DynaDojo: An Extensible Platform for Benchmarking Scaling in Dynamical System Identification

Logan Mondal Bhamidipaty[*]
Stanford University
loganmb@cs.stanford.edu

Tommy Bruzzese[*]
Stanford University
tbru@cs.stanford.edu

Caryn Tran[*]
Northwestern University
caryn@u.northwestern.edu

Rami Mrad
UC Berkeley
ramiratlmrad@berkeley.edu

Maxinder S. Kanwal[†]
Stanford University
kanwal@stanford.edu

Abstract

Modeling complex dynamical systems poses significant challenges, with traditional methods struggling to work on a variety of systems and scale to high-dimensional dynamics. In response, we present DynaDojo, a novel benchmarking platform designed for data-driven dynamical system identification. DynaDojo provides diagnostics on three ways an algorithm's performance scales: across the number of training samples, across the complexity of a dynamical system, and across resources needed to maintain a target error. Furthermore, DynaDojo enables studying out-of-distribution generalization (by providing unique test conditions for each system) and active learning (by supporting closed-loop control). Through its user-friendly and easily extensible API, DynaDojo accommodates a wide range of user-defined `Algorithms`, `Systems`, and `Challenges` (evaluation metrics). The platform also prioritizes resource-efficient training with parallel processing strategies for running on a cluster. To showcase its utility, in DynaDojo `0.9`, we include implementations of 7 baseline algorithms and 20 dynamical systems, along with several demos exhibiting insights researchers can glean using our platform. This work aspires to make DynaDojo a unifying benchmarking platform for system identification, paralleling the role of OpenAI's Gym in reinforcement learning.[1]

1 Introduction

Dynamical systems, fundamental to disciplines like physics, engineering, economics, and neuroscience, are difficult to predict and control when they exhibit nonlinear and high-dimensional behaviors. Traditional methods, which rely on known underlying equations, fall short when faced with modern problems like stock market forecasting or modeling human social interactions, where such equations are unknown or non-existent. This has prompted a shift toward data-driven computational modeling—leveraging machine learning to learn directly from measured data (bypassing the need for predefined equations)—known as *system identification* [1]. To benefit from these data-driven approaches, however, researchers and practitioners need tailored benchmarks to easily evaluate and compare methods for system identification in their area of study. In this work, we present DynaDojo, a novel benchmarking platform, modeled after OpenAI's Gym [2] and Procgen [3], to standardize benchmarking *any* algorithm on *any* dynamical system.

[*]Equal contribution

[†]Corresponding Author

[1]Code available at `https://github.com/FlyingWorkshop/dynadojo` for access to the implemented challenges, systems, baseline models, and analysis code.

37th Conference on Neural Information Processing Systems (NeurIPS 2023) Track on Datasets and Benchmarks.

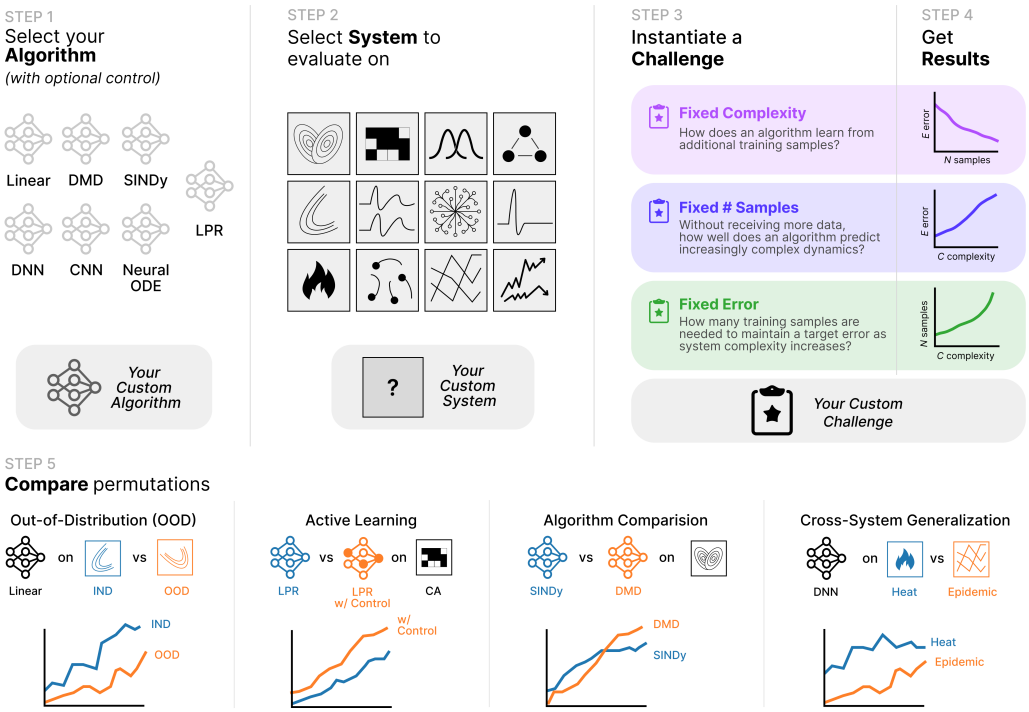


Figure 1: Pipeline for how to use DynaDojo. Select an algorithm and a system, then instantiate a challenge with them, and evaluate to get results. It is easy to run repeated DynaDojo challenges to compare performance on out-of-distribution data, with active learning, between algorithms, and across systems. See Section 3.2.1 and Figures 7, 6, 8 and 9 for examples of DynaDojo results.

2 Overview of DynaDojo

DynaDojo operates on three core objects: `Algorithms`, `Systems`, and `Challenges`.[2] Run a `Challenge` with any given `Algorithm` and `System` to evaluate how the algorithm's performance scales when fixing either the complexity of a system, the number of training samples, or a target error to achieve (see Figure 1). DynaDojo currently provides a suite of 7 algorithms, 20 systems, and 3 challenges to be used. Additionally, DynaDojo provides abstract interfaces for `Algorithms`, `Systems`, and `Challenges` that can be extended to support custom implementations (see Figure 11).

2.1 Algorithms

Algorithms are subclasses of `AbstractAlgorithm`, an interface designed to wrap any learning algorithm that can be used for system identification. At initialization, algorithms are provided the dimensionality of the data which can be used to define the appropriate parameters for the algorithm, for example, the number of layers in a neural network or degree of a polynomial in a polynomial regression. The `AbstractAlgorithm` interface further abstracts the details of the algorithm under `fit`, `predict`, and `act` methods for ease of use and simplicity. See Appendix C for additional details on the 7 algorithms included in DynaDojo.

2.2 Systems

Systems are subclasses of `AbstractSystem`, an interface to wrap any procedurally-generated dynamical systems. Systems are initialized with a latent dimension and embedding dimension which

[2]We indulge in a minor abuse of notation. Technically, DynaDojo only implements `AbstractAlgorithm`, `AbstractSystem`, and `Challenge`. We use `Algorithms` and `Algorithm` to refer to concrete subclasses of `AbstractAlgorithm`; `System` and `Systems` to refer to concrete cublasses of `AbstractSystem`; and the plural `Challenges` to refer to multiple `Challenge` classes.

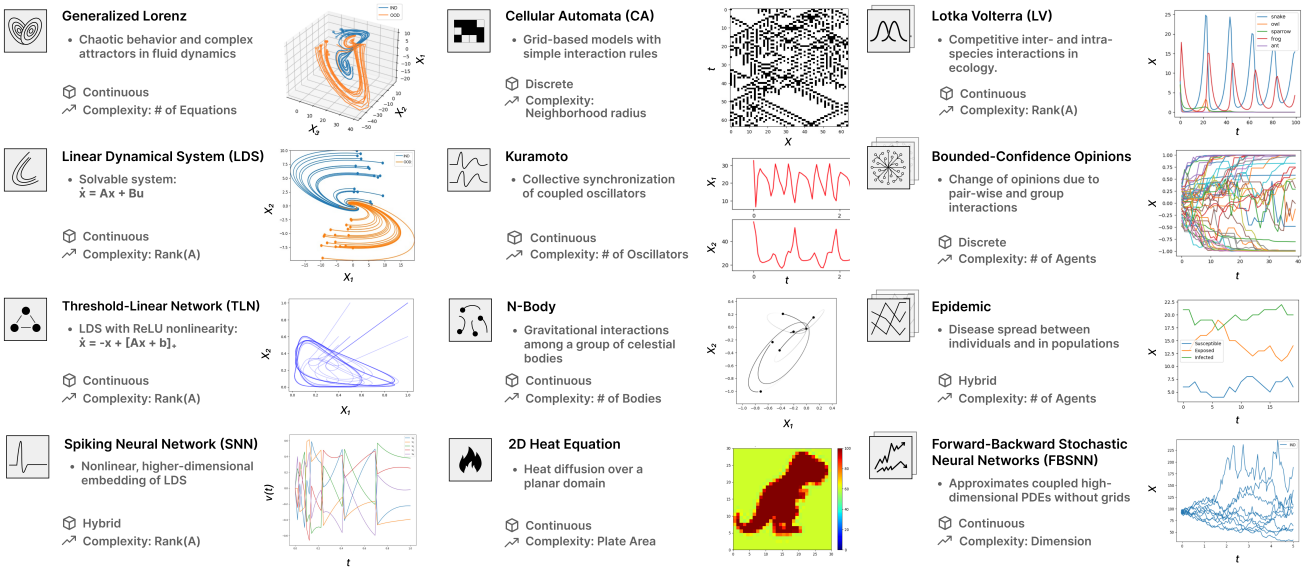


Figure 2: The 20 dynamical systems currently packaged in DynaDojo. Systems are annotated with the system type (discrete, continuous, or hybrid) and their tunable measure of complexity. We implement two types of Lotka-Volterra (Competitive and Prey-Predator); five types of Bounded-confidence opinion systems (Algorithmic Media Bias [4], Deffuant [5], Hegselmann-Krause (HK) [6], Weighted HK (WHK) [7], and Attraction-Repulsion WHK [7]); three types of epidemic systems (SIR, SIS, and SEIS), and two types of FBSNNs (Black-Scholes-Barenblatt (BSB) [8], Hamilton-Jacobi-Bellman (HJB) [8])

