Exploring the Potential of Neural Networks for Nonlinear System Identification

Abstract

System identification is an essential tool in many fields of engineering and science, including control engineering, signal processing, robotics, and neuroscience. The ability to build accurate mathematical models of dynamic systems from measured data is critical for understanding and predicting the behavior of these system. System identification can also provide insight into the underlying mechanisms that govern the system's behavior and can help identify potential sources of instability or error. In this project neural-network architecture with integrated symbolic-regression, known in literature as an equation learner (EQL), is utilized to extract the latent variables of various dynamical systems from synthesized time series data. The proposed method can detect and identify unknown parameters efficiently, and it has shown to reproduce the underlying dynamic equations in a human-readable form. The proposed method is also tested with increasingly complex systems, and its ability to extrapolate equations from these systems is evaluated.

Problem Overview

System identification is an essential tool in many fields of engineering and science, including control engineering, signal processing, robotics, and neuroscience. The ability to build accurate mathematical models of dynamic systems from measured data is critical for understanding and predicting the behavior of these systems, as well as for designing controllers or other interventions to achieve desired outcomes. System identification can also provide insight into the underlying mechanisms that govern the system's behavior and can help identify potential sources of instability or error.

However, system identification is not without its challenges. In many cases, the underlying system is complex and highly nonlinear, making it difficult to capture its behavior with simple models. Like the principle of superposition used in Fourier analysis, these systems can be represented as a summation of infinitely many linear functions. The greater the complexity of such a function may increase the predictive performance of the model, but at the cost of increased computational complexity and reduced interpretability.

In most cases, the underlying system that generates the data has a sparse structure, meaning that only a small subset of the available features or variables are relevant to the system's behavior. By leveraging this sparsity, one can develop models that are more interpretable, require less data for training, and can generalize better to unseen data. This has driven the development of many techniques using methods that promote sparsity in their regularization and has proven useful with linear regression models.

Unfortunately, we see complexity rise again with this approach when the underlying system is multivariate or contains only a few highly non-linear terms. E.g., for a system with two states x_1 and x_2 and one control input u, if a third order term is known to be required to fully capture the dynamics, 64 different possible coefficients would need to be evaluated. These broad assumptions can make regression difficult.

$$f(x_1, x_2, u) = (\beta_n\, x_1 + \beta_1)^3 (\beta_n x_2 + \beta_1)^3 (\beta_n u + \beta_1)^3$$

Several artificial neural network methods have been developed that reduce the library of terms required. In this application, we do not desire for the artificial neural network (ANN) to act as a black-box interpreter, but rather to produce an interpretable model from the given data. “Symbolic regression is a type of regression analysis that searches the space of mathematical expressions to find the best model that fits the data and can thus fit a much wider range of data sets than other models such as linear regression” [4]. While an argument for relative computational expense can be made, it is my desire to explore this domain of system identification in this project.

Background and Literature

One of the recent popular forms of linear regression – Sparse Identification of Nonlinear Dynamics (SINDy) [8] has been used in conjunction with autoencoders (AE) to learn dynamic models from artificially high-dimension input [1]. This is essentially standard linear regression with a simple sparsity-promoting L_1 regularization. In this paper, a pendulum's motion was simulated as a video, or system states were transformed using Legendre polynomials. Data was passed through a convolutional AE, and SINDy was used on the latent space variables to determine the system dynamics.

Neural networks have the potential to model complex systems due to their ability to learn patterns and relationships in large amounts of data. They can capture non-linearities and interactions in the data, making them useful for a wide range of applications, including predicting outcomes, identifying patterns, and generating symbolic expressions.

The idea of using ANNs to generate symbolic expressions is not necessarily new. In fact, this has been a rich area of research since at least 1989 [3]. One of the more relevant and seminal works in the field uses data-driven symbolic regression to successfully recover Hamiltonians and Lagrangians for dynamic systems [7]. However, this early approach did not scale with the dimensionality of the system.

Other applications with developments in the field have pursued graph networks [2] to examine particle movements and apply the approach to solving the movements of distant celestial bodies. A similar feat was accomplished with sparse symbolic regression in combination with genetic programming [6].

Similarly, David Zheng, et al, proposed a model that extracts symbolic representations of inelastic collisions through using a perceptron network [10]. This is the foundational paper on which the Equation Learner (EQL), as discussed later in this paper, is built.

