f_1}{\partial x_m}x_m\right) + \cdots \right. \\
+ \frac{1}{2!}\left[\left(\frac{\partial^2 f_2}{\partial x_1^2}x_1^2 + \frac{\partial^2 f_2}{\partial x_2 \partial x_1}x_2 x_1 + \cdots + \frac{\partial^2 f_2}{\partial x_m \partial x_1}x_m x_1\right) + \cdots \right. \\
\left. + \left(\frac{\partial^2 f_2}{\partial x_1 \partial x_m}x_1 x_m + \left(\frac{\partial^2 f_2}{\partial x_2 \partial x_m}x_2 x_m + \cdots + \frac{\partial^2 f_2}{\partial x_m^2}x_m^2\right) + \cdots + R_2\left(\boldsymbol{x}^{k\otimes}\right)\right. \\
\vdots \\
f_n(\boldsymbol{x_0}) + \left(\frac{\partial f_n}{\partial x_1}x_1 + \frac{\partial f_n}{\partial x_2}x_2 + \cdots + \frac{\partial f_n}{\partial x_m}x_m\right) + \cdots \\
+ \frac{1}{2!}\left[\left(\frac{\partial^2 f_n}{\partial x_1^2}x_1^2 + \frac{\partial^2 f_n}{\partial x_2 \partial x_1}x_2 x_1 + \cdots + \frac{\partial^2 f_n}{\partial x_m \partial x_1}x_m x_1\right) + \cdots \right. \\
\left. + \left(\frac{\partial^2 f_n}{\partial x_1 \partial x_m}x_1 x_m + \frac{\partial^2 f_n}{\partial x_2 \partial x_m}x_2 x_m + \cdots + \frac{\partial^2 f_n}{\partial x_m^2}x_m^2\right) + \cdots + R_n\left(\boldsymbol{x}^{k\otimes}\right)\right.
\end{bmatrix} \tag{12}
$$

$$
\tilde{\boldsymbol{x}}^{\otimes 3} = \left[\, x_1^3,\ x_1^2 x_2, \cdots, x_1^2 x_m,\ x_1 x_2^2, \cdots, x_1 x_2 x_m, \cdots, x_1 x_m^2,\ x_2^3,\ x_2^2 x_3, \cdots, x_2^2 x_m,\ x_2 x_m^2, \cdots, x_m^3 \,\right] \tag{13}
$$

$$
\begin{aligned}
\boldsymbol{a}^3 &= \frac{1}{3!}\left[\binom{3}{3}\ \binom{3}{2,1}\ \cdots\ \binom{3}{2,1}\ \binom{3}{1,2}\ \cdots\ \binom{3}{1,1,1}\ \cdots\ \binom{3}{2,1}\ \binom{3}{3}\ \binom{3}{2,1}\ \cdots\ \binom{3}{2,1}\ \binom{3}{1,2}\ \cdots \binom{3}{3}\right] \\
&= \frac{1}{3!}\left[\,1 \quad 3 \quad \cdots \quad 3 \quad 3 \quad \cdots \quad 6 \quad \cdots \quad 3 \quad 1 \quad 3 \quad \cdots \quad 3 \quad 3 \quad \cdots \quad 1\,\right]
\end{aligned} \tag{14}
$$

This augmented state vector is the unique multinomial state vector, $\tilde{\boldsymbol{x}}^{\otimes d}$, which is $\tilde{\boldsymbol{x}}$ exponentiated to degree d in Eq. 15, where j is the vector term index and $i_1, i_2, \ldots, i_d$ are the tensor dimension indices.

$$\begin{aligned}\tilde{x}^{\otimes d}(j) &= x^{\otimes d}(i_1, i_2, \ldots, i_d)\\ &= x_{i_1} x_{i_2} \cdots x_{i_d}\\ &\text{where}\quad j = i_1 + \frac{i_2(i_2-1)}{2} + \cdots + \prod_{k=1}^{d} \frac{i_d + k - 2}{k}\\ &\quad \text{for}\quad i_1 = 1: m,\\ &\quad \text{for}\quad i_2 = i_1: m,\\ &\quad \cdots,\\ &\quad \text{for}\quad i_{d-1} = i_{d-2}: m,\\ &\quad \text{for}\quad i_d = i_{d-1}: m \end{aligned} \tag{15}$$

$$n_d = \prod_{i=1}^{d} \frac{m + i - 1}{i} \tag{16}$$

This augmented state vector, $\tilde{x}^{\otimes d}$, is of size n_d, given in Eq. 16 and calculated with the multinomial theorem (Hildebrand, 2009). Unfolding tensors is a non-unique process, for which a different unfolding yields a different vector of coefficients. These solution sets are all minima for the same unfolding problem. Supplementary Appendix Section S10.1 gives the explicit description of augmented state vectors in ascending degree and corresponding multinomial coefficient vector.

The general solution for the multinomial coefficient vector $a^d(j)$ is given in Eq. 17, where the operator $(\cdot)$ is the binomial coefficient of choosing d states out of m total number of states and n_i is the number of individual x_i states in the multinomial state $\tilde{x}^{\otimes d}(j)$ (Hildebrand, 2009). The size of a^d also follows (Eq. 16). The multinomial coefficient vector index j in Eq. 17 aligns with the state j index in Eq. 17. Explicit coefficient definitions for ascending orders of $\tilde{x}^{\otimes d}(j)$ are given in Table 8 in Supplementary Appendix Section S10.1.

$$\begin{aligned} a^d(j) &= \frac{1}{d!}(d, n_1, n_2, , \ldots, n_m)\\ \text{for}\quad n_1 &= \mathcal{O}(x_1) \in \tilde{x}^{\otimes d}(j), \quad \cdots, \quad n_m = \mathcal{O}(x_m) \in \tilde{x}^{\otimes d}(j) \end{aligned} \tag{17}$$

The general solution for the Jacobian matrix $\tilde{J}^2_f(:, j)$ is given in Eq. 18. The index j of the modified Jacobian $\tilde{J}^d_f$ mapping follows the index j of the multinomial state vector $\tilde{x}^{\otimes d}$ mapping with one additional rule: the first dimension's index i_0 in the original Jacobian tensor J^d_f directly translates to the first dimension's index i_0 in the modified Jacobian matrix $\tilde{J}^d_f$. An example of the explicit definition of $\tilde{J}^2_f(:, j)$ can be found in Table 8 in Supplementary Appendix Section S10.1. Note that for $d \le m$, i_1 to i_{d-m+1} does not increment in index but stays at index 1 as dummy dimensions for the multinomial state, coefficient, and Jacobian derivations.

$$\begin{aligned}\tilde{J}^d_f(i_0, j) &= J^d_f(i_0, i_1, i_2, \ldots, i_d)\\ \text{for}\quad j &= i_1 + \frac{i_2(i_2-1)}{2} + \cdots + \prod_{k=1}^{d} \frac{i_d + k - 2}{k}\\ \text{for}\quad i_0 &= 1: n\\ \text{for}\quad i_1 &= 1: m,\\ \text{for}\quad i_2 &= i_1: m,\\ \cdots&,\\ \text{for}\quad i_{d-1} &= i_{d-2}: m,\\ \text{for}\quad i_d &= i_{d-1}: m \end{aligned} \tag{18}$$

Compressing redundant state terms saves an immense amount of computation, especially for the number of states in real-world applications and for deriving approximations with more than two derivatives. The general reduction ratio, r, between the number of unique terms and the full expansion is given in Eq. 19.

$$r = 1 - \frac{\prod_{i=1}^{d} \frac{m+i-1}{i}}{m^d} \tag{19}$$

The reduction can be seen to approach 100% for large states and 0% for scalar domain, Figure 1C. The real reduction rate falls somewhere beneath the $m = \infty$ bounding curve and the $m = 1$ origin. The minimum number of states and derivatives to yield a significant computation reduction of 25% is already achieved at $m = 2$ states and $d = 2$ derivatives. Table 6 in Supplementary Appendix Section S10.1 illustrates how the reduction ratio scales with the derivative order, explicitly calculating the corresponding number of states for full state tensor expansion compared to the unique state terms in the augmented state vector $\tilde{x}^{\otimes d}$. Motivated by linear solutions and significant computational cost, a framework was derived such that the tensor derivatives and states in the Taylor expansion can be reshaped into a unique multinomial matrix and vectors, respectively.

4 Tensor derivatives of a neural network

This section derives different single-layer neural network's tensor derivatives J^d_f that populate the Taylor series. The tensor derivatives evaluated at the origin $J^d_f|_{x=0}$ are the coefficient tensors A^d. This section's derivation is the next step in the overall process of approximating a single-layer neural network, with a polynomial function. The derivative formulation covers a wide range of the most popular networks, classified into network type and activation function. Network layers contained in this section are feedforward, convolution, and recurrent layers. Activation functions include binary, max, linear, ReLU, softmax, sigmoid, tanh, and probabilistic, referenced in Table 1.