determine the complexity of the system and its generated trajectories. The `AbstractSystem` interface abstracts the details of simulating the system under the `make_init_conditions` and `make_data` methods. Initial conditions can be produced in-distribution or out-of-distribution. Data can be generated with noise or optional control inputs. The `AbstractSystem` interface also abstracts the evaluation metric for any particular dynamical system with the `calc_error` method. For example, continuous systems, such as linear dynamical systems, might implement mean-squared error to evaluate the accuracy of predicted trajectories, whereas a binary system, such as cellular automata, might use Hamming distance instead. Other system-specific metrics, such as achieving stability with control, can likewise be defined in `calc_error`. See Figure 2 for the systems that we package with DynaDojo and Figure 10 for the adjustable parameters available for systems.

2.3 Challenges

Challenges are subclasses of `Challenge`, an interface for orchestrating evaluation of algorithms on systems.[3] The primary use of challenges is to simplify and parallelize the process of repeatedly training and testing algorithms on systems, especially when scaling parameters of the system, algorithm, or training process. In a challenge, the `evaluate` method manages repeated and parallelized trials of algorithm instantiation, system instantiation, data generation, testing (see pseudo-code in Algorithm 1 of Appendix A). The `plot` method visualizes the results of the challenge.

In DynaDojo `0.9`, we provide three challenges to evaluate scaling: `FixedComplexity`, `FixedTrainSize`, and `FixedError`. In `FixedComplexity`, we repeatedly train and test an algorithm on a system of fixed complexity while scaling the number of training samples. In `FixedTrainSize`, we repeatedly train and test an algorithm on systems of increasing complexity while fixing the number of training samples. In `FixedError`, we search for the number of training

[3]Unlike `AbstractSystem` and `AbstractAlgorithm`, `Challenge` is not an abstract base class and can be used off-the-shelf. While subclasses of `Challenge` suffice for most experiments, advanced developers who want to implement a bespoke challenge may chose to use `Challenge` rather than a subclass because it exposes more parameters.

samples necessary for an algorithm to achieve a target error rate on systems of increasing complexity. In Figure 3, we visualize the relationship between these scaling challenges.

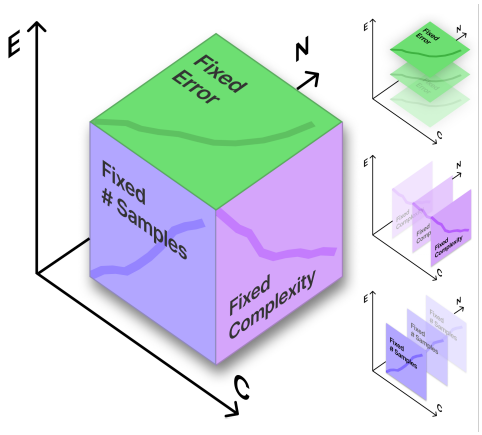


Figure 3: DynaDojo challenges provide a snapshot of an algorithm's scaling behavior along one slice of a "performance landscape" by varying three dimensions: system complexity (C), error (E), and number of training samples (N). Each challenge varies parameters along one dimension while holding another constant, in order to measure the algorithm's performance on the third.

3 Example usage

3.1 Running a single algorithm on a single system

DynaDojo algorithms and systems can be used independently of challenges. To train and test a single algorithm instance on a single system instance, first instantiate the system and create the training and test data (which, in this example, is OOD).

```
latent_dim, embed_dim, train_size, test_size, timesteps = (3, 3, 50, 10, 50)
lorenz= LorenzSystem(latent_dim, embed_dim, seed=100)
x0 = lorenz.make_init_conds(n)
y0 = lorenz.make_init_conds(30, in_dist=False)
x = lorenz.make_data(x0, control=np.zeros((n, timesteps, embed_dim)),
    timesteps=timesteps)
y = lorenz.make_data(y0, control=np.zeros((test_size, timesteps, embed_dim)),
    timesteps=timesteps, noisy=True)
```

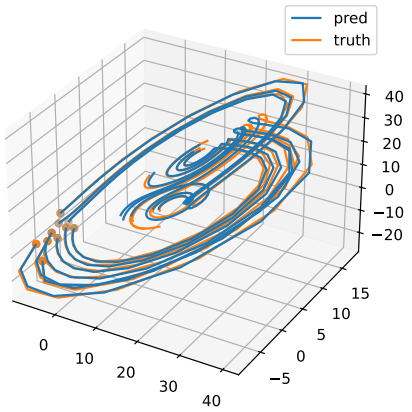
Then instantiate the algorithm, fit on the training data, predict and calculate error. Plotting utilities, demos, and examples are provided in our GitHub repository.

```
sindy = SINDy(embed_dim, timesteps,
    seed=100)
sindy.fit(x)

# predict trajectories
y_pred = sindy.predict(y[:, 0],
    timesteps)
error = lorenz.calc_error(y, y_pred)

fig, ax = dynadojo.utils.plot([y_pred,
    y], target_dim=min(3, latent_dim),
    labels=["pred", "truth"],
    max_lines=15)
```

(a) Code for fitting and testing `SINDy` on a `LorenzSystem` using `DynaDojo`. Plotting utilities are provided to visualize high dimensional systems.



(b) Plot of `SINDy` algorithm prediction for a Lorenz system showing high overlap with ground truth.

3.2 Running a challenge

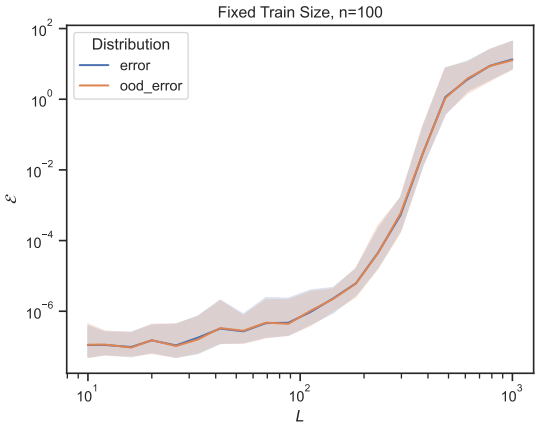
To evaluate, for example, how linear regression generalizes to linear dynamical systems of increasing complexity, run `FixedTrainSize`. First, decide on the complexities (latent dimensions) to scale across, the number of training samples, the number of trials, and the training and testing conditions. Supply these arguments to instantiate a `FixedTrainSize` challenge. Then, evaluate the challenge with the algorithm class and plot the results. By default, challenges are run without parallelization; however, code examples showing parallelization across cores and computers are provided on GitHub.

```python
challenge = FixedTrainSize(
    Latent_dims=np.logspace(1, 3, 20, include_end=True),
    train_size=100,
    timesteps=50,
    control_horizons=0,
    max_control_cost_per_dim=0,
    system_cls=LDSSystem,
    trials=100,
    test_examples=50,
    test_timesteps=50 )

data = challenge.evaluate(
    LinearRegression,
    noisy=True,
    ood=True,
    seed=1027 )

g = FixedTrainSize.plot(data)
```

(a) Code for running `FixedTrain` challenge with on a training set size of 100 for `LinearRegression` on `LDSSystem` with latent dimensions from 10 to 1000.



(b) `FixedTrainSize` plot for linear regression on a linear dynamical system with 100 training samples. Latent dimensionality (L) is plotted on the x-axis and error (E) on the y-axis. Curves for testing on in-distribution and out-of-distribution (OOD) data are overlapping, showing OOD generalization. After L = 100, error rapidly increases, showing weaker sample efficiency with respect to system complexity.