Data Source

The data for this project was initially planned to come in the form of progressively more complicated dynamical systems. Unfortunately, due to the complexity of the selected network design, only one constant value is able to be passed into the EQL (described below). I mistakenly thought it would be simple to change the dimension of the latent variable Z, but it was hard coded into the network. As a result, I was unable to test several of the planned models. Table 1 below lists the planned and executed dynamic models.

	Model		Achieved	x_1	x_2	*const.*
1	Simple Kinematics	$\ddot{x} = F/m$	Yes	$\mathcal{U}(-1,1)$	$\mathcal{U}(-1,1)$	$a = \mathcal{U}(-1,1)$
2	Simple Harmonic Oscillator	$\ddot{x} = -(k/m)x$	Yes	$\mathcal{U}(-1,1)$	$\mathcal{U}(-0.5,0.5)$	$w^2 = \mathcal{U}(0.1,1)$
3	Nonlinear w/ Bifurcation	$\ddot{x} = \mu x + x^3 - x^5$	Yes	$\mathcal{U}(-1.5,1.5)$	$\mathcal{U}(-1,1)$	$\mu = \mathcal{U}(0.1,1)$
4	Logistic Growth	$\dot{x} = rx(1 - x/K)$	No	-	-	-
5	Lotka-Voltera Model	$\dot{x} = (b - py)x = bx - pyx$ $\dot{y} = (rx - d)y = rxy - dy$	No	-	-	-
6	Van der Pol Oscillator (1-d)	$\ddot{x} = \mu(1 - x^2)\dot{x} - x$	Yes	$\mathcal{U}(-1,1)$	$\mathcal{U}(-1,1)$	$\mu = \mathcal{U}(0.1,4)$
7	Van der Pol Oscillator (2-d)	$\dot{x} = y$ $\dot{y} = \mu(x - x^2)y - x$	No	-	-	-
8	Lorenz Attractor	$\dot{x} = \sigma(y - x) = \sigma y - \sigma x$ $\dot{y} = x(\rho - z) - y = x\rho - xz - y$ $\dot{z} = xy - \beta z$	No	-	-	-

Table 1 – Planned and executed dynamic models with the sample ranges for initial conditions x_1 and x_2 and constants.

Parameter and Initial Condition Selection

Data sets were generated programmatically from a set range of initial conditions and parameter values. A maximum and minimum value for each was selected, and uniformly distributed samples were selected from this range $\mathcal{U}(\min, \max)$ to initialize the trajectories. This method allows the input data to span bifurcated regions, or more narrowly selected regions if necessary. In all, 1000 time-series trajectories were created over 50 seconds, with 500 time-steps, for $\Delta t = 0.1$ seconds. Kim, et al (2020) used a similar method with the simple kinematic and harmonic oscillator [4]. The final output dimension for each dataset is: $x = (2 \times 1000 \times 500)$, $const\ (e.g.: \mu) = (1000,)$.

Proposed Approach

The key paper used to develop this project uses a variation on an autoencoder. The encoder uses non-variational convolutions, and the decoder is a series of multi-layered perceptron subnetworks. This approach is referred to as DE-PD, or Dynamics Encoder - Propagating Decoder. Both encoder and decoder are trained simultaneously using the mean-squared error loss between the predicted $\hat{\boldsymbol{y}}$ and calculated dynamics $\boldsymbol{y}$. Thankfully, the author published the network on github [11]