The tensor derivatives that approximate the neural network layer are dictated by the pairing of cell layer type and activation

TABLE 1 Activation function type by ascending complexity with their associated vector and index expressions.

Activation function name	Activation function vector expression	Activation function index expression
Binary		$y_j = \begin{cases} 1, & \text{if } \sum_{i=1}^{m} w_{ji} x_i + b_j \geq 0 \\ 0, & \text{otherwise} \end{cases}$
Max	$y = \max(W\boldsymbol{x} + \boldsymbol{b})$	$y_j = \max(w_{ji}x_i + b_j)$ for $i = 1, 2, \ldots, m$
Linear	$\boldsymbol{y} = W\boldsymbol{x} + \boldsymbol{b}$	$y_j = \sum_{i=1}^{m} w_{ji} x_i + b_j$
Softmax	$\boldsymbol{y} = \frac{e^{W\boldsymbol{x}+\boldsymbol{b}}}{e^{1\cdot(W\boldsymbol{x}+b)}}$	$y_j = \frac{e^{\sum_{i=1}^{m} w_{ji}x_i + b_j}}{\sum_{k=1}^{m} e^{\sum_{i=1}^{m} w_{ki}x_i + b_k}}$
Probabilistic	$y = e^{-\beta_i \lVert \boldsymbol{x} - \boldsymbol{c}_i \rVert}$	$y_j = e^{-\beta_i} \sqrt{\sum_{i=1}^{m} (x_i - c_{ij})^2}$

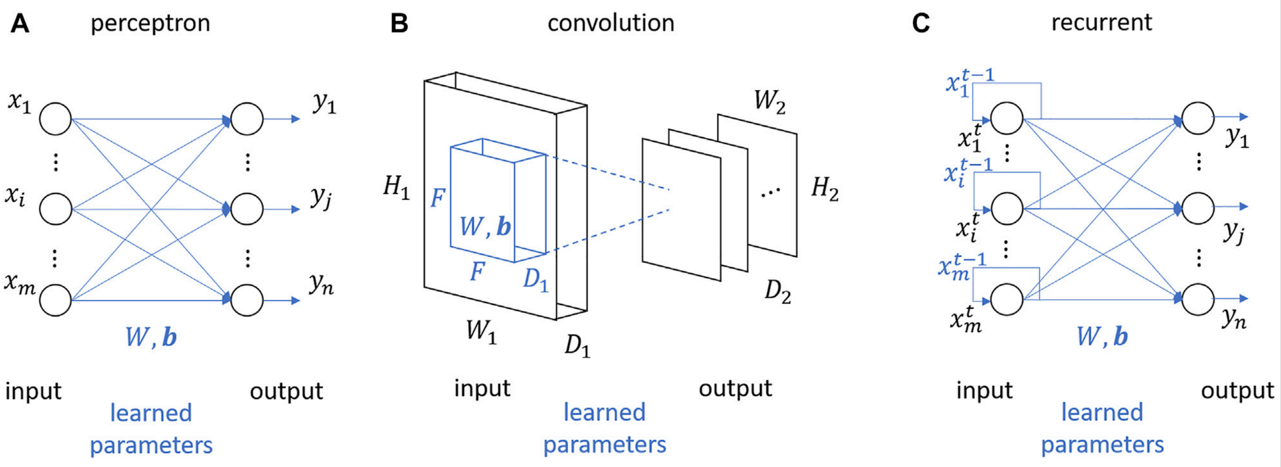

FIGURE 2
Basis single layer types transforming input to output: **(A)** perceptron layer, **(B)** convolution layer, and **(C)** recurrent layer.

function type; a feedforward layer with a ReLU activation has different tensor derivatives than a feedforward layer with a sigmoid activation. The following subsections vary the relevant transformations $\sigma(\cdot)$ and calculates ascending orders of different NN layer derivatives to then populate a Taylor approximation in tensor form (Eq. 5), and then explicitly in matrix form (Eq. 18). The layer types are illustrated in Figure 2.

4.1 Perceptron layer

A single-layer feedforward network with n number of neurons in the layer is depicted in Figure 2A. The output from a hidden layer $\boldsymbol{y}_F$ is a transformation $\boldsymbol{f}_F$ of the input $\boldsymbol{x}$, and a learned weight and bias matrix, W and $\boldsymbol{b}$. The input and output are in vector form, where $\boldsymbol{x}$ is of size $\mathbb{R}^{m\times 1}$ and $\boldsymbol{y}_F$ is of size $\mathbb{R}^{n\times 1}$. The vector equation is given in Eq. 20, where $\sigma(\cdot)$ is the activation function.

$$\boldsymbol{y}_F = \boldsymbol{f}_F(\boldsymbol{x}) = \sigma(W, \boldsymbol{b}, \boldsymbol{x}) \tag{20}$$

The index equation is given in Eq. 21, where the order of indices follows the dimension. The index, i, corresponds to a value in input state $\boldsymbol{x}$ and the index, j, corresponds to a value in output state $\boldsymbol{y}_F$.

$$y_j = f_j(x_i) = \sigma\left(w_{ji}, b_j, x_i\right) \tag{21}$$

For many activation functions, derivatives truncate in finite order, such as binary, linear, and ReLU functions. The coefficient tensors for these terminating derivatives are trivial solutions as their Taylor series are equivalent to their original function expressions. These derivations may be found in Supplementary Appendix Section S10.2. For continuously differentiable activation functions, like the sigmoid, tanh, softmax, and probabilistic, the Taylor approximation series is artificially truncated at a user-defined order d (Eq. 22).

$$
\begin{aligned}
y_j \approx a_j^0 + \sum_{i=1}^{m} A_{ji}^1 x_i + \frac{1}{2!}\sum_{i_1=1}^{m}\sum_{i_2=1}^{m} A_{ji_1i_2}^2 x_{i_1i_2}^{\otimes 2} + \cdots \\
+\frac{1}{d!}\sum_{i_1=1}^{m}\cdots\sum_{i_d=1}^{m} A_{ji_1\cdots i_d}^d x_{i_1\cdots i_d}^{\otimes d}
\end{aligned}
\tag{22}
$$

The higher order polynomial coefficient tensors are iteratively derived with immediately previous polynomial coefficient tensors. The general derivative relationships for each activation function are proved by induction from lower order derivatives. The general sigmoid derivative is given in Eq. 23 with the 0th to 2nd order terms given in Eqs 88–90 in Supplementary Appendix as proof of the iterative pattern.

$$
A_{ji_1\cdots i_d}^d = \frac{\partial A_{ji_1\cdots i_{d-1}}^{d-1}}{\partial \sigma} w_{ji_d}\sigma(b_j)\left(1-\sigma(b_j)\right) \tag{23}
$$

The tanh derivative is given by Eq. 24 with 0^{th} to 2^{nd} derivatives in Eqs 91–93 in the Supplementary Appendix.

$$
A_{ji_1\cdots i_d}^d = \frac{\partial A_{ji_1\cdots i_{d-1}}^{d-1}}{\partial \sigma} w_{ji_d}\left(1-\sigma^2(b_j)\right) \tag{24}
$$

The softmax derivative is given by Eq. 25 with 0^{th} to 1^{st} derivatives in Eqs 94, 95 in Supplementary Appendix.

$$
A_{ji_1\cdots i_d}^d = A_{ji_1\cdots i_{d-1}}^{d-1} w_{ji_d} \tag{25}
$$

Probabilistic activation functions come in many different forms of which the Gaussian or radial basis function is the most popular function, given in Table 1. The polynomial coefficients for this function are complicated expressions for which two intermediate expressions simplify and reveal iterative patterns more clearly in Eq. 26: α_t^D and s_t^D. Each polynomial coefficient tensor is a sum n_t terms composed of α_t^D and s_t^D, where t the increment of a subexpression and D is the term order. The number of terms n_t depends only on the term's degree order D, defined in Eq. 27.

$$
A_{ji_1\cdots i_d}^d = \sum_{t=1}^{n_t} \alpha_t^D s_t^D \tag{26}
$$

$$
n_t = \left\lceil \frac{D+1}{2} \right\rceil \tag{27}
$$

The most general terms, α_t^D and s_t^D, for any term and degree definition are in Eqs 28, 29, with explicit definitions in Table 9 in Supplementary Appendix Section S10.2 and 0th to 4th order terms are explicitly given in Eqs 99–103 in Supplementary Appendix.

$$
\alpha_t^D = \prod_{k=1}^{t-1}(2k-1)\left(\frac{-\beta_j}{\|\boldsymbol{c}_j\|}\right)^{t-1}\alpha_1^{D-t+1} \tag{28}
$$

$$
s_t^D = \prod_{l=1}^{D-2(t-1)}\left(\sum_{k=l}^{D}\frac{\beta_j c_{ji_k}}{\|\boldsymbol{c}_j\|}\right) \tag{29}
$$