3.2.1 Analyses

With a suite of baseline algorithms and dynamical systems, DynaDojo is designed to support running repeated challenges to analyze algorithms across systems, schematically depicted in Figure 1.

Out-of-Distribution Generalization To test how an algorithm generalizes to out-of-distribution (OOD) data, run a challenge with the `ood` parameter enabled. Challenges will test the algorithm on data simulated *both* from initial conditions drawn from the same distribution—as the training set initial conditions—and from those drawn OOD. See Figure 6 for an example analysis of OOD generalization in a `FixedComplexity` challenge.

Active Learning DynaDojo algorithms can optionally implement an `act` method which generates control inputs that DynaDojo systems accept when generating data. To compare the effect of active learning, run a challenge for a given algorithm without control and the same algorithm with control on a given system. See Figure 7 for an example analysis of active learning in a `FixedError` challenge.

Comparing Algorithms To compare algorithms, run a challenge for two different algorithms on the same system. See Figure 8 for an example comparison of two algorithms in a `FixedError` challenge.

Cross-System Generalization To investigate whether an algorithm's performance is generalizable across systems, run a challenge with an algorithm on two different systems. See Figure 9 for an example analysis of cross-system generalization in a `FixedTrainSize` challenge.

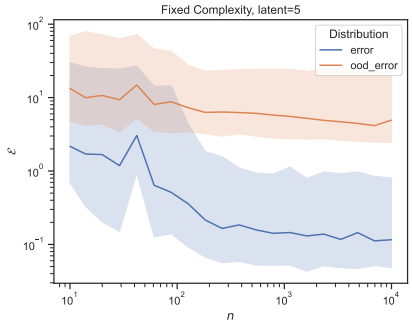


Figure 6: Comparing OOD generalization for deep neural networks (DNN) on linear dynamical systems (latent dimension 5) in a `FixedComplexity` challenge. In-distribution test error (blue line) is decreasing steeply but OOD test error (orange line) is constant as number of training samples (x-axis) increases, showing lack of OOD generalization as training size scales.

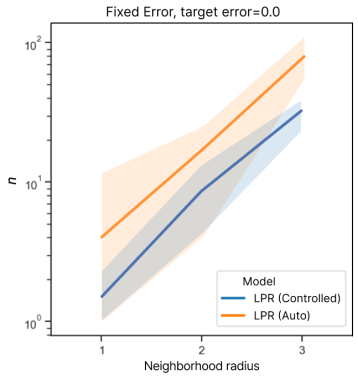
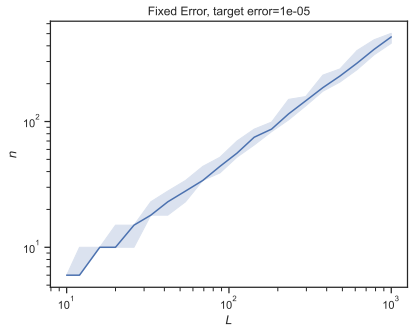


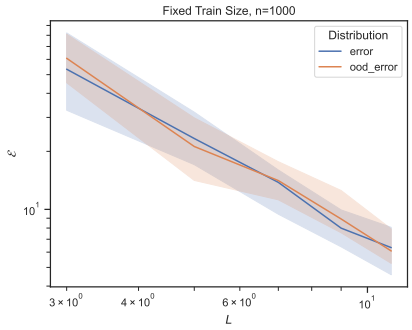
Figure 7: Comparing active learning for Lowest Possible Radius (LPR) on cellular automata (CA) in a `FixedError` challenge. As CA complexity (x-axis) scales, LPR w/ control (blue line) requires less training samples (y-axis) than LPR w/o control (orange line) to achieve zero error.



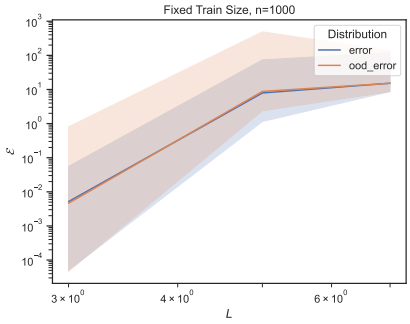
(a) `FixedError` for LR on LDS, target error=1e-5.

(b) `FixedError` for DNN on LDS, target error=1e0.

Figure 8: Comparing two algorithms: Linear regression (LR) and deep neural network (DNN) are evaluated on linear dynamical systems in a `FixedError` challenge. As complexity (x-axis) of the system scales, LR (left) achieves a much lower target error of 10^{-5} with fewer or comparable number of training samples (y-axis) than needed by a DNN (right) for a *higher* target error rate of 1.



(a) `FixedTrainSize` with SINDy and Lorenz.



(b) `FixedTrainSize` with SINDy and LDS.

Figure 9: Cross-system generalization: SINDy is evaluated on Lorenz Systems (left) and linear dynamical systems (LDS) (right) in a `FixedTrainSize` challenge. Error (y-axis) decreases with complexity (x-axis) for Lorenz, but increases and plateaus with complexity for LDS. Also, OOD (orange) and in-distribution (blue) error are matched, showing OOD generalization.

4 Design

Benchmark platform, not datasets. Rather than designing a single or suite of benchmarks, DynaDojo is a benchmarking *platform*. That is, we provide dynamical systems to benchmark on *and* baseline algorithms to compare against. In addition, scaling challenges orchestrate and parallelize the benchmarking process. We take this approach to standardize the process of benchmarking and unify benchmarks under one interface. To show the versatility of DynaDojo as a benchmarking platform, we provide numerous example Jupyter notebooks for implemented algorithms, systems, and challenges, available in our GitHub repository.

Simple by default. A key challenge in system identification is being able to compare one algorithm against other algorithms on many systems. This is difficult because various existing models and systems either lack a simple API or all have different APIs. Wrestling together APIs and setting the right parameters is a barrier to benchmarking in system identification. We designed DynaDojo with a focus on simplicity of the API to ensure that different algorithms can be painlessly run on any system. To enable this, all systems and algorithms must come with default presets, with optional overrides. This comes at a cost of developers having to determine reasonable or adaptive settings for the algorithms or systems they contribute.

Procedurally generated data. Rather than a static dataset, we choose to have dynamical systems in DynaDojo procedurally generate data at train and test time in order to support active learning, where system trajectories are altered by control inputs provided from an algorithm. Additionally, we require a tunable, continuous measure of complexity for each system to support scaling metrics (see Figure 3). This requires that dimensionality of system trajectories must be dependent on the system complexity–another reason for procedural data generation. Compared to pre-computed datasets, procedural generation of data can be costly especially for systems of high complexity.

Focused on scaling, not performance. Often, algorithms are evaluated on their performance for a single task. On the contrary, we are focused on how that performance scales as the task gets harder, or more data is provided (see Figure 3). Specifically in system identification, we saw an opportunity to numerically define task difficulty via dynamical system complexity measures. With benchmarking scalability as our goal, it became necessary to implement challenges to orchestrate the scaling process and to design abstractions over algorithms *and* systems to enable this. To ameliorate the inflexibility of requiring interfaces over both algorithms and systems, we designed them to be decoupled from another and thus independently usable outside of DynaDojo (see Section 3.1). To evaluate scaling, many rounds of instantiating algorithms and systems and training them must be repeated as parameters scale. This can be very resource intensive so we implemented challenges to support parallelization across cores and computers. We also packaged a DynaDojo Docker container to be easily used on cluster environments.

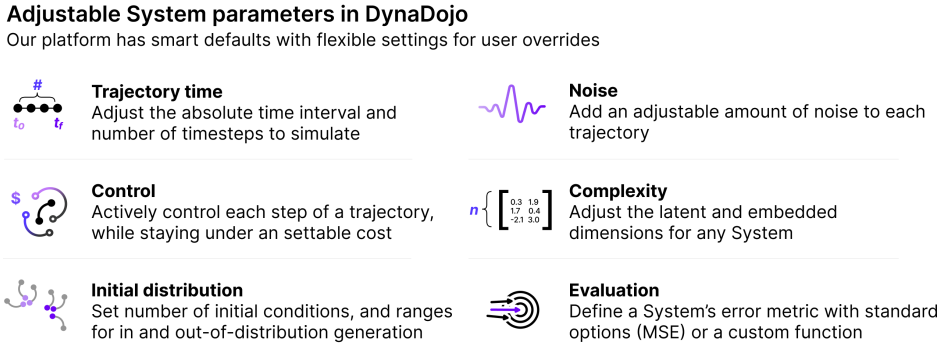


Figure 10: Users can adjust nearly all DynaDojo parameters, including how trajectories are initialized, simulated, and controlled, as well as system complexity and evaluation.

Flexible and extensible. We support a variety of settings to enable features such as OOD data generation and control inputs in order to ensure DynaDojo covers broad use-cases (Figure 10).

As mentioned above, we isolate algorithm, system, and challenge implementations so users can choose to use whatever parts of DynaDojo are relevant to their work. Additionally, we provide extensible interfaces for algorithms, systems, and challenges to ensure that DynaDojo can be adapted to use-cases we have not covered (Figure 11).