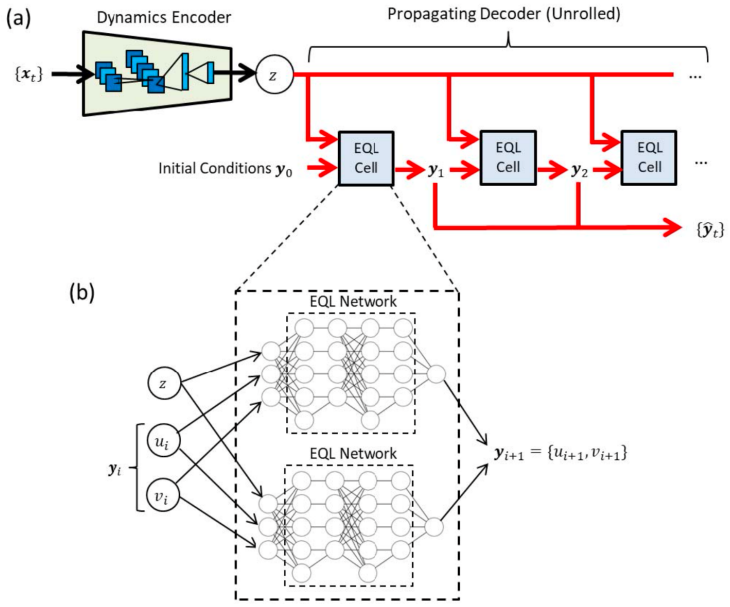

Fig. 1. (a) Architecture to learn the equations that propagate a dynamical system. (b) Each EQL cell in the propagating decoder consists of separate EQL networks for each dimension of **y** to be predicted. Here:
y = {*u*,*v*}, where *u* is the position and *v* is velocity, so there are 2 EQL networks per cell [3].

Dynamics Encoder (DE)

The DE takes in the full time series data and outputs a single-dimensional latent variable $\boldsymbol{z}$ using several layers of convolutions and a final batch normalization layer.

Propagating Decoder (PD)

The parameter $\boldsymbol{z}$ and the initial conditions $\boldsymbol{y_0}$ are propagated down through a recurrent neural network (RNN) consisting of a series of EQL cells. Further detail is provided in the section on multi-phase training.

Equation Learner (EQL)

The core functionality of the symbolic regressor is the EQL. Here, the latent variable and initial conditions are passed through a fully connected NN where the typical activation functions $f(\boldsymbol{g})$ are replaced with a library of proposed functions. Each $f_i(g_i)$ may be a sin, exponential, sigmoid or product of $g_{n-1} * g_n$ such as

$$f(\boldsymbol{g}) = \left[f_1(g_1), f_2(g_2), \ldots, f_{n_h}\left(g_{n_g-1}, g_{n_g}\right)\right]^T .$$

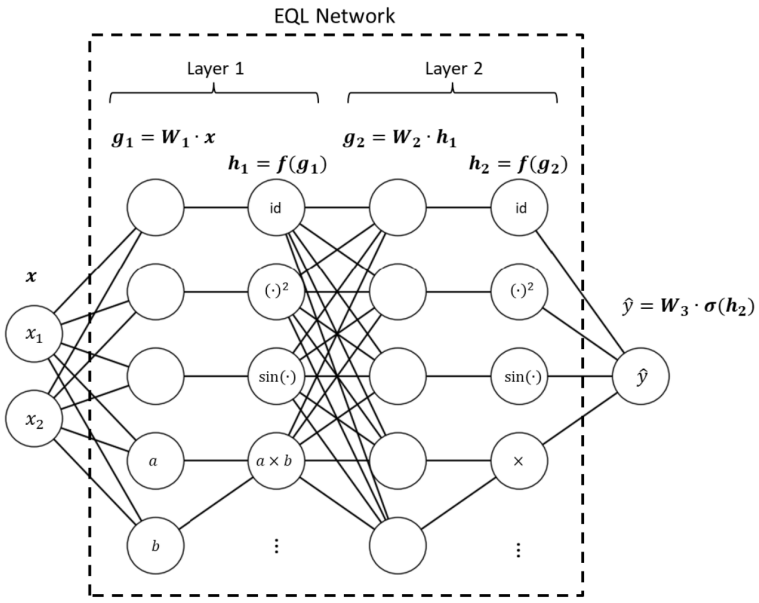

Fig. 2. Example of the EQL subnetwork for symbolic regression. Only 4 activation functions (id, square, sin, and multiplication) and two hidden layers are shown for simplicity [3].

Multi-Step Training

Training is performed Tensorflow utilizing RMSProp – a backpropagation optimizer with mean-squared error (MSE), and the loss function:

$$\mathcal{L} = \frac{1}{N}\sum(y_i - \hat{y_i})^2 \; + \; \lambda \, L_{0.5}^*$$

Where N is the size of the training data set and λ is a hyperparameter that balances the regularization versus the MSE. The $L_{0.5}^*$ regularization uses a piecewise function to smooth out normal $L_{0.5}$ at small values (Fig. 3).

$$L_{0.5}^*(w) = \begin{cases} |w|^{1/2} & |w| \geq a \\ \left(-\dfrac{w^4}{8a^3} + \dfrac{3w^2}{4a} + \dfrac{3a}{8}\right)^{1/2} & |w| < a \end{cases}$$

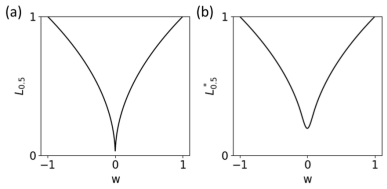

Fig. 3. $L_{0.5}$ (left) vs $L_{0.5}^*$ (right) regularization for small values of w.