4.2 Convolution layer

The convolution layer commonly uses two types of activation functions, linear and max, to achieve filtering and pooling. The derivatives of these activation functions truncate either on the 0th or 1st order derivative. The coefficient tensors when injected into the Taylor series yield exact representations of the original activation functions but offer standard indexing for matrix and vector conversion. A single-layer convolution layer with n filters of f spatial extent, s stride, and p zero padding is depicted in Figure 2B. Like the perceptron layer, the output from the hidden layer, Y, is a transformation of the input, X, and a learned weight and bias tensor, W and $\boldsymbol{b}$. Unlike the perceptron layer, the input and output are typically in the matrix or third order tensor form, where X is of size $\mathbb{R}^{w_1\times h_1\times d_1}$ and Y is of size $\mathbb{R}^{w_2\times h_2\times d_2}$. The tensor equation is given in Eq. 30.

$$
Y = f_C(X) = \sigma(W, \boldsymbol{b}, X) \tag{30}
$$

The tensor equation with indices is given in Eq. 31. The indices i, j, k for the output range from: $i = 1, 2, \ldots, w_2$, $j = 1, 2, \ldots, h_2$, and $k = 1, 2, \ldots, d_2$. The indices l, m, n range from: $l = 1, 2, \ldots, w_1$, $m = 1, 2, \ldots, h_1$, and $n = 1, 2, \ldots, d_1$.

$$
Y_{ijk} = \sigma\left(W_{ij}^k, b_k, X_{lmn}\right) \tag{31}
$$

The linear activation function within a convolution layer is often called a convolution filter and offers feature extraction of input data. The convolution filter function utilizing the convolution operator in matrix and index form is given in Eq. 32, where the definition of $\tilde{X}_{ij}$ is given in Eq. 33.

$$
Y_{ijk} = W^k \circledast \tilde{X}_{ij} + b_k \tag{32}
$$

$$
\tilde{X}_{ij} = X\left[s(i-1)-p+1 : s(i-1)-p+f, s(j-1)-p+1 : s(j-1)-p+f, 1 : d_1\right] \tag{33}
$$

The relationship between input and output sizes is defined by the network hyperparameters, reiterated here: number of filters n, height and width of the filter f, stride length s, and padding p, given in Eq. 34.

$$
w_2 = \frac{w_1 - f + 2p}{s} + 1, \quad h_2 = \frac{h_1 - f + 2p}{s} + 1, \quad d_2 = n \tag{34}
$$

The hyperparameters n, f, s, p are user-defined for which a common setting for the hyperparameters is: $f = 3$, $s = 1$ and $p = 1$. The constraint $p = (f-1)/2$ preserves input to output size. The learned parameters W is of size $\mathbb{R}^{f\times f\times n}$ and b is of size $\mathbb{R}^n$. The output equation in strictly index form is given in Eq. 35.

$$
Y_{ijk} = \sum_{l=1}^{f}\sum_{m=1}^{f}\sum_{n=1}^{d_1} W_{lmn}^k X_{l+s(i-1)-p, m+s(j-1)-p, n} + b_k \tag{35}
$$

The linear activation function is a continuously differentiable function with a Taylor approximation series that truncates after the 1st order term, given in Eq. 36.

$$Y_{ijk} = a^0_{ijk} + \sum_{l=1}^{w_1} \sum_{m=1}^{h_1} \sum_{n=1}^{d_1} A^1_{ijklmn} X_{lmn} \tag{36}$$

The 0th and 1st order terms are given in Eqs 37, 38. A^1_{ijklmn} is a sparse six dimensional tensor that performs an equivalent convolution operation with W^k. The Taylor approximation yields an exact representation to the original function expression despite the difference in operation representation.

$$a^0_{ijk} = f_{ijk}(X_{lmn} = 0) = b_k \tag{37}$$

$$\begin{aligned} A^1_{ijklmn} &= \left.\frac{\partial f_{ijk}}{\partial X_{lmn}}\right|_{X_{lmn}=0} \\ &= \begin{cases} W^k_{l+s(1-i)+p,m+s(1-j)+p,n}, & \text{if } 1 + s(i-1) + p \le l \le f + s(i-1) + p \\ & \text{and } 1 + s(j-1) + p \le m \le f + s(j-1) + p \\ 0, & \text{otherwise} \end{cases} \end{aligned} \tag{38}$$

The maximum activation function within a convolution layer is often called a pooling layer. A pooling layer typically follows a filter layer, offering translation invariance in terms of convolution filter output. The max function outputs the maximum value of the input that falls within the kernel. The matrix form is given in Eq. 39.

$$Y_{ijk} = \max_{i,j,k}\left(X\left[s(i-1)+1{:}\ s(i-1)+f, s(j-1)+1{:}\ s(j-1)+f, k\right]\right) \tag{39}$$

As there are no distinct kernels and no padding, the only hyperparameters are field size f and stride s for which the relationship between input and output sizes is defined in Eq. 40.

$$w_2 = \frac{w_1 - f}{s} + 1, \quad h_2 = \frac{h_1 - f}{s} + 1, \quad d_2 = d_1 \tag{40}$$

Common settings for the hyperparameters are $f = 2$ and $s = 2$. The Taylor approximation follows the same form as Eq. 36, which truncates after the 1st order term. The 0^{th} and 1^{st} order terms are given in Eqs 41, 42. The Taylor approximation yields an exact representation to the original function expression despite the difference in operation representation.

$$a^0_{ijk} = f_{ijk}(X_{lmn} = 0) = 0 \tag{41}$$

$$A^1_{ijklmn} = \begin{cases} 1, & \text{if } X_{lmn} = \max\left(\tilde{X}_{ijk}\right) \\ 0, & \text{otherwise} \end{cases} \tag{42}$$

4.3 Recurrent layer

The structure of a single-layer with n number of recurrent units in the layer is depicted in Figure 2C. The output from the convolution hidden layer, $\boldsymbol{y}^t_C$ or equivalently $\boldsymbol{s}^t$, is a transformation of the current time step's input, $\boldsymbol{x}^t$, the previous time step's internal state, $\boldsymbol{s}^{t-1}$, and a learned weight and bias matrices, W, U, and $\boldsymbol{b}$. The input, internal state, and output are typically in vector form, where $\boldsymbol{x}^t$ is of size $\mathbb{R}^{m\times 1}$ but $\boldsymbol{s}^{t-1}$ and $\boldsymbol{y}^t_C$ are of size $\mathbb{R}^{n\times 1}$. The weights are typically in matrix form, where W is of size $\mathbb{R}^{n\times n}$, U is of size $\mathbb{R}^{n\times m}$, and $\boldsymbol{b}$ is of size $\mathbb{R}^{n\times 1}$. The vanilla recurrent unit vector relationship is given in Eq. 43.

$$\boldsymbol{y} = f\left(\boldsymbol{s}^{t-1}, \boldsymbol{x}^t\right) = \sigma\left(W, \boldsymbol{s}^{t-1}, U, \boldsymbol{x}^t, \boldsymbol{b}\right) \tag{43}$$

The index relationship is given in Eq. 44, where the order of indices follows the dimension. The index i corresponds to a value in the internal state $\boldsymbol{s}^{t-1}$, the index k corresponds to a value in the input state $\boldsymbol{x}^t$, and the index j corresponds to a value in the output state $\boldsymbol{y}^t_C$.

$$y_j = f_j\left(s^{t-1}_i, x^t_k\right) = \sigma\left(w_{ji}, s^{t-1}_i, u_{jk}, x^t_k, b_j\right) \tag{44}$$

The explicit vanilla RNN vector equation and index equation are given in Eqs 45, 46, respectively.

$$\boldsymbol{y} = \sigma\left(W\boldsymbol{s}^{t-1} + U\boldsymbol{x}^t + \boldsymbol{b}\right) \tag{45}$$

$$y_j = \sigma\left(\sum_i w_{ji} s^{t-1}_i + \sum_k u_{jk} x^t_k + b_j\right) \tag{46}$$

The vanilla RNN vector equation can be reformed into a feed-forward layer with analogous tensor and coefficient derivations. The previous state $\boldsymbol{s}^{t-1}$ and the current input $\boldsymbol{x}^t$ may be joined together into one input $\boldsymbol{z}$, given in Eq. 47.

$$\boldsymbol{z}^t = \left[\boldsymbol{s}^{t-1} \quad \boldsymbol{x}^t\right] \tag{47}$$

The combined input $\boldsymbol{z}$ is of additive size $\mathbb{R}^{(n+m)\times 1}$ with associated dimension index l. The combined weight matrix, V, is given in Eq. 48 and is of size $\mathbb{R}^{n\times(n+m)}$.