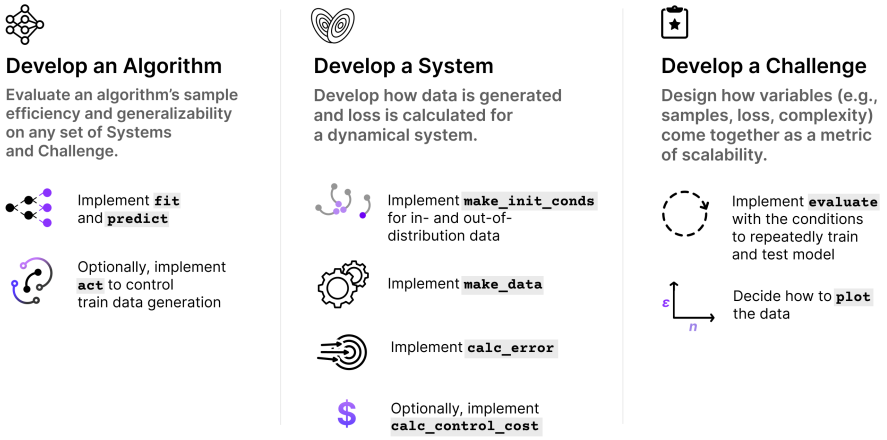


Figure 11: Users can extend DynaDojo by implementing your own algorithm, system, or challenge.

5 Related work

5.1 Benchmarks and benchmarking frameworks

System identification focuses on accurately modeling the dynamics of a system, whereas reinforcement learning (RL) aims to optimize actions within a system to maximize rewards. Despite their distinct objectives, they both seek to model and interact with complex environments and can often be used in solving similar or overlapping problems. Given their overlap, we draw upon and connect literature from both areas to motivate the development of our benchmarking platform.

Single-System Benchmarks In the field of system identification, benchmarks have been created that measure one specific dynamical system from a specific physical phenomena [9, 10]. These single-system benchmarks have been used to evaluate specific model architectures [11] or to compare different approaches [12]. These narrowly-focused benchmarks do not facilitate the evaluation of a method's generalization across a diverse set of systems, which is a central aim of our work.

Benchmark Suites Benchmark suites, such as [13, 14, 15, 16], offer a broader scope for evaluating system identification by covering a larger subset of system classes. These suites, however, are restricted to specific types of systems or representations, such as chaotic systems [13, 14], physical systems [15], or partial differential equations [16]. Our platform takes a similar benchmark suite approach but is designed to be extensible, supporting the addition of diverse dynamical systems through a common interface. For a comparison of DynaDojo with existing benchmarks, see Table 1.

OpenAI Gym To address the fragmentation and lack of standardization in system identification benchmarks, we adopt a similar approach to OpenAI's Gym environment [2]. OpenAI Gym offers a common interface for RL benchmarking tasks and, with over 5000 citations, has become a standard benchmark framework in RL [2]. Our work creates an extensible, standardized gym-like platform to unify system identification benchmarks. Contrary to OpenAI's focus on environments, not agents [2]—which are analogous to DynaDojo's systems and algorithms, respectively—we provide abstractions over both entities and implement challenges that orchestrates the process of benchmarking scaling. In Sections 4 and 5.2, we explain the motivation behind this decision and related works.

5.2 Generalization and scaling

We draw upon literature to motivate the settings and challenges implemented in our platform.

Table 1: Comparison with Related Benchmarks

	# Systems	Extensible	OOD	Control	SE	Complexity Tunability	Evaluation
Single-System [9, 10]	1						
PDEBench [16]	11	✓	✓			Fixed, Quantitative	TS
Otness et al [15]	4	✓	✓			Limited	
Chaos [13]	131					Fixed, Quantitative	TS
New et al [17]	2					Continuous, Quantitative	TS
Procgen [3]	16 Games	✓	✓	✓	✓	Binary (Easy/Hard)	TS
DynaDojo	20	✓	✓	✓	✓	Continuous, Quantitative	TS, FT, FC, FE

DynaDojo, to the best of knowledge, is the first extensible dynamical systems benchmark that supports control and evaluates Out-of-Distribution (OOD), Complexity, and Sample Efficiency (SE). (TS = Test Set, FT=Fixed Training, FC=Fixed Complexity, FE=Fixed Error)

Out-of-Distribution Generalization In system identification, there is interest in understanding how well models can generalize beyond the training distribution. [18] probes how deep learning models generalize to trajectories from out-of-distribution initial conditions for dynamical systems. [19] investigates whether deep neural networks learning cellular automata show out-of-distribution generalization for unseen initial configurations with different rule sets. Recent RL benchmarks have sought to split train and test data to draw from different gaming environments in response to problems of overfitting on training environments [20, 21]. Motivated by this work, DynaDojo enables algorithm evaluation on both in-distribution and out-of-distribution data.

Active Learning In RL, active learning—where the algorithm intelligently selects control inputs to explore state-space—enhances sample efficiency. While dynamical systems can accept control inputs and therefore can support active learning, this property is often underutilized in system identification benchmarks that predominantly use static datasets [13, 15, 16, 9, 10]. In our work, we enable algorithms within the benchmark to provide control inputs to dynamical systems, thereby facilitating active learning approaches and interactive data generation.

Scaling Complexity There is a desire to understand how algorithms perform on systems of varied complexity. For example, within system identification, [19] and [22] test deep neural networks on their generalization capabilities for learning cellular automata with varied neighborhood sizes.

In RL, benchmarks have been designed to train and test algorithms on game environments with varying difficulty levels [3]. This work on games, however, uses imprecise notions of difficulty which are calibrated roughly on the training speeds of baseline algorithms, but this definition is inconvenient because it is hard to generalize and test.

In system identification, [13] provides a suite of over a hundred benchmark datasets of known chaotic dynamical systems. Each system is annotated with mathematical properties reflecting the complexity of the system. These annotations facilitate comparison of learning methods across dynamical systems of varying complexity; our work is similarly motivated. In DynaDojo, instead of providing fixed datasets corresponding to different complexity levels, dynamical system classes are defined to allow their complexity levels to be programmatically scaled.

Cross-System Generalization While we are unaware of any work in system identification that explores generalization across different systems, the concept has been explored in RL. Generalization in RL has expanded from considering unseen states to learning across different domains, as exemplified by AlphaZero's ability to learn Go, Chess, and Shogi [23], compared to AlphaGo's specialization in Go [24]. It is also similar the field of multi-task learning in which one general policy might train on several different environments [25]. This trend is reflected in RL benchmarks and toolkits that measure performance across a diverse variety of environments [2, 3]. Our work takes an analogous approach by facilitating the easy evaluation of an algorithm's performance across different classes of dynamical systems, thereby capturing the algorithm's cross-system generalization capabilities.

Scaling Training Size Sample efficiency, the ability to learn effectively from a limited number of training samples, is particularly relevant as it directly influences an algorithm's performance in real-world scenarios where data may be scarce, expensive to obtain, or hard to simulate [26].

In DynaDojo, we've designed metrics that measure how a algorithm's performance scales with changes in system complexity and training dataset size. This feature enables users to assess an algorithm's sample efficiency, particularly as it handles increasing complexity.

6 Future work

DynaDojo is still a work in progress. As of our most recent release, our platform has certain limitations worth noting: While we implement parallel computing across all `Challenges`, the `FixedError` challenge is still especially resource-intensive and susceptible to noise. We aim to implement more sophisticated root-finding search algorithms to replace our exponential search.

To improve the rigor of our scaling results, we would like to move to a more objective measure of system complexity (e.g., the intrinsic dimension of the objective landscape [27]). We also seek to develop scaling metrics that summarize the results plotted by DynaDojo. Furthermore, we would like to define a unified generalization metric that captures an algorithm's capacity to work on OOD test data, across scales of complexity, and on different dynamical systems altogether.

We will, of course, always look for ways to include more state-of-the-art algorithms and dynamical systems of interest. In particular, we would like to develop simple wrappers for OpenAI Gym environments and algorithms to immediately accommodate their vast library, and vice-versa, wrapping DynaDojo `Systems` and `Algorithms` for OpenAI Gym.

To further broaden the scope and applicability of DynaDojo, we plan on introducing new `Challenges` of interest to the community. For example, we aim to incorporate an optimal control challenge involving stabilizing a system around a target trajectory, a transfer learning challenge focusing on fine-tuning to new system data, a multi-task learning challenge around maintaining consistent performance across multiple dynamical systems, and a curriculum learning challenge focusing on leveling up system complexity without retraining an algorithm from scratch. All of the `Challenges` can also be extended to measure prediction/control error on multiple timescales.

Lastly, in light of the emerging importance of scaling laws in deep learning, we hope to incorporate new scaling dimensions. These will include the number of model parameters, computational cost of training, and activation sparsity, on top of the three existing scaling dimensions. By including these dimensions, we aim to offer a more comprehensive view of how algorithms scale.