The training is broken up into two or more phases. In the first phase, the network is trained with a small learning rate value of λ to allow for the network to evolve freely. In the second phase, λ is increased to make the EQL network sparse, and weights below a threshold α are set to 0 and frozen. In the final phase, the system is fine-tuned without $L_{1/2}$ regularization and with an increased learning rate.

In the foundational paper, in the simple kinematic motion, the model is only trained on the first time-step (T=1) for phases one and two, and T=5 for the remainder of training after the small wights are frozen. Similarly, in the SHO model, T = 1 for the first 500 epochs, and then 2 additional time steps are added each 500 epochs of training through phase 1. In phase 2, the total number of steps used for training is increased to T = 25 for the rest of the training. See table 1 for learning and regularization weights α and λ used for both models in all 3 phases. The 3-phase setup was repeated for both the nonlinear bifurcation and 1-D Van-der-Pol models.

	Kinematics Model			SHO Model		
	Epochs	Learning rate (α)	Regul. weight (λ)	Epochs	Learning rate (α)	Regul. weight (λ)
Phase 1	5000	$1x10^{-2}$	$1x10^{-3}$	2000	$4x10^{-5}$	$4x10^{-5}$
Phase 2	5000	$1x10^{-3}$	N/A	5000	$2x10^{-3}$	$2x10^{-4}$
Phase 3	N/A	N/A	N/A	5000	$1x10^{-3}$	0

Table 2 – Learning rates and Regularization weights for each phase of training for both the kinematic and SHO models

Handling Bifurcation

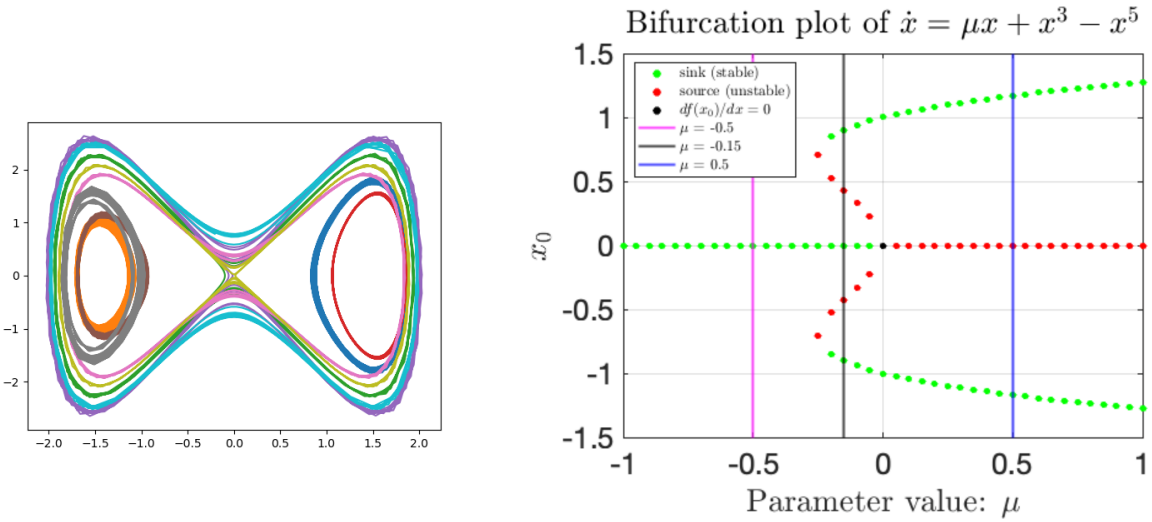

Fig. 4. Phase diagram (left) and bifurcation plot (right) of the nonlinear bifurcating dynamics (equation 3 from Table 1).