$$V = [W \quad U] \tag{48}$$

The reformed state prediction vector equation and index equation are given in Eqs 49, 50.

$$\boldsymbol{y}^t_C = \sigma\left(V\boldsymbol{z}^t + \boldsymbol{b}\right) \tag{49}$$

$$y_j = \sigma\left(\sum_l v_{jl} z_l + b_j\right) \tag{50}$$

The similarity between the reformed equations of the recurrent layer and feed-forward layer is apparent. To be explicit, the modified Taylor series expansion about the reformed state $\boldsymbol{z}$ is given in Eqs 51, 52.

$$\boldsymbol{f}(\boldsymbol{z}) = \boldsymbol{f}(\boldsymbol{z_0}) + J^1_f(\boldsymbol{z_0})z + \frac{1}{2!}J^2_f(\boldsymbol{z_0}) \otimes z^{2\otimes} + \cdots + \boldsymbol{R_n}(\boldsymbol{z}) \tag{51}$$

$$y_j \approx a^0_j + \sum_{l=1}^{n+m} A^1_{jl} z_l + \frac{1}{2!}\sum_{l_1=1}^{n+m}\sum_{l_2=1}^{n+m} A^2_{jl_1l_2} z^{\otimes 2}_{l_1l_2} + \cdots + \frac{1}{d!}\sum_{l_1=1}^{n+m}\cdots\sum_{l_d=1}^{n+m} A^d_{jl_1\cdots l_d} z^{\otimes d}_{l_1\cdots l_d} \tag{52}$$

The tensor and coefficient derivations can be found in Supplementary Appendix Section S10.2] as the form of the equations are identical but the input state, weight matrix, and respective indices are directly analogous to the feedforward derivation.

5 Multiple layer network approximation

The output of the multi-layer network is an embedded Taylor approximation of each individual layer. Given a network with k layers, the final output $\boldsymbol{y}$ is a transformation of the kth intermediate state $\boldsymbol{z}^k$. The output expression is given in Eq. 53.

$$\begin{aligned}\boldsymbol{y} &= \boldsymbol{f_o}\left(\boldsymbol{z}^k\right)\\ &= a^{k,0} + A^{k,1}\boldsymbol{z}^k + A^{k,2}\boldsymbol{z}^{k,\otimes 2} + \cdots + A^{k,d}\boldsymbol{z}^{k,\otimes d}\end{aligned} \tag{53}$$

The last hidden layer prior to the output layer transforms intermediate state $\boldsymbol{z}^{k-1}$ to $\boldsymbol{z}^k$, given in Eq. 54, repeated for all intermediate states z for hidden layers 1 to $k-1$. The input layer transforms the data input $\boldsymbol{x}$ to the first intermediate state $\boldsymbol{z}^1$, given in Eq. 55.

$$\begin{aligned}\boldsymbol{z}^k &= \boldsymbol{f_k}\left(\boldsymbol{z}^{k-1}\right)\\ &= a^{k-1,0} + A^{k-1,1}\boldsymbol{z}^{k-1} + A^{k-1,2}\boldsymbol{z}^{k-1,\otimes 2} + \cdots + A^{k-1,d}\boldsymbol{z}^{k-1,\otimes d}\end{aligned} \tag{54}$$

$$\begin{aligned}\boldsymbol{z}^1 &= \boldsymbol{f_1}(\boldsymbol{x})\\ &= a^{i,0} + A^{i,1}\boldsymbol{x} + A^{i,2}\boldsymbol{x}^{\otimes 2} + \cdots + A^{i,d}\boldsymbol{x}^{\otimes d}\end{aligned} \tag{55}$$

The mapping from the input to the output is a recursive function embedding intermediate functions backward from the final layer, given in Eq. 56.

$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f_o}(\boldsymbol{f_k}(\cdots\boldsymbol{f_2}(\boldsymbol{f_1}(\boldsymbol{x}))\cdots)) \tag{56}$$

A verification of the multi-layer approximation may be found in Supplementary Appendix Section S10.4.

6 Physical constraints in the form of semi-algebraic constraints

The development of new semi-algebraic optimization routines motivate a polynomial representation of an unknown dynamical system extracted from NN. Since many physical vector fields are at least piecewise smooth, we can use Taylor's theorem to construct a polynomial that uniformly approximates the underlying ground truth arbitrarily tightly as the degree increases.

In the context of dynamical systems, physical laws provide valuable context that may be applied to state prediction. A more informed state prediction is found by combining the NN-derived polynomial state equations and user-defined semi-algebraic constraints. This is mainly enticing when learning and updating the new polynomial expression on-the-fly when new data is gathered. Hierarchies of semi-definite and sometimes even, linear program relaxations may be possible toward convergence of a global minimizer within the set of coefficients as parameters for such a problem. Moreover, incorporation of such constraints on the admissible set for its domain may indeed collapse the degree and required state dimension of the learned polynomial. Indeed, after the initial and possibly large polynomial is accurately extracted from the NN differential equation from the analytic unconstrained methods presented in this article, a new optimization goal would be to set up a hierarchy of optimization problems minimizing simultaneously, degrees and state dimensions, thus compressing dimension of the parameter space. The dimension of the parameter space, n_s, is given in Eq. 57, where n is the size of the state $\boldsymbol{s}$ and d is the degree of the polynomial.

$$n_s(d) = \begin{pmatrix} n+d \\ d \end{pmatrix} = \frac{(n+d)!}{d!n!} \tag{57}$$

This section offers common constraints but in application, any constraint in polynomial form may be defined and appended to this larger set of linear equations. A comprehensive state for rigid body dynamics is translation $[x\ y\ z]$ and rotation, $[\theta_x\ \theta_y\ \theta_z]$, in all six degrees of freedom and the associated velocity states, denoted by a derivative dot above the state. The full state vector is shown in Eq. 58.

$$\boldsymbol{s} = \begin{bmatrix} x & y & z & \dot{x} & \dot{y} & \dot{z} & \theta_x & \theta_y & \theta_z & \dot{\theta}_x & \dot{\theta}_y & \dot{\theta}_z \end{bmatrix} \tag{58}$$

For a predictive dynamics model, the neural network could be trained to develop a discrete transition model $\boldsymbol{f_d}$ to yield the next state $\boldsymbol{s}_{k+1}$ from the current state s_k or a continuous model $\boldsymbol{f_c}$ to yield the current state derivative $\dot{\boldsymbol{s}}_k$, given in Eqs 59, 60, respectively.

$$\boldsymbol{s}_{k+1} = \boldsymbol{f_d}(\boldsymbol{s}_k) \tag{59}$$

$$\dot{\boldsymbol{s}}_k = \boldsymbol{f_c}(\boldsymbol{s}_k) \tag{60}$$

Some suggested constraints of interest can be separated into systems under no external influence, external influence, and physical constraints. Under no external influence, a system's total energy (kinetic energy and potential energy U) is conserved, given in Eq. 61. Similarly, momentum is conserved in both translation and angular, given in Eq. 62. If rotational degrees of freedom are constrained, the θ terms in Eq. 62 are zero, and for constrained translation, the x, y, z terms are zero.

$$\begin{aligned}&\frac{1}{2}\left(\begin{bmatrix}\dot{x}_k & \dot{y}_k & \dot{z}_k\end{bmatrix} M \begin{bmatrix}\dot{x}_k \\ \dot{y}_k \\ \dot{z}_k\end{bmatrix} + \begin{bmatrix}\theta_{xk} & \theta_{yk} & \theta_{zk}\end{bmatrix} I \begin{bmatrix}\theta_{xK} \\ \theta_{yk} \\ \theta_{zK}\end{bmatrix}\right) + U(\boldsymbol{s}_k)\\ &= \frac{1}{2}\left(\begin{bmatrix}\dot{x}_{k+1} & \dot{y}_{k+1} & \dot{z}_{k+1}\end{bmatrix} M \begin{bmatrix}\dot{x}_{k+1} \\ \dot{y}_{k+1} \\ \dot{z}_{k+1}\end{bmatrix} + \begin{bmatrix}\theta_{x,k+1} & \theta_{y,k+1} & \theta_{z,k+1}\end{bmatrix} I \begin{bmatrix}\theta_{x,k+1} \\ \theta_{y,k+1} \\ \theta_{z,k+1}\end{bmatrix}\right) + U(\boldsymbol{s}_{k+1})\end{aligned} \tag{61}$$

$$M\begin{bmatrix}\dot{x}_k \\ \dot{y}_k \\ \dot{z}_k\end{bmatrix} + I\begin{bmatrix}\theta_{x,k} \\ \theta_{y,k} \\ \theta_{z,k}\end{bmatrix} = M\begin{bmatrix}\dot{x}_{k+1} \\ \dot{y}_{k+1} \\ \dot{z}_{k+1}\end{bmatrix} + I\begin{bmatrix}\theta_{x,k+1} \\ \theta_{y,k+1} \\ \theta_{z,k+1}\end{bmatrix} \tag{62}$$