Acknowledgments

MSK was supported by NSF GRFP, TB by Stanford's Bioengineering REU program, LMB by Stanford's CURIS program. The authors thank the Stanford Brains in Silicon Lab and Delta Lab at Northwestern for helpful feedback. This work used Stanford's Sherlock and Northwestern's Quest clusters.

References

[1] Oliver Nelles. *Nonlinear dynamic system identification*. Springer, 2020.

[2] Greg Brockman et al. "OpenAI Gym". In: *CoRR* abs/1606.01540 (2016). arXiv: 1606.01540. URL: http://arxiv.org/abs/1606.01540.

[3] Karl Cobbe et al. "Leveraging Procedural Generation to Benchmark Reinforcement Learning". In: *CoRR* abs/1912.01588 (2019). arXiv: 1912.01588. URL: http://arxiv.org/abs/1912.01588.

[4] Valentina Pansanella, Giulio Rossetti, and Letizia Milli. "Modeling algorithmic bias: simplicial complexes and evolving network topologies". In: *Applied Network Science* 7.1 (2022), p. 57.

[5] Guillaume Deffuant et al. "Mixing beliefs among interacting agents". In: *Advances in Complex Systems* 3.01n04 (2000), pp. 87–98.

[6] Hegselmann Rainer and Ulrich Krause. "Opinion dynamics and bounded confidence: models, analysis and simulation". In: (2002).

[7] Letizia Milli. "Opinion dynamic modeling of news perception". In: *Applied Network Science* 6.1 (2021), pp. 1–19.

[8] Maziar Raissi. "Forward-backward stochastic neural networks: Deep learning of high-dimensional partial differential equations". In: *arXiv preprint arXiv:1804.07010* (2018).

[9] Florent Bonnet et al. "An extensible Benchmarking Graph-Mesh dataset for studying Steady-State Incompressible Navier-Stokes Equations". In: *ICLR 2022 Workshop on Geometrical and Topological Representation Learning*. 2022. URL: https://openreview.net/forum?id=rqUUi4-kpeq.

[10] Jean-Philippe Noel and Maarten Schoukens. "Hysteretic benchmark with a dynamic nonlinearity". In: *Workshop on nonlinear system identification benchmarks*. 2016, pp. 7–14.

[11] Max Schüssler, Tobias Münker, and Oliver Nelles. "Deep Recurrent Neural Networks for Nonlinear System Identification". In: *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*. 2019, pp. 448–454. DOI: 10.1109/SSCI44817.2019.9003133.

[12] Max Schüssler, Tobias Münker, and Oliver Nelles. "Local Model Networks for the Identification of Nonlinear State Space Models". In: *2019 IEEE 58th Conference on Decision and Control (CDC)*. 2019, pp. 6437–6442. DOI: 10.1109/CDC40024.2019.9028945.

[13] William Gilpin. "Chaos as an interpretable benchmark for forecasting and data-driven modelling". In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*. 2021. URL: https://openreview.net/forum?id=enYjtbjYJrf.

[14] Luã Streit, Vikram Voleti, and Tegan Maharaj. "Understanding Generalization and Robustess of Learned Representations of Chaotic Dynamical Systems". In: *ICML 2022: Workshop on Spurious Correlations, Invariance and Stability*. 2022. URL: https://openreview.net/forum?id=sJg-BRQcLxr.

[15] Karl Otness et al. "An Extensible Benchmark Suite for Learning to Simulate Physical Systems". In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*. 2021. URL: https://openreview.net/forum?id=pY9MHwmrymR.

[16] Makoto Takamoto et al. "PDEBench: An Extensive Benchmark for Scientific Machine Learning". In: *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. 2022. URL: https://openreview.net/forum?id=dh_MkX0QfrK.

[17] Alexander New et al. "Tunable complexity benchmarks for evaluating physics-informed neural networks on coupled ordinary differential equations". In: *2023 57th Annual Conference on Information Sciences and Systems (CISS)*. IEEE. 2023, pp. 1–8.

[18] Rui Wang et al. "Learning dynamical systems requires rethinking generalization". In: *NeurIPS 2020 Workshop on Interpretable Inductive Biases and Physically Structured Learning*. 2020. URL: https://www.amazon.science/publications/learning-dynamical-systems-requires-rethinking-generalization.

[19] Marcel Aach, Jens Henrik Göbbert, and Jenia Jitsev. "Generalization over different cellular automata rules learned by a deep feed-forward neural network". In: *CoRR* abs/2103.14886 (2021). arXiv: 2103.14886. URL: https://arxiv.org/abs/2103.14886.

[20] Karl Cobbe et al. "Quantifying generalization in reinforcement learning". In: *International Conference on Machine Learning*. PMLR. 2019, pp. 1282–1289.

[21] Alex Nichol et al. “Gotta learn fast: A new benchmark for generalization in rl”. In: *arXiv preprint arXiv:1804.03720* (2018).

[22] William Gilpin. “Cellular automata as convolutional neural networks”. In: *Phys. Rev. E* 100 (3 Sept. 2019), p. 032402. DOI: 10.1103/PhysRevE.100.032402. URL: https://link.aps.org/doi/10.1103/PhysRevE.100.032402.

[23] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419 (2018), pp. 1140–1144. DOI: 10.1126/science.aar6404. eprint: https://www.science.org/doi/pdf/10.1126/science.aar6404. URL: https://www.science.org/doi/abs/10.1126/science.aar6404.

[24] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *nature* 529.7587 (2016), pp. 484–489.

[25] Tianhe Yu et al. “Meta-World: A Benchmark and Evaluation for Multi-Task and Meta Reinforcement Learning”. In: *CoRR* abs/1910.10897 (2019). arXiv: 1910.10897. URL: http://arxiv.org/abs/1910.10897.

[26] Andrei Kramer, Ben Calderhead, and Nicole Radde. “Hamiltonian Monte Carlo methods for efficient parameter estimation in steady state dynamical systems”. In: *BMC bioinformatics* 15 (2014), pp. 1–11.

[27] Chunyuan Li et al. “Measuring the Intrinsic Dimension of Objective Landscapes”. In: *International Conference on Learning Representations*. 2018.

[28] Bo-Wen Shen. “Aggregated negative feedback in a generalized Lorenz model”. In: *International Journal of Bifurcation and Chaos* 29.03 (2019), p. 1950037.

[29] Martin Boerlin, Christian K Machens, and Sophie Denève. “Predictive coding of dynamical variables in balanced spiking networks”. In: *PLoS computational biology* 9.11 (2013), e1003258.

[30] Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. “Discovering governing equations from data by sparse identification of nonlinear dynamical systems”. In: *Proceedings of the national academy of sciences* 113.15 (2016), pp. 3932–3937.

A DynaDojo API

DynaDojo is a Python API that lets researchers evaluate how well algorithms generalize on dynamical systems challenges. The platform's purpose is to address the difficulties associated with modeling and predicting dynamical systems by providing a standardized framework for assessment. The API is structured around three core classes: `AbstractSystem`, `AbstractAlgorithm`, and `Challenge`. Their relationship is shown in the UML diagram shown in Figure **??** and described in detail below.

Pipeline For DynaDojo's evaluation pipeline, we first instantiate a `Challenge` and set a `AbstractSystem` subclass for that challenge using the `system_cls` attribute. We then call the challenge's `evaluate` method, which interfaces between the system and a `AbstractAlgorithm` subclass of that we pass to the method. For example, researchers can evaluate a `LinearRegression` algorithm by passing it into the `evaluate` method of an `FixedTrainSize` challenge set with an `LDSSystem`.

The selected algorithm is repeatedly initialized and `fit` on data generated from the system and optionally controlled by algorithm instance itself. With `evaluate`, `Challenge` iteratively adjusts how the data is generated, varying the complexity, size, timesteps, distribution and noise. For testing, `evaluate` compares a test set generated from the system instance with the predicted trajectories from the algorithm instance, returning the results as a pandas DataFrame.

Systems `AbstractSystem` is an abstract base class for dynamical systems. We currently introduce DynaDojo with 20 systems outlined in 2, and described in further detail below. Each system is responsible for generating data associated with that dynamical system. Each system must implement several abstract methods from `AbstractSystem`: `make_init_conds`, `make_data`, `calc_error`, and `calc_control_cost`.

Initial conditions are generated using the `make_init_conds` method. The method accepts arguments specifying the number of initial conditions, the embedded dimension, and whether to sample out-of-distribution initial conditions. Note that in DynaDojo, systems are responsible for recovering their own latent initial coordinates. To illustrate this, consider an instance of `LDSSystem` with `latent_dim` ℓ and `embed_dim` e. The latent dynamics are generated by the equation

$$\dot{x}(t) = Ax(t) + Bu(t)$$

where $A \in \mathbb{R}^{\ell \times \ell}$, $x(t) \in \mathbb{R}^{\ell}$, $B \in \mathbb{R}^{e \times \ell}$, and $u(t) \in \mathbb{R}^{e}$ for all real t. Note that if no control is set, the default u is a zero function. Solutions are mapped into the embedded dimension through a linear transformation $C \in \mathbb{R}^{\ell \times e}$, so initial conditions can be recovered with C^{+} where $\cdot^{+}$ is the Moore–Penrose inverse.