Initially, once the project had progressed enough to begin handling some of the more advanced dynamic models, the plan was to break up the data into regions if the full parameter space was not as easy to explore due to the function’s bifurcating nature. For the example, we see in Fig. 4 (right) that for a value of \mu = 0.5 there are 4 distinct dynamic regions for the behavior of x: $x < -1.2, -1.2 < x < 0, 0 < x < 1.2, 1.2 < x$. The plan would be to create 4 datasets, with each sampling x_1 from one of those regions, and then comparing the results. This method was not implemented due to complications stemming from the underlying model structure as discussed below.

Preliminary Results

Benchmarking

Kim published this model along with a simple benchmark code to evaluate the models' ability to perform simple regression using the 'primitive' functions within the EQL network's hidden layers on a variety of simple equations. My results are comparable to the original author's and given in Table 3.

$f(x,y)$	Success rate
x	1
x^2	0.55
x^3	0.05
$\sin(2x)$	0.6
xy	0.75
$sigmoid(-10x)$	0.7
e^{-x^2}	0.05
$x^2+\sin(2\pi y)$	0.3
x^2+y-2z	0.75

Table 3 – Success rate for specific functions in benchmarking the core regression algorithm in the EQL layer. Each function was regressed 20 times and the success rate is the number of times the model returned a solution with the same structure divided by 20 total attempts.

From Table 3, we can see that even some of the most basic functions (e.g. x^2 and x^3) only have a success rate of approximately 50%, even though these functions are explicit 'primitive' functions expressed in the regression network. One can only assume that the initial random weights expressed during instantiation must play a role in the success rate. This will come into play hevily in the following results.

Kinematics

For the simple kinematic solution, (equation 1), five trials were performed. As Seo et al describe the kinematic equation $\ddot{x} = F/m$ can be expressed as two finite-difference equations using Euler's method as follows:

$$x_{i+1} = x_i + \dot{x}_i + 0.5\ddot{x}$$
$$\dot{x}_{i+1} = \dot{x}_i + \ddot{x}$$

Where the i^{th}+1 step is a function of the i^{th} index of position and velocity, and a constant acceleration. With his in mind, the best latent representation found by the EQL was:

$$x_{i+1} = 0.998955x + 0.997594\dot{x}_i + 0.250238z$$
$$\dot{x}_{i+1} = 0.998197\dot{x}_i + 0.501092z$$

The error as stated by the NN was 0.00066.

Sampling from the same uniform distributions of the variables used in the network yields a linear correlation between the latent variable z and the desired variable $a = \ddot{x}$, and we can then extrapolate the value for z:

$$z = 1.9979\ddot{x} - 5.43EE-4$$

With the value of z now known, we can plot the trajectory of the true finite-difference equations with that of the found variables (Fig 2.). These results are consistent with the original paper, thus proving the application of the method.

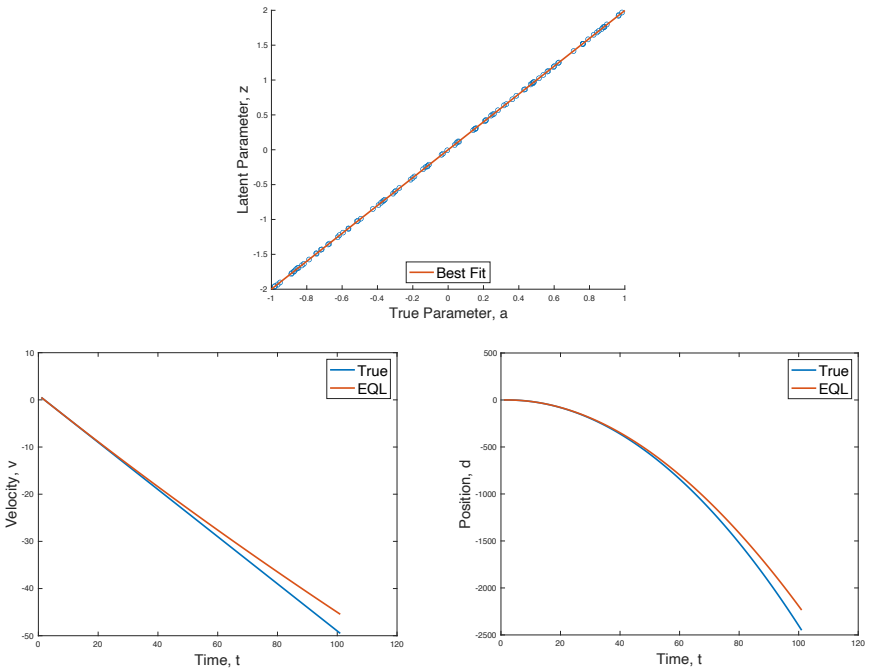

Fig. 5. (Top) The latent parameter z plotted as a function of $\ddot{x}$. (Bottom) Comparisons of the finite-difference equation for simple kinematics using the true solution (blue) and the EQL solved parameters (red) for the position (bottom left) and velocity (bottom right)

Simple Harmonic Oscillator (SHO)

For the SHO, (equation 2), ten trials were performed using the original network parameters as originally defined, and another 10 with a modification to the available function set.