Under external influence, these matrix equalities turn into matrix inequalities and additional external terms are integrated. Regarding conservation of energy, if this external influence has a known energy relationship, the external work term $W(\boldsymbol{s})$ may be seen in Eq. 63. Likewise, the momentum conservation equation is modified with external influences in the form of force and torque, seen in Eq. 64. If rotational degrees of freedom are constrained, the θ terms in Eq. 64 are zero, and for constrained translation, the x, y, z terms are zero.

$$\frac{1}{2}\left(\begin{bmatrix}\dot{x}_k & \dot{y}_k & \dot{z}_k\end{bmatrix} M \begin{bmatrix}\dot{x}_k\\ \dot{y}_k\\ \dot{z}_k\end{bmatrix} + \begin{bmatrix}\theta_{xk} & \theta_{yk} & \theta_{zk}\end{bmatrix} I \begin{bmatrix}\theta_{xk}\\ \theta_{yk}\\ \theta_{zk}\end{bmatrix}\right) + U(\boldsymbol{s}_k) + W(\boldsymbol{s}_k)$$
$$\geq \frac{1}{2}\left(\begin{bmatrix}\dot{x}_{k+1} & \dot{y}_{k+1} & \dot{z}_{k+1}\end{bmatrix} M \begin{bmatrix}\dot{x}_{k+1}\\ \dot{y}_{k+1}\\ \dot{z}_{k+1}\end{bmatrix} + \begin{bmatrix}\theta_{x,k+1} & \theta_{y,k+1} & \theta_{z,k+1}\end{bmatrix} I \begin{bmatrix}\theta_{x,k+1}\\ \theta_{y,k+1}\\ \theta_{z,k+1}\end{bmatrix}\right)$$
$$+U(\boldsymbol{s}_{k+1}) + W(\boldsymbol{s}_{k+1}) \tag{63}$$

$$M\begin{bmatrix}\dot{x}_k\\ \dot{y}_k\\ \dot{z}_k\end{bmatrix} + F(\boldsymbol{s}_k)\Delta t + I\begin{bmatrix}\theta_{x,k}\\ \theta_{y,k}\\ \theta_{z,k}\end{bmatrix} + \tau(\boldsymbol{s}_k)\Delta t$$
$$= M\begin{bmatrix}\dot{x}_{k+1}\\ \dot{y}_{k+1}\\ \dot{z}_{k+1}\end{bmatrix} + F(\boldsymbol{s}_{k+1})\Delta t + I\begin{bmatrix}\theta_{x,k+1}\\ \theta_{y,k+1}\\ \theta_{z,k+1}\end{bmatrix} + \tau(\boldsymbol{s}_{k+1})\Delta t \tag{64}$$

Outside of physical laws of dynamics, a more general constraint is a state bound in the form of an inequality. A state boundary could result from a physical boundary, in which a body cannot intersect another body. A lower bound and higher bound are given by Eq. 65, where $\boldsymbol{r}$ is a reference point.

$$\boldsymbol{s}_{k+1} \leq \boldsymbol{r} \quad \text{OR} \quad \boldsymbol{s}_{k+1} \geq \boldsymbol{r} \tag{65}$$

These two constraints can be superimposed as long as the constraints do not conflict with each other. Another set of bounding conditions results from a state relationship and sign of state derivative, given in state inequality (Eq. 66).

$$\begin{matrix}\boldsymbol{s}_{k+1} \leq \boldsymbol{s}_k & \text{if} & \dot{\boldsymbol{s}}_k \leq 0\\ & \text{OR} & \\ \boldsymbol{s}_{k+1} \geq \boldsymbol{s}_k & \text{if} & \dot{\boldsymbol{s}}_k \geq 0\end{matrix} \tag{66}$$

7 Solving for state prediction simultaneously with constraints

The proposed NN-Poly method can be used as a system identification method and/or a state prediction method. The state prediction solution in matrix form and in polynomial basis space is given in Eq. 67, where $\boldsymbol{y}_k$ is either the state derivative $\boldsymbol{s}_k$ or the propagated state $\boldsymbol{s}_{k+1}$.

$$\boldsymbol{y}_k = \tilde{J}_f^d \boldsymbol{a}^d \tilde{\boldsymbol{s}}^{\otimes d} \tag{67}$$

As derived previously, $\tilde{J}_f^d$ and $\boldsymbol{a}^d$ are both matrices populated only with numerical coefficients that approximate the NN function. Together, $\tilde{J}_f^d \boldsymbol{a}^d$ are an analogous state transition matrix, where the traditional input state $\boldsymbol{s}_k$ is augmented into the previously defined higher order polynomial basis space $\tilde{\boldsymbol{s}}^{\otimes d}$. As previously shown, constraints may be represented as linear equations in the same polynomial basis space. To solve for equality constraints simultaneously with state prediction, the constraint equation matrix, h_e, is appended to the polynomial state equation matrix, shown in Eq. 68, and solved with semi-definite programs (SDPs).

$$\begin{bmatrix}\boldsymbol{y}_k\\ h_e\end{bmatrix} = \begin{bmatrix}\tilde{J}_f^d \boldsymbol{a}^d\\ H_e\end{bmatrix} \tilde{\boldsymbol{s}}^{\otimes d} \tag{68}$$

For inequality constraints, numerical programs may be utilized to solve the semi-algebraic optimization problem, given in Eq. 69, where h_i are inequality constraint equations. Solving the semi-algebraic optimization problem is outside scope of this study, but the authors refer us to prevalent tools, such as the linear programming and semi-definite programming packages (Lasserre, 2015). These SDPs are convex but grow arbitrarily large as upon iteration. Future research includes exploiting the information incorporated in constraints known about the system to reduce the size of these SDPs, building upon work of Ahmadi and El Khadir (2020).

$$\boldsymbol{y}_k = \tilde{J}_f^d \boldsymbol{a}^d \tilde{\boldsymbol{s}}^{\otimes d} \qquad \text{such that} \qquad h_i \leq H_i \tilde{\boldsymbol{s}}^{\otimes d} \tag{69}$$

One can implement constrained state prediction in real-time on an embedded dynamic system in a learning application. In an algorithmic loop, a neural network may be trained with every iteratively collected measurement, then the subsequent polynomial may be generated and the state prediction/control effort constrained with predefined constraints. Other applications include post-processing system identification for data-constrained applications, in which neural network parameters are more effective to communicate than large data sets, like space or deep-sea applications.

8 NN-Poly state prediction performance

Our aim in these results is to show the minimal approximation loss between the NN-Poly and the models that were derived directly from data: the neural network and polynomial. In the unconstrained case, the NN-Poly can only be as accurate as the neural network from which the polynomial is derived from as the polynomial approximates the neural network. Similarly, the NN-Poly can only be as accurate as a polynomial found from the original data because NN-Poly lost information between the original data and the neural network representation. In the constrained case, the NN-Poly has the ability to incorporate domain knowledge that the neural network cannot capture, which offers NN-Poly the ability to better represent the underlying dynamic system generating the data; for example, a neural network trained on data generated from a bouncing ball may predict the ball's state to intersect with the ground, but constraints placed on the polynomial model would bound predictions such that the ball never intersects with the ground, yielding a more accurate state prediction. The proposed Taylor expansion with the NN parameters is coded in MATLAB (https://github.com/alexdhjing/

TABLE 2 Metrics for evaluating system identification methods on dynamic systems.

Metrics	**Measurement**	**Definition/equation**
Computing time	Efficiency	Time cost to run calculation
State error (mean squared error)	State accuracy	$MSE = \frac{1}{T}\sum_{t=1}^{T} (\boldsymbol{x}_k - \hat{\boldsymbol{x}}_k)^2$
Coefficient error	Expression accuracy	MSE of coefficients (if in polynomial form) $CE = \frac{1}{m}\sum_{i=1}^{m} (c_i - \hat{c}_i)^2$Marked as N/A if not in polynomial form
Parameter stability	Expression accuracy and complexity	Largest parameter divided by smallest coefficient $PS = \frac{\max(\boldsymbol{p})}{\min(\boldsymbol{p})}$
Length of solution	Expression accuracy and complexity	Number of nonlinear terms in solution

NNX_matlab). Metrics to evaluate performance of each method are efficiency, accuracy, and complexity. This section discusses the method of comparison, metrics to evaluate performance, and subsequent results of the state prediction methods.