The developer also must determine the distinction between in and out-of-distribution initial conditions. For example, in our `LDSSystem`, in-distribution (IND) initial conditions have positive coordinates and out-of-distribution (OOD) initial conditions have negative coordinates.

`make_data` evolves a collection of initial conditions for a number of timesteps, with optional control inputs and noise; `calc_error` calculates the error between two sets of trajectories; and `calc_control_cost` calculates the control cost of every control input set by a `AbstractAlgorithm` subclass.

Algorithms In DynaDojo, algorithms must subclass from `AbstractAlgorithm` and implement several abstract methods: `fit`, `predict`, and (if the algorithm uses control) `act`. `fit` trains a algorithm on a collection of trajectories in embedding space. `predict` predicts the evolution of a set of initial conditions over a given number of timesteps. The first element of the trajectory should always be the initial condition. `act` allows algorithms to set a control for a given control horizon (described below).

For example, consider `LinearRegression`. It subclasses from `AbstractAlgorithm`. The `fit` method applies the linear regression algorithm to data generated from `LDSSystem` (note that without a Kalman filter, linear regression only works when the latent and embedding dimension are the same). The linear regression predicts some matrix $\hat{A} \in \mathbb{R}^{\ell \times \ell}$ then predicts trajectories from a set of initial conditions passed into `predict` by repeatedly multiplying the latest state conditions in the

trajectories:

$$x_i = \hat{A}x_{i-1}$$

for $i = 1, \ldots, t$ where t is the number of timesteps to predict and x_0 is the initial condition of the trajectory to be predicted.

For completeness, `LinearRegression` also implements an unprincipled `act` that returns a random control $U \sim \text{Uni}[-1, 1]^{n \times t \times e}$ where n is the number of trajectories in the data and e is the embedding dimension. For every action horizon, data is then generated with control (assuming the control constraint is satisfied). More details are in the psuedocode for `evaluate` in Algorithm 1.

Challenges The purpose of the `Challenge` class is to evaluate how certain algorithms perform on dynamical systems challenges with scaling parameters. Unlike `AbstractSystem` and `AbstractAlgorithm`, `Challenge` is not an abstract base class. Developers may want to use `Challenge` directly if they want to implement custom challenge without writing a new child subclass. They may choose to do write `Challenge` subclasses for modularity. For example, DynaDojo provides three base subclasses of `Challenge`: `FixedTrainSize`, `FixedComplexity`, and `FixedError` each corresponding to different evaluation metrics.

The `Challenge` class only has one method: `evaluate` (see Algorithm 1) which takes in an algorithm subclass constructor as well as any additional parameters that are needed for an algorithm's instantiation, fitting, or acting. It also accepts arguments which determine the complexity of the data that algorithm instances are fitted and evaluated on. The main loop in `evaluate` iterates over different sets of training data sizes, timesteps, embedded dimensions, and latent dimensions. When needed, a system is instantiated (using the challenge instance's `system_cls` attribute). The system generates data for the algorithm to be fit on. If the challenge is instantiated with a nonzero number of control horizons, the algorithm is made to generate control tensors in $\mathbb{R}^{n \times t \times e}$ which the system instance uses to extend the training trajectories from their endpoints; however, if the selected control exceeds the control cost per dimension, `evaluate` throws an error and stops early. This happens `control_horizons` times until the algorithm instance is finished training. The system is then used to generate test trajectories. The error between the predicted test trajectories and the true test trajectories is calculated using the `calc_error` method defined by the system.

B Implemented systems

B.1 N-body problem

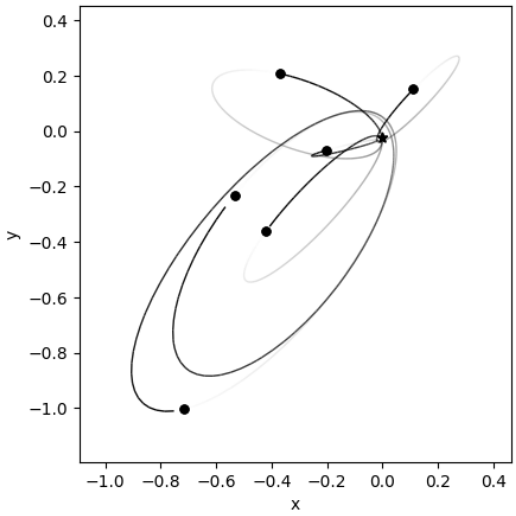


Figure 12: A numerical solution to the 6-body problem. Five bodies orbit a central star in the center.

The N-body problem exudes a captivating allure as it unravels the intricate interplay of celestial bodies moving under the influence of gravitational forces. Beyond its ethereal beauty, this problem

Algorithm 1 Evaluate

Input: System constructor, model constructor
Input: Training sizes N, latent dimensions L, timesteps T
Optional: Embedding dimensions E
Input: Control constraint c, control horizons h
Input: Repetitions r
Input: Test size n' and timesteps t'
Output: Evaluation data $\mathcal{D}$

for each repetition $1, \dots, r$ **do**
  for $n, \ell, t \in N \times L \times T$ **do**
    if E is constant **then**
      $e \leftarrow E$
    else if E is array **then**
      $e \leftarrow E[i]$ where i is ℓ's index in L
    else E is none
      $e \leftarrow \ell$
    end if
    Initialize system s and model f with ℓ, e
    Make training initial conditions $x_0 \leftarrow s(n)$
    Create training data without control $x \leftarrow s(x_0, \varnothing, t)$
    Fit model on x
    Total cost $\bar{c} \leftarrow 0$
    for control horizon $1, \dots, h$ **do**
      Model sets control $u \leftarrow f(x)$
      System calculates control cost $\tilde{c} \leftarrow \|u\|_s$
      Check control meets constraint $\tilde{c} < c$
      $\bar{c} \leftarrow \bar{c} + \tilde{c}$
      Extend training trajectories $x \leftarrow s(x_t, u, t)$
      Fit model on x
    end for
    Make test initial conditions $y_0 \leftarrow s(n')$
    Create test data without control $y \leftarrow s(y_0, \varnothing, t')$
    Model makes predictions $\hat{y} \leftarrow f(y_0)$
    System calculates loss $l \leftarrow \mathcal{L}_s(\hat{y}, y)$
    Update data $\mathcal{D} \leftarrow \mathcal{D} \cup \{(n, \ell, e, t, l, \bar{c})\}$
  end for
end for
Return: $\mathcal{D}$

holds a significant role as a dynamical system of paramount importance, offering insights into the limits of predictability and the emergence of chaos in complex interactions. Its mathematical essence encapsulates the challenge of solving for the trajectories of multiple interacting bodies, encompassing the elegance of abstraction and the ever-growing sophistication of numerical methods and simulations, transcending disciplines and inviting us to navigate the fascinating terrain where simplicity and complexity harmoniously converge.

Consider N point masses m_i for $i = 1, \dots, N$ in an inertial reference frame. Each mass has a position $\mathbf{q}_i$. Letting G be the gravitational constant, the N-body problem can thus be formulated as the following systems of differential equations:

$$m_i \frac{d^2 \mathbf{q}_i}{dt^2} = \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{G m_i m_j \left(\mathbf{q}_j - \mathbf{q}_i\right)}{\left\|\mathbf{q}_j - \mathbf{q}_i\right\|^3} = -\frac{\partial U}{\partial \mathbf{q}_i}$$

where U is the self-potential energy.

B.2 2-dimensional heat equation

The heat equation is a fundamental partial differential equation that governs the distribution of heat over time in various physical systems. It describes how heat diffuses through a material or substance, and its solutions provide insights into temperature changes and thermal behavior. The equation has widespread applications in fields such as physics, engineering, and mathematics. Researchers have extensively studied the heat equation due to its relevance in understanding heat transfer, diffusion processes, and thermal dynamics in a wide range of contexts. From analyzing the cooling of objects to modeling the behavior of fluids and even exploring the behavior of quantum systems, the heat equation remains a cornerstone of scientific investigation and technological innovation.

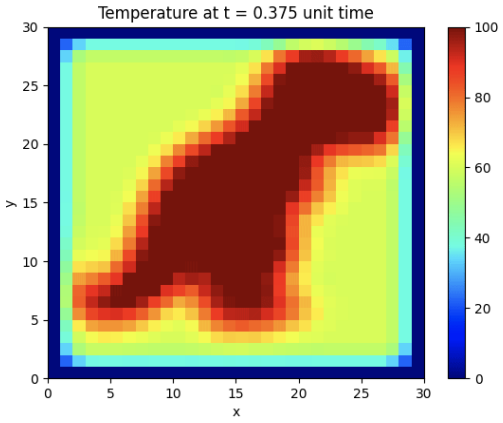


Figure 13: An example of heat diffusing across a 2-dimensional plate. The initial values of the problem were adapted from the DynaDojo logo.