Again, using Euler’s first-order finite difference approximation on the coupled ODEs $\dot{x} = \dot{x}$, $\ddot{x} = -(k/m)x$, and allowing $k/m = \omega^2$, and $\Delta t = 0.1$ we obtain:

$$x_{i+1} = x_i + \Delta t\, \dot{x}_i = x_i + 0.1\, \dot{x}_i$$
$$\dot{x}_{i+1} = \dot{x}_i - \Delta t x \omega^2 = \dot{x}_i - 0.1 x \omega^2$$

The best-found solution using the original network parameters was:

$$x_{i+1} = x_i + 0.099\dot{x}_i - 0.003$$
$$\dot{x}_{i+1} = -0.049x_i + 0.995\dot{x}_i + 0.001x_i\dot{x}_i - 0.0001\dot{x}_i^2 + (0.130x_i + 0.008\dot{x}_i)z$$

The error as stated by the NN was 0.002201.

This solution proved much more difficult to solve as the network insisted on including a $sin(x)$ term in the final solution. After much testing and messing with phase epoch lengths, and primitive functions, the end solution was to increase the threshold for the frozen weights from 0.01 to 0.05.

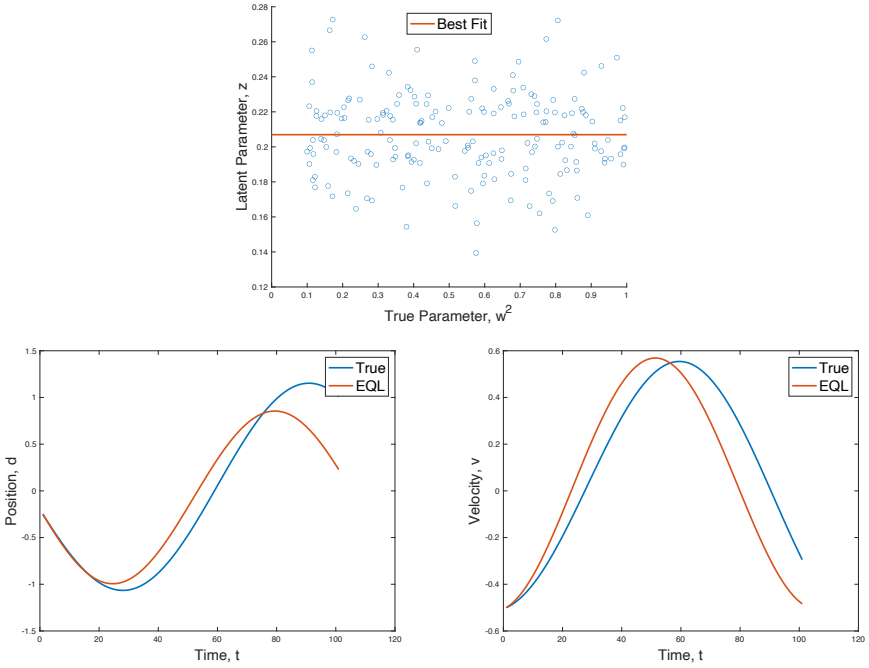

Fig. 6. (Top) The latent parameter z plotted as a function of ω^2. (Bottom) Comparisons of the finite-difference equation for the SHO using the true solution (blue) and the EQL solved parameters (red) for the position (bottom left) and velocity (bottom right)

Van der Pol Oscillator and Nonlinear w/ Bifurcation models

The neural network struggled with both of these models. After hours of playing with hyper parameters, the model never converged on a correct solution, though the error given was low. With over 100 terms in the x_{i+1} and $\dot{x}_{i+1}$ equations, the only notable part of the best solution found is the reported error at 0.018599 and 0.021516 for the Van der Pol and bifurcating models respectively. This level of reported error is on par with the SHO model, but the solutions are still clearly wrong. It does give us a clue on how to correct the problem moving forward, however.