8.1 Methods

Several dynamic models are used to measure efficiency and accuracy of each state prediction method, not only in the state prediction but also in the model representation. Each algorithm is trained with the same pairwise input and output data, generated from the true dynamic model, to predict a state for a range of dynamic systems. The input–output data pairs are of form $\{\boldsymbol{x}_k, \boldsymbol{x}_{k+1}\}$, in which the input is state vector at time step k and the output is of time $k+1$. A majority of data pairs train the various models and the rest of the data pairs are set aside to evaluate state error and computation time. Other metrics evaluate the model coefficients and structure after training. All metrics are formally defined in the next subsection.

The selected dynamic models were chosen in ascending complexity, from a two-dimensional feature space with linear dynamics and non-linear dynamics. The 1DOF underdamped linear spring mass damper system has a linear transition matrix from previous state to next state, constituting the simplest dynamic system to identify, described in Eq. 70. The 1DOF non-linear spring system has non-linear spring stiffness and is governed by a differential equation, given in Eq. 71. A 2DOF spring pendulum demonstrates non-linearity and coupling of dynamics, given in Eq. 72. In both the 1DOF non-linear spring and 2DOF spring pendulum cases, the differential equations of motion propagate the state to generate data. The system identification methods do not generate expressions for acceleration but generate a mapping from previous state to next state, implicitly integrating the double derivative.

$$\begin{bmatrix} x_{t+1} \\ v_{t+1} \end{bmatrix} = \begin{bmatrix} 0.9995 & 0.01 \\ -0.0999 & 0.9985 \end{bmatrix} \begin{bmatrix} x_t \\ v_t \end{bmatrix} \tag{70}$$

$$\ddot{x} = \frac{-3\mu_0}{2\pi(x+a)^4} \tag{71}$$

$$\begin{bmatrix} \ddot{x} \\ \ddot{y} \end{bmatrix} = \begin{bmatrix} -5\left(\sqrt{x^2+y^2} - 1\right)\dfrac{x}{\sqrt{x^2+y^2}} \\ -5\left(\sqrt{x^2+y^2} - 1\right)\dfrac{y}{\sqrt{x^2+y^2}} - 10 \end{bmatrix} \tag{72}$$

8.2 Metrics of evaluation

Metrics for evaluation are computation time, state error (mean squared error), coefficient error, length of solution, and coefficient stability, given in Table 2. The state error is of mean squared error form of the algorithm's state prediction and the true state for which smaller error represents a more accurate approximation. Coefficient error refers to a coefficient vector $\boldsymbol{c}$ of length m, geared toward polynomial solutions for which smaller error again signifies a more accurate model. Parameter stability refers to the learned parameters, relevant for all methods, for which values closer to 1 represent high stability and values approaching infinity represent instability. The length of solution refers to the number of $h(\boldsymbol{x})$ for which a smaller number of terms represents a more parsimonious solution.

8.3 Complexity

We will explore the complexity of our proposed method by deriving and comparing the number of flops for the following methods: a polynomial directly from data, training a NN and NN-Poly. Intuitively before even approaching a rigorous derivation, the NN-Poly method explicitly relates NN parameters to polynomial coefficients, emulating a "hard-coded" computation. We added computing time into the evaluation metrics as calling table elements from memory would likely be the most computationally intensive task in the NN-Poly method. Calculating polynomial coefficients directly from a large data set with the least-squares method will be computationally intensive due to the matrix inverse operation that scales cubed with dimension of the data set. A neural network's flops depend on training epochs until convergence, stochastically related to the initial parameterization of the neural network. For all flop calculations, we will assume that:

- The input state size m and output state size n are equivalent $m = n$
- The state size m is much smaller than the amount of training points t, $m \ll t$

The least-squares method to calculate polynomial coefficients A with a matrix of inputs X and outputs Y is given in Eq. 73, where A_p is the set of polynomial coefficients derived from the least-squares method.

$$A_p = Y\left(X^{\otimes d}\right)^T\left(\left(X^{\otimes d}\right)\left(X^{\otimes d}\right)^T\right)^{-1} \tag{73}$$

The number of flops of a polynomial derived directly from data is on the order given in Eq. 74. Intermediate steps are given in Eqs 135–138 in Supplementary Appendix Section S10.5.

$$\mathcal{O}\left(A_p\right) \approx t\frac{m^d}{d!} + \frac{m^{3d}}{d!} \tag{74}$$

To train neural networks, we must account for forward passes and backward propagation per epoch overall training data. In implementation, the neural network does not necessarily train across all the data but for this derivation of flops will assume so to simplify the calculation. We will also assume that the network has only a single layer with k neurons. The total number of flops to train a neural network is given in Eq. 75.

$$\mathcal{O}(\text{NN}) \approx mnkt \tag{75}$$

To convert this neural network into a polynomial, the polynomial coefficients for each polynomial degree scales with the number of neurons in the layer. The highest order polynomial dominates the number of flops, which approximates the total number of flops in converting a single-layer NN with k neurons to a polynomial of degree d given in Eq. 76. The set of polynomial coefficients derived from the NN is given by A_{NN}^d. Intermediate steps are shown in Eqs 140–145 in Supplementary Appendix Section S10.5.

$$\mathcal{O}\left(A_{NN}^d\right) \approx km^{d+1} \tag{76}$$

The combined complexity of training a neural network and converting those NN parameters to polynomial coefficients is given in

$$\mathcal{O}\left(\text{NN} + A_{NN}^d\right) \approx km^2t + km^{d+1} \tag{77}$$

$$= k\left(m^2t + m^{d+1}\right) \tag{78}$$

The condition for the neural network training to be more computationally complex than the NN-Poly conversion is given in Eq. 79.

$$\begin{cases} \mathcal{O}(\text{NN}) > \mathcal{O}\left(A_{NN}^d\right), & \text{if } t > m^{d-1} \\ \mathcal{O}\left(A_{NN}^d\right) > \mathcal{O}(\text{NN}), & \text{otherwise} \end{cases} \tag{79}$$

Given the order of complexity of each method that ultimately yields a polynomial (least-squares polynomial directly from data and polynomial from NN parameters), what are the conditions on state size m, training data size t, and neural network size by neurons k that make either method more or less complex than the other? The condition for the NN-Poly method to be less complex than a polynomial from the raw data is given in Eq. 80.

$$\begin{cases} \mathcal{O}\left(A_p\right) > \mathcal{O}\left(\text{NN} + A_{NN}^d\right), & \text{if } mk < \dfrac{t + m^d}{d!} \\ \mathcal{O}\left(\text{NN} + A_{NN}^d\right) > \mathcal{O}\left(A_p\right), & \text{otherwise} \end{cases} \tag{80}$$

Intuitively, the NN-Poly method is computationally advantageous for the following conditions:

TABLE 3 1DOF spring evaluation of performance for all system identification methods.

Metrics	Polynomial	NN	NN-Poly
Computing time	**0.29 s**	3.63 s	0.37 s
State error (MSE)	$\epsilon(x)$	**1.1** $\times 10^{-8}(x)$	$1.1 \times 10^{-5}(x)$
	$\epsilon(x)$	**1.5** $\times 10^{-8}(v)$	$1.5 \times 10^{-5}(v)$
Coefficient error (CE)	0.006 (x)	N/A	**0 (x)**
	0.006 (v)		**0 (v)**
Parameter stability	99.95 (x)	$99.95(x)$	$99.95(x)$
	9.95 (v)	$9.995(v)$	$9.995(v)$
Number of parameters	6	6	6

Bolded values are the best values across methods for each metric.

TABLE 4 1 DOF flux-pinned system evaluation of performance for all system identification methods.

Metrics	Polynomial	NN	NN-Poly
Computing time	**0.076 s**	1.36 s	0.20 s
State error (MSE)	$1.9, \times, 10^{-3}(x)$	**1.3** $\times 10^{-4}(x)$	**1.3** $\times 10^{-4}(x)$
	$9.0, \times, 10^{-4}(v)$	**1.1** $\times 10^{-4}(v)$	**1.1** $\times 10^{-4}(v)$
Parameter stability	$5.8 \times 10^4(x)$	**67.8(x)**	94.8 (x)
	$2.8 \times 10^3(v)$	**574(v)**	579.4 (v)
Number of parameters	30	**6**	**6**

Bolded values are the best values across methods for each metric.

TABLE 5 2 DOF spring pendulum system evaluation of performance for all system identification methods.

Metrics	Polynomial	NN	NN-Poly
Computing time	**0.29 s**	3.58 s	0.36 s
State error (MSE)	**5.0** $\times 10^{-4}(x)$	$3.0 \times 10^{-3}(x)$	$3.2 \times 10^{-3}(x)$
	3.5 $\times 10^{-3}(v)$	$1.62 \times 10^{-2}(v)$	$1.7 \times 10^{-2}(v)$
Parameter stability	**6.6** $\times 10^3(x)$	$7.6 \times 10^3(x)$	$8.1 \times 10^3(x)$
	8.6 $\times 10^3(v)$	$1.3 \times 10^3(v)$	$1.4 \times 10^3(v)$
Number of parameters	120	**20**	140

Bolded values are the best values across methods for each metric.