DynaDojo implements a 2-dimensional version of the heat equation. We consider how heat diffuses across a plate. The square plate's width is the square root of the latent dimension. The dynamics of the 2-dimensional heat equation can be expressed

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right),$$

where α is the thermal diffusion and u is the temperature at time t of position (x, y) on the plate.

B.3 Generalized Lorenz System

In its classical formulation, the Lorenz system is a set of three nonlinear differential equations that describe a simplified model of atmospheric convection. Developed by mathematician Edward Lorenz, this system illustrates the concept of deterministic chaos, where small changes in initial conditions can lead to significantly different outcomes over time. The Lorenz system's solutions exhibit a fascinating butterfly-like trajectory in a three-dimensional space known as the Lorenz attractor, highlighting the sensitivity of complex dynamic systems to initial inputs. This system has applications in various fields, including weather prediction, fluid dynamics, and chaos theory.

The Lorenz system can be generalized to n-dimensions using the following formulation [28]:

$$
\begin{aligned}
\frac{dX}{d\tau} &= -\sigma X + \sigma Y \\
\frac{dY}{d\tau} &= -XZ + rX - Y \\
\frac{dZ}{d\tau} &= XY - XY_1 - bZ \\
\frac{dY_j}{d\tau} &= jXZ_{j-1} - (j+1)XZ_j - d_{j-1}Y_j && j \in [1, N] \\
\frac{dZ_j}{d\tau} &= (j+1)XY_j - (j+1)XY_{j+1} - \beta_j Z_j && j \in [1, N] \\
N &= \frac{M-3}{2} \\
d_{j-1} &= \frac{(2j+1)^2 + a^2}{1+a^2} \\
\beta_j &= (j+1)^2 b
\end{aligned}
$$

where M is odd and at least 5, $\tau = \kappa(1+a^2)(\pi/H)^2 t$ and t is time, κ is thermal conductivity, a is the ration of vertical scale of the convection cell to its horizontal scale, H is the domain height, $\sigma = \nu/\kappa$ is the Prandtl number and ν is kinematic viscosity, $r = R_a/R_c$ is the normalized Rayleigh number, R_a is the Rayleigh number, and R_c is its critical value for the free-slip Rayleigh-Bernard problem. For the base conditions, $Z_0 = Z$ and $Y_{N+1} = 0$.

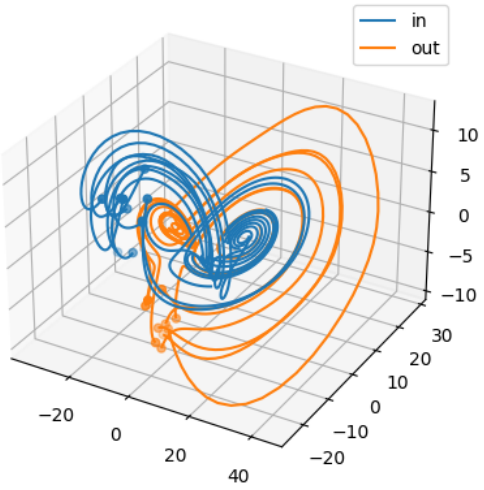


Figure 14: An example a 7-dimensional generalized Lorenz system projected into 3-dimensions using principal component analysis.

B.4 Lotka-Volterra

Lotka-Volterra systems are widely used in ecology to predict how various species will interact with each other, for example lynx and hares, multi-tropic ecosystem, or various bacteria in an environment. Many implementations of Lotka-Volterra in existing literature are focused on a specific number of species (often $n \in \{2, 3, 4\}$). With this specific number of species, the various equations for species interactions are set ahead of time. The Lotka-Volterra systems for DynaDojo generalize for any n number of species, and the equations of the interactions are randomly generated.

We achieve this by creating an A matrix that has all the interactions between species. There is additional nuance and power afforded to the user on DynaDojo in determining if they want a prey-predator system (where species "eat" each other) or competitive (where species all vie for the same

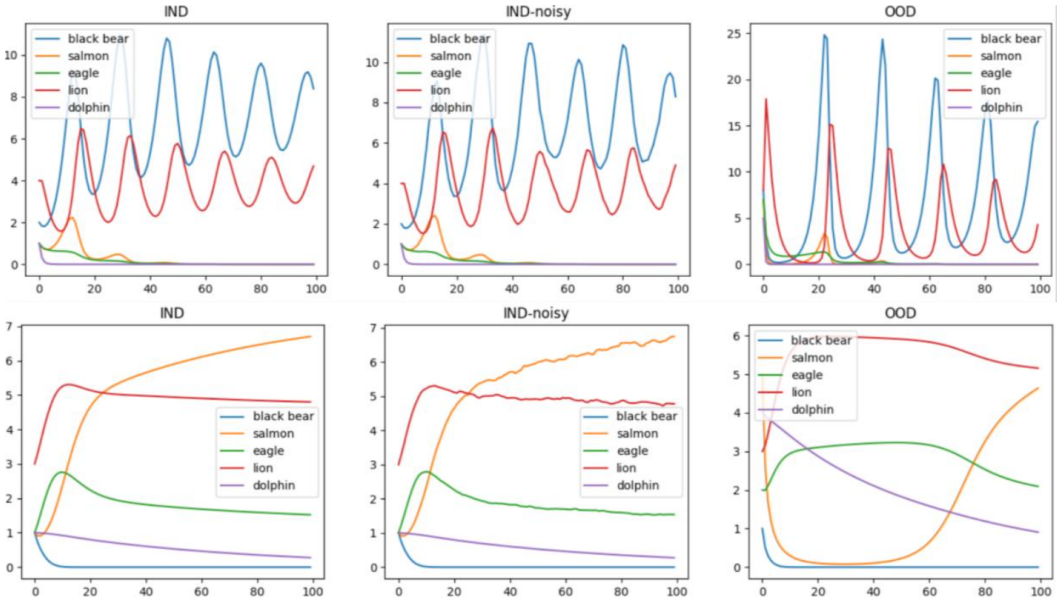


Figure 15: Species in a prey-predator system (top) and competitive system (bottom). We observe the oscillation cycles in prey-predator as well as the species fallout in competitive. For the same underlying dynamics and A matrix, DynaDojo can simulate different trajectories: In-distribution (IND, left), in-distribution with noisy data (IND-noisy, center), and Out-of-distribution (OOD, right). In competitive, OOD leads to a different species being the dominant one (lion), emphasizing the importance of a model that can handle slight deviations in starting conditions in still predicting the underlying dynamics.

resources and only some typically survive) In competitive, each species has a carrying capacity K of the upper population limit that an environment can support for that species. Users can set all of these parameters, including specifically the overall number of species *4* as well as the number of Preys they want. They are specific rules governing the random generation of A, especially in prey-predator to prevent matrix explosion, including that prey intraspecies competition must be positive (without a predator, more prey begets more prey through reproduction), predator intraspecies must be negative (more predators crowd out the predators for resources), and that predators can eat prey while also being eaten by other predators. Specifics of*A* are documented in the codebase.

B.5 Kuramoto

The Kuramoto N-oscillators system simulates multiple weakly-coupled oscillators that synchronize over time. It is an important tool for simulating biochemical processes of objects that work in tandem like pacemaker cells for hearts and fireflies. The System gets more complex as more oscillators are added.

B.6 Epidemic

In epidemic systems, there are a specified number of agents represented by a node graph, a probability of an edge being formed between two nodes, and the probability that infections are passed on and probability that an agent recovers. There are many popular epidemic Systems implemented: SIR (Susceptible/Infected/Removed), SIS (Susceptible/Infected/Susceptible), and SEIS: (Susceptible/Exposed/Infected/Susceptible), all representing different possible states. With DynaDojo it is possible to predict the trajectories of each individual agent or instead the trajectories of each state status as a whole for epidemics:

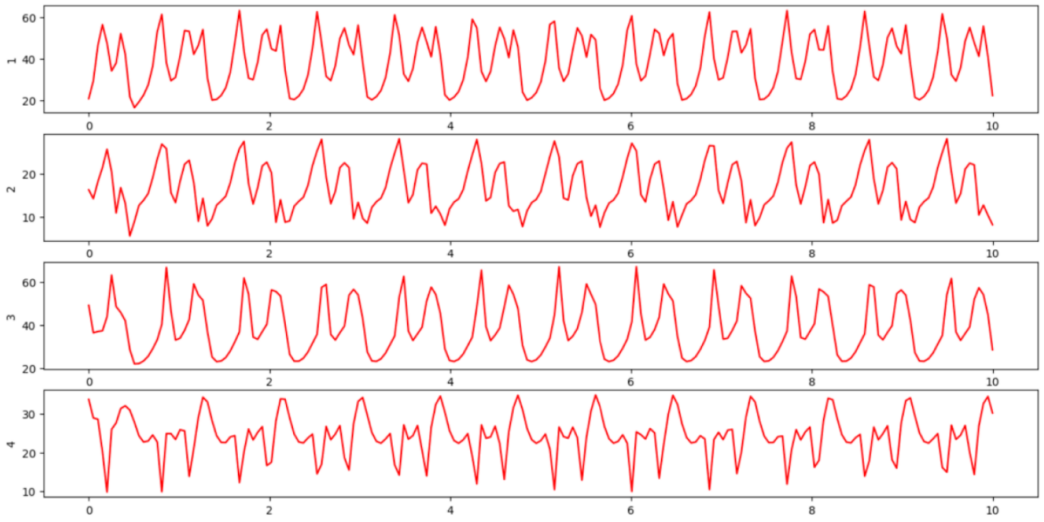


Figure 16: Four oscillators in a Kuramoto system, simulated over 10 timesteps. Note the patterns of peaks and troughs occurring together across the oscillations.