Conclusion and Discussion

This model’s structure makes it very hard to tune for a given model simply due to the fact there are so very many hyperparameters to tune. There’s the learning rate, regularization weight and number of epochs must be determined for each phase of regression, as well as the threshold level for freezing the undesirable weights. For a regression model with 3 phases, that makes 9 different hyperparameters to tune up and down.

The model is also inflexible. While I am certain that it is possible to extend the model’s structure to accept a different input size and adjust the size of the latent dimension, I was unable to do so after a week’s worth of researching Tensorflow documents and code analysis. This would have enabled mode functions to be tested and a more rigorous testing routine.

The core problem, however, lies with the model’s inability to solve some of the most fundamental functions reliably. As discussed with table 3, the fact that the functions explicitly listed among the EQL networks primitives is not 100% should be reason for alarm. This means that for any input dynamics that are more complicated than the primitives will necessarily have a probability of correct solution based upon them. We can observe this directly from the benchmarking data:

(1)	x^2	0.55
(2)	$\sin(2x)$	0.6
(3)	$x^2 + \sin(2\pi y)$	0.3

The probability for success of function (3) can be predicted by the success of (1) and (2): $p(3) \approx p(1)p(2)$. By this measure, the Van der Pol model would have had $(0.75)(0.75)(0.75)(0.55) \approx 0.23$ success rate under optimal training conditions. The nonlinear model has slimer chances: $(0.75)(0.05)(0.05)(0.55) < 0.001$.

There are a few ways of correcting this. The regularization could be changed to include a term that penalizes non-sparse solutions, or possibly optimizes the thresholding term instead. Additionally, since the error rates reported were so low, it would follow that they may in fact be a reasonable approximation over small Δt. An appropriate solution for this would just be to use longer data series for training. This limits application for complex systems however, as they will typically approach a stability point after which time series data may become static, or cyclical.

Using neural networks has been shown to be a useful tool for data-driven system identification, and autoencoders play a useful role in the literature. However, something mode akin to a genetic algorithm in on the latent-space variables may prove to be more reliable. In a more broad sense, a GNN may be a more appropriate network structure for this application.

References

[1]. Champion, Kathleen, et al. "Data-driven discovery of coordinates and governing equations." Proceedings of the National Academy of Sciences 116.45 (2019): 22445-22451.

[2]. Cranmer, Miles, et al. "Discovering symbolic models from deep learning with inductive biases." Advances in Neural Information Processing Systems 33 (2020): 17429-17442

[3]. Cybenko, George. "Approximation by superpositions of a sigmoidal function." Mathematics of control, signals and systems 2.4 (1989): 303-314

[4]. Kim, Samuel, et al. "Integration of neural network-based symbolic regression in deep learning for scientific discovery." *IEEE transactions on neural networks and learning systems* 32.9 (2020): 4166-4177.

[5]. Lu, Peter Y., Samuel Kim, and Marin Soljačić. "Extracting interpretable physical parameters from spatiotemporal systems using unsupervised learning." Physical Review X 10.3 (2020): 031056

[6]. Manzi, Matteo, and Massimiliano Vasile. "Orbital anomaly reconstruction using deep symbolic regression." *71st International Astronautical Congress*. 2020. Fortin, Félix-Antoine, et al. "DEAP: Evolutionary algorithms made easy." *The Journal of Machine Learning Research* 13.1 (2012): 2171-2175.

[7]. Schmidt, Michael, and Hod Lipson. "Distilling free-form natural laws from experimental data." *science* 324.5923 (2009): 81-85

[8]. de Silva, Brian M., et al. "Pysindy: a python package for the sparse identification of nonlinear dynamics from data." arXiv preprint arXiv:2004.08424 (2020).

[9]. Zhang, Michael, et al. "Deep learning and symbolic regression for discovering parametric equations." *arXiv preprint arXiv:2207.00529* (2022).

[10]. Zheng, David, et al. "Unsupervised learning of latent physical properties using perception-prediction networks." arXiv preprint arXiv:1807.09244 (2018)

[11]. Kim, Samuel. "DeepSymReg" https://github.com/samuelkim314/DeepSymReg *Github Repo* (2022)