- There are many training points
- The neural network is small
- The state size is large
- The state size is sufficiently large and the polynomial degree is large

Complexity does not fully capture computational load though, as the NN-Poly method relies heavily on memory calls. The next section incorporates computing time for this reason.

8.4 Results

For each dynamic system, Tables 3–5 report the performance of the strict polynomial, the sole NN and NN-Poly for each

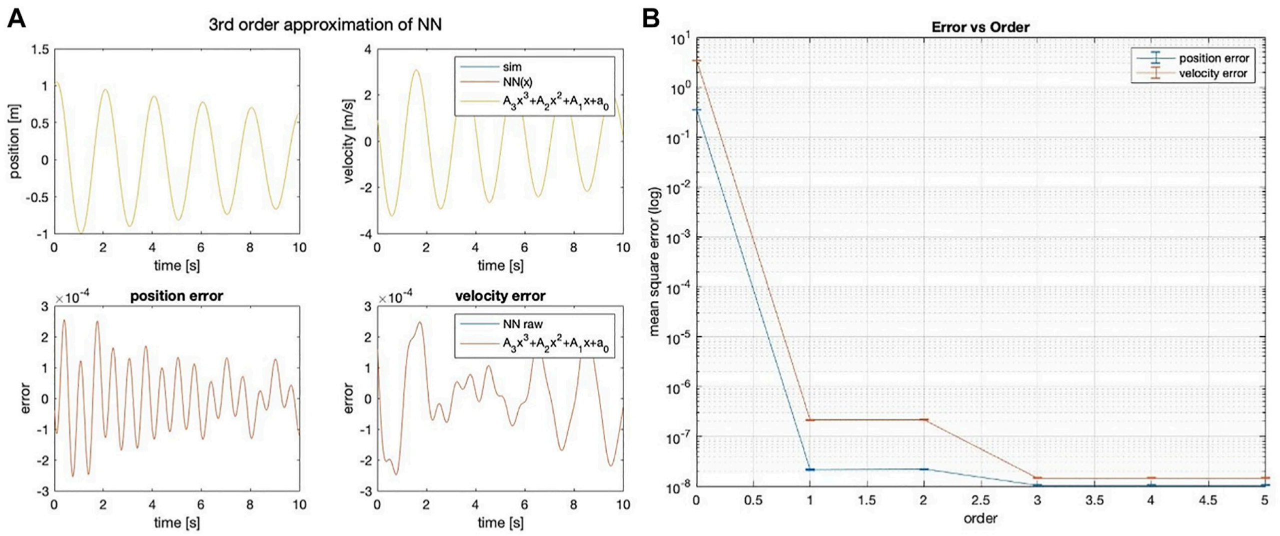

FIGURE 3
(A) Third-order NN-Poly approximation of 1DOF linear system and error as difference of each state. **(B)** MSE error of position and velocity state across different orders of approximation for NN-Poly.

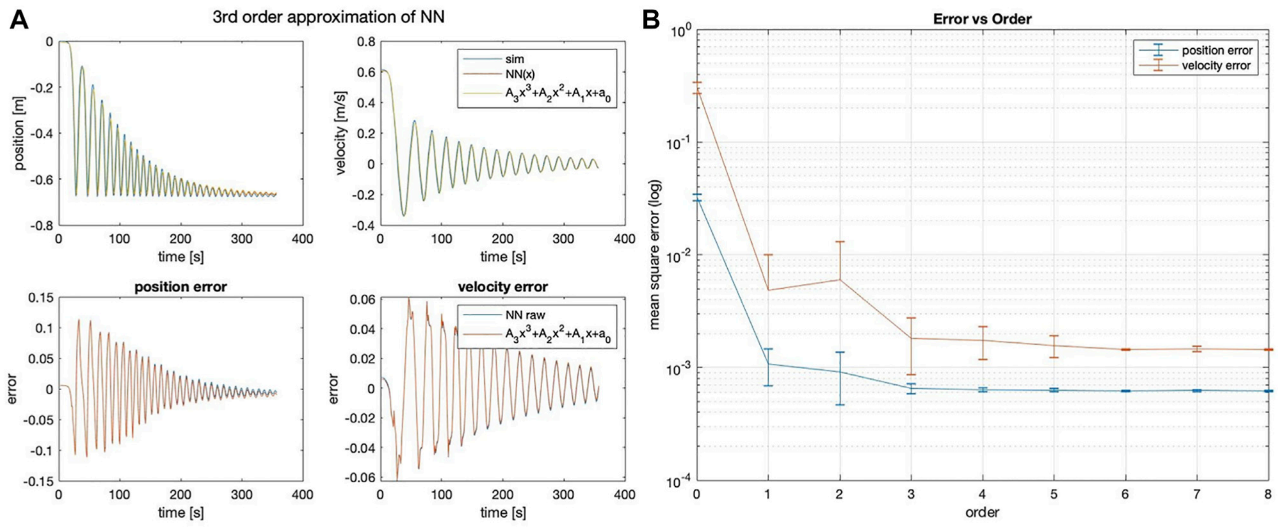

FIGURE 4
(A) Third-order NN-Poly approximation of 1DOF non-linear spring system and error as difference of each state. **(B)** MSE error of position and velocity state across different orders of approximation for NN-Poly.

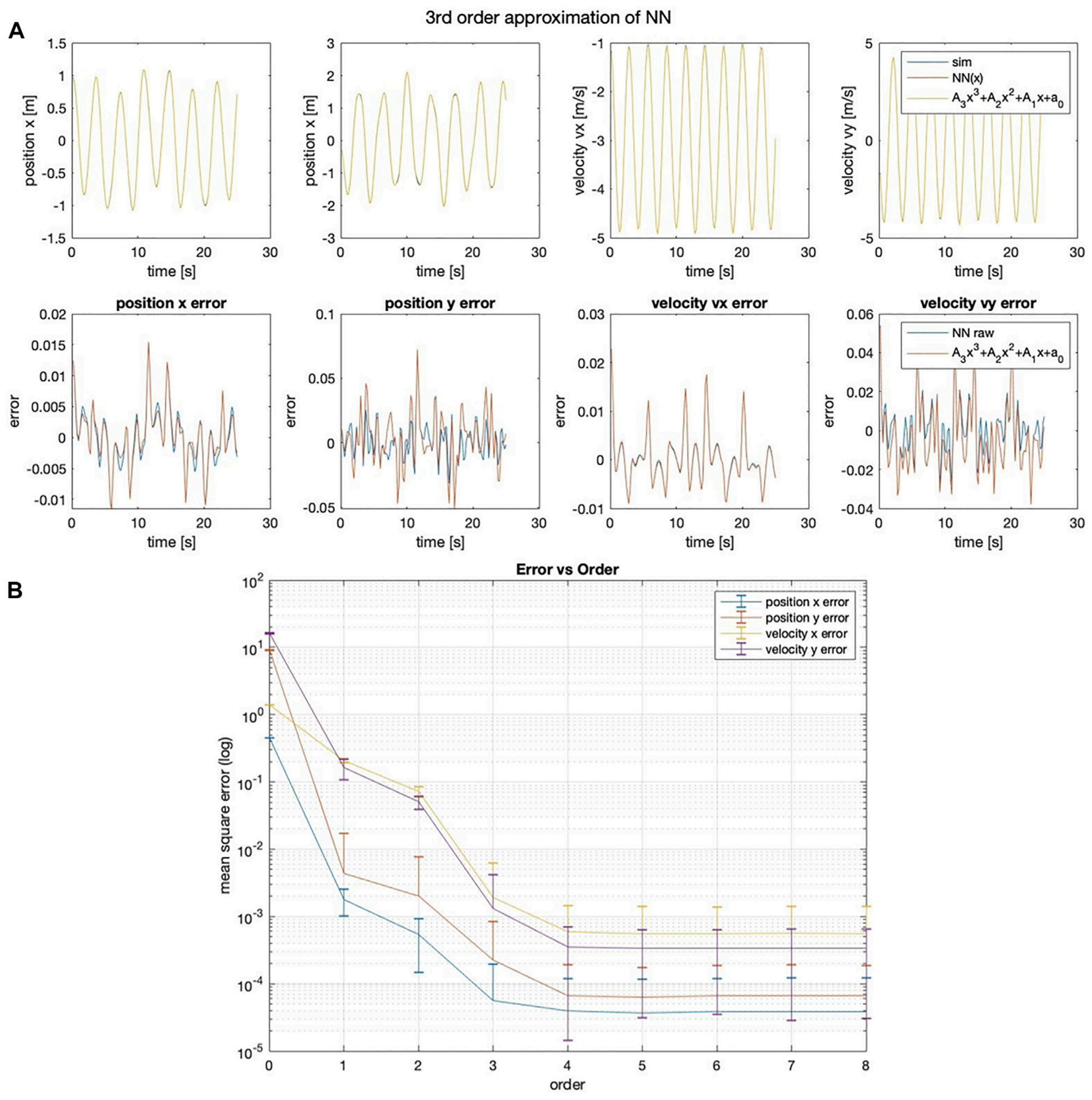

FIGURE 5
(A) Third-order NN-Poly approximation of a 2DOF non-linear coupled spring–pendulum system and error as difference of each state. **(B)** 2DOF spring–pendulum individual MSE error of position and velocity state across different orders of NN-Poly approximation.

dynamic system. Wherever an ϵ is listed in the tables, ϵ denotes an immensely small value within machine precision. The computation time for the NN-polynomial expansion does not include the NN training time, only the time to transform the NN parameters into polynomial coefficients.

The method that best predicts the 1 DOF linear spring state is the strict polynomial; although curiously, the NN-Poly exactly predicts the original system state matrix coefficients. The performance across all methods is shown in Table 3. The proposed NN-Poly approximation method performance in each state's time series and across increasing polynomial degrees is shown in Figure 3. As expected, increasing the order of polynomial approximation yields less state error, although not much performance is gained past the second-order approximation, seen in Figure 3. The small error bars for each degree signals that the final solution from each NN random initialization does not vary error much. The NN-polynomial conversion is comparable in computation but includes more state error as expected. This straightforward test case is best solved with the simplest solver, the direct polynomial solution; even a least squares solution is sufficient. This system does not need a complex, expressive system identification method due to the simplicity of its linear form.

The method that best predicts the 1 DOF non-linear spring state are NN and NN-Poly, with performance reported in Table 4. Unlike the linear system approximation, the proposed NN-Poly approximation varies from the NN parameter initialization and does not converge to a steady state error until the fourth order, seen in Figure 4. The sole NN and NN-Poly approximate the data with the least state error, generate a minimal representation, and yield the most stable parameters of all methods. Subsequently, the NN-Poly retains the same value of state error from NN, demonstrating the accuracy of transformation from a NN form to polynomial form, while also offering a form from which to apply domain knowledge and safety guarantees.

For the 2DOF system, NN and NN-Poly predict state accurately, seen in Table 5. The third order NN-poly approximation for each state is shown in Figure 5A. The NN-Poly MSE error decreases with increasing polynomial degree and does not converge until after the fourth order, seen in Figure 5B. The NN and NN-Poly predict the 2DOF system with fewer terms. Each method has comparable parameter stability and computation time. This coupled and slightly non-linear dynamic system straddles the boundary in deciding which model to use.

9 Conclusion

Leveraging recent advances in deep learning, NN-Poly provides accurate predictions of non-linear, coupled system dynamics with minimal context. A NN-to-polynomial mapping avoids the need to download an immense amount of data and fit a polynomial directly to a large dataset by exploiting the compactness of the NN. A polynomial enhances interpretability of the neural network by making its feature dependence explicit, as polynomials have a long history of analysis and mathematical literature, including safety verification and guarantees. This work's major contribution is offering a polynomial form and semi-algebraic constraints, such polynomial inequality and equality constraints, to capture the NN model and system context in the final predictive function solution. These semi-algebraic constraints represent the application of domain knowledge, such as conservation of energy in the form of quadratic rates, and safety constraints, such as linear inequalities that bound specific state values. The results of this effort show comparable prediction and computation performance between a sole NN, sole polynomial, and the proposed method for linear systems but great improvement in the proposed method for highly nonlinear systems. Further future work also includes approximation of more complex systems, in increasing degrees of freedom, in the degree of non-linearity, and in the coupling of states.

Data availability statement

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found as follows: https://github.com/alexdhjing/NNX_matlab.

Author contributions

FZ was the primary author for the majority of this manuscript and derived all the theory that produced all equations. DJ wrote the code that generated the plots in the results. FL contributed the insight and section on semi-algebraic constraints. SF reviewed the paper and assisted in identifying gaps in explanation. FL and SF guided the trajectory of the research contributions.

Funding

FZ thanks the NASA Space Technology Research Fellowship Grant NNX15AP55H and the NSF AI Institute in Dynamics Systems award number 2112085 for supporting this research. SF's work was funded by the Office of Naval Research (ONR) grant N00014-19-1-2266.

Acknowledgments

FZ would like to thank Chelsea Sidrane and Anish Potnis for their valuable feedback.

Conflict of interest

Author DJ was employed by the company Huawei.

The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors, and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.

Supplementary material

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/frobt.2022.968305/full#supplementary-material

References

Ahmadi, A. A., and El Khadir, B. (2020). "Learning dynamical systems with side information," in Learning for Dynamics and Control (PMLR), 718.

Ahmadi, A. A., and Parrilo, P. A. (2011). "Converse results on existence of sum of squares lyapunov functions," in 2011 50th IEEE conference on decision and control and European control conference (IEEE), 6516.

Cranmer, M., Greydanus, S., Hoyer, S., Battaglia, P., Spergel, D., and Ho, S. (2020). Lagrangian neural networks. *arXiv preprint arXiv:2003.04630*.

Djeumou, F., Neary, C., Goubault, E., Putot, S., and Topcu, U. (2022). "Neural networks with physics-informed architectures and constraints for dynamical systems modeling," in Learning for Dynamics and Control Conference (PMLR), 263–277.

Dutta, S., Chen, X., and Sankaranarayanan, S. (2019). "Reachability analysis for neural feedback systems using regressive polynomial rule inference," in Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control. (IEEE).

Ferrari, S., Rudd, K., and Muro, G. (2013). "A constrained backpropagation (cprop) approach to function approximation and approximate dynamic rogramming," in Reinforcement Learning and Approximate Dynamic Programming for Feedback Contro, 162–181.*l*.

Ferrari, S., and Stengel, R. F. (2005). Smooth function approximation using neural networks. *IEEE Trans. Neural Netw.* 16, 24–38. doi:10.1109/tnn.2004.836233

Granados, A. (2015). *Taylor series for multi-variable functions*.

Greydanus, S., Dzamba, M., and Yosinski, J. (2019). "Hamiltonian neural networks," in *Advances in neural information processing systems*. Editors H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alche-Buc, E. Fox, and R. Garnett (Curran Associates, Inc.), 32.

Hildebrand, A. (2009). *Multinomial coefficients*.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Netw.* 2, 359–366. doi:10.1016/0893-6080(89)90020-8

Huang, C., Fan, J., Li, W., Chen, X., and Zhu, Q. (2019). Reachnn: Reachability analysis of neural-network controlled systems. *ACM Trans. Embed. Comput. Syst.* 18, 1–22. doi:10.1145/3358228

Lagaris, I. E., Likas, A., and Fotiadis, D. I. (1998). Artificial neural networks for solving ordinary and partial differential equations. *IEEE Trans. Neural Netw.* 9, 987–1000. doi:10.1109/72.712178

Lasserre, J. B. (2015). *An introduction to polynomial and semi-algebraic optimization*, 52. Cambridge: Cambridge University Press.

Lipton, Z. C. (2018). The mythos of model interpretability. *Queue* 16, 31–57. doi:10.1145/3236386.3241340

Liu, W., Lai, Z., Bacsa, K., and Chatzi, E. (2022). Physics-guided deep markov models for learning nonlinear dynamical systems with uncertainty. *Mech. Syst. Signal Process.* 178, 109276. doi:10.1016/j.ymssp.2022.109276

Narasimhamurthy, M., Kushner, T., Dutta, S., and Sankaranarayanan, S. (2019). "Verifying conformance of neural network models," in 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (IEEE), 1.

Pinkus, A. (1999). Approximation theory of the mlp model in neural networks. *Acta Numer.* 8, 143–195. doi:10.1017/s0962492900002919

Psichogios, D. C., and Ungar, L. H. (1992). A hybrid neural network-first principles approach to process modeling. *AIChE J.* 38, 1499–1511. doi:10.1002/aic.690381003

Raissi, M., Perdikaris, P., and Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.* 378, 686–707. doi:10.1016/j.jcp.2018.10.045

Rolnick, D., and Tegmark, M. (2017). The power of deeper networks for expressing natural functions. *arXiv preprint arXiv:1705.05502*.

Rudy, S. H., Brunton, S. L., Proctor, J. L., and Kutz, J. N. (2017). Data-driven discovery of partial differential equations. *Sci. Adv.* 3, e1602614. doi:10.1126/sciadv.1602614

Sidrane, C., Katz, S., Corso, A., and Kochenderfer, M. J. (2022). Verifying inverse model neural networks. *arXiv preprint arXiv:2202.02429*.

Wang, S., Sankaran, S., and Perdikaris, P. (2022). Respecting causality is all you need for training physics-informed neural networks. *arXiv preprint arXiv:2203.07404*.