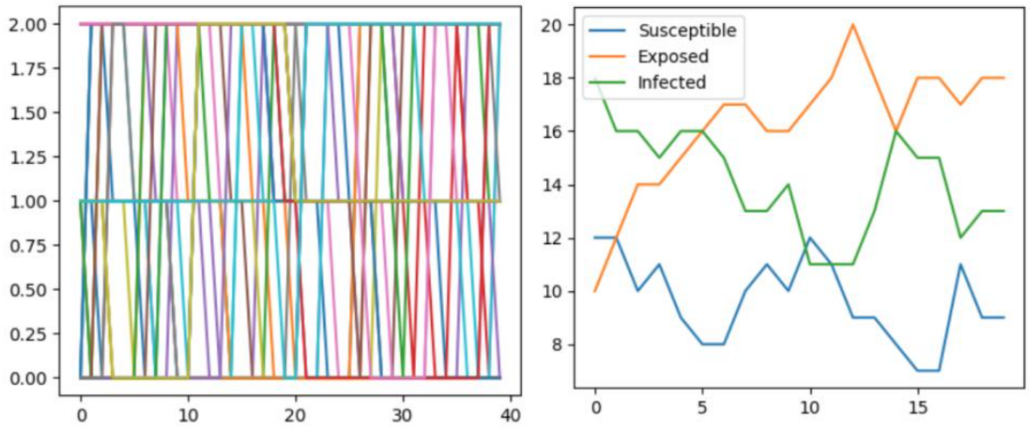


Figure 17: A SEIS System with 40 agents simulated for 40 timesteps, represented as the various states of each agent (left) with Susceptible = 0, Infected = 1, Exposed = 2, and with each state status' overall population (right).

B.7 Bounded confidence opinion

The Bounded-Confidence Opinion Systems are similar to the Epidemic Systems in that they are represented by nodes in a NetworkX graph. However, what is being updated at each time step is an agent's opinion continuously represented from -1 to 1, or 0 to 1. The Deffuant and Media Bias Systems are concerned with pair-wise interactions between two agents — and where if they are similar enough (as defined by a confidence threshold of $epsilon$ between 0 and 1), then the two opinions become their average in the next timestep. The algorithm can be programmed to be more biased in selecting these pairs, creating stronger and stronger echo chambers on more similar pairs, or with media that exerts a strong fixed opinion on others.

The Hegselmann-Krause (HK), Weighted HK (WHK), and Attraction-Repulsion WHK (ARWHK) are additional bounded confidence models. HK deals with an agent being influenced by a group of similar agents (again defined by an $epsilon$ bound) and often shows clustering of opinions. WHK shows quick polarization as only similar pairs of agents interact with each other. ARWHK shows even more complex dynamics as random pairs of agents are chosen and if they are similar they become the average of each other but if unsimilar they grow further apart

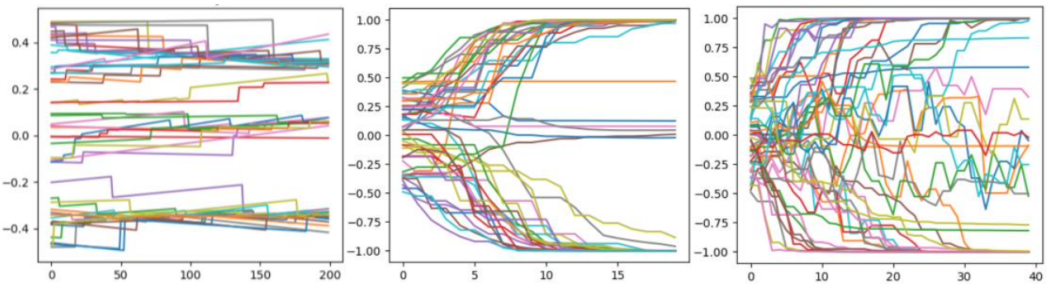


Figure 18: An HK System showing three random clustering groups at $e = 0.2$(left), a WHK system showing rapid polarization at $e = 0.6$ (center) and an ARWHK system (right) showing complex attraction/repulsion dynamics with some polarization at $e = 0.3$.

B.8 Forward-backward stochastic neural networks

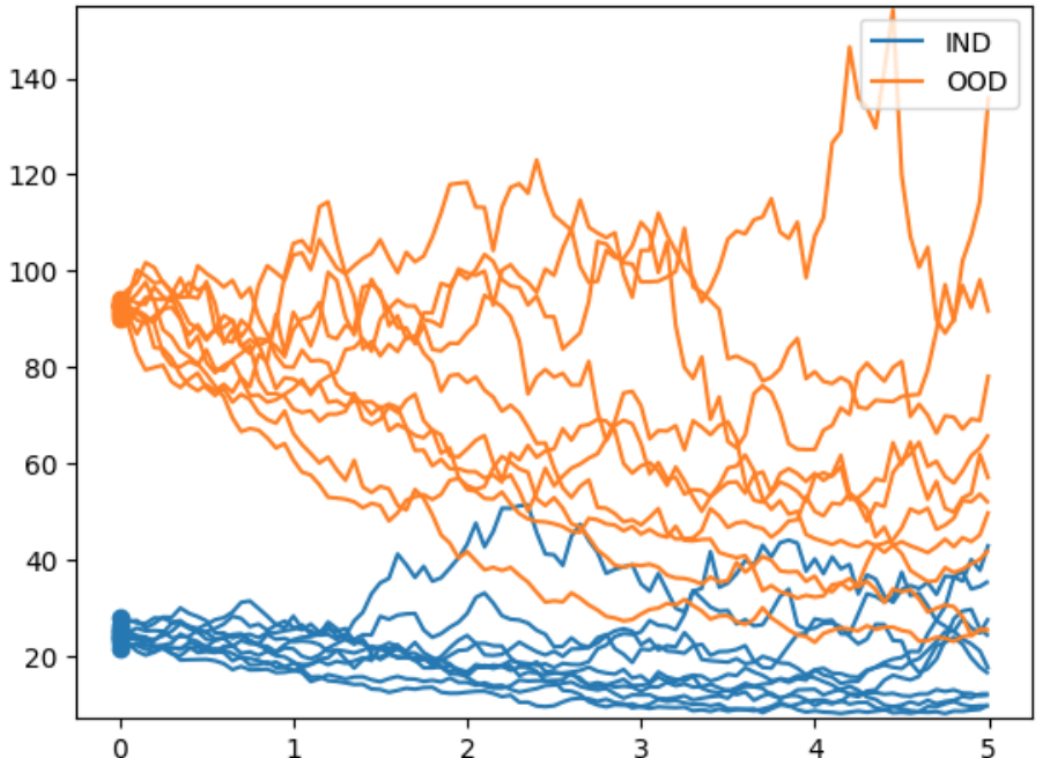


Figure 19: An Black-Scholes-Barenblatt system simulated at complexity/dimension=40, for 100 timesteps between the interval of 0 and 5. The blue trajectories are sampled in-distribution from $(0, 0.8)$ and the orange trajectories are sampled out-of-distribution from $(0.8, 1)$.

To remedy the curse of dimensionality that makes it overly difficult to solve high-dimensional partial differential equations (PDEs), [8] levaraged the relationship between PDEs and forward-backward stochastic differential equations. The solutions to systems like Black-Scholes-Barenblatt (often used in finance to predict asset modeling and call options) and Hamilton-Jacobi-Bellman (often used for optimal control) are able to be obtained at much higher dimensions than typically possible with PDEs (e.g., C=100).

While we sought to find other notions of complexity with the spatiotemporal grid itself in our 2D Heat system, by working without the spatiotemporal grid, we are also able to use dimension as

complexity. This is especially important for models like Black-Scholes-Barenblatt that are often needed in high-dimensions to simulate dozens of finacial assets interacting with each other.

B.9 Cellular automata

Cellular automata (CAs) are discrete dynamical systems that generate complex behaviors from simple rules. They consist of a regular grid of cells, each in one of a finite number of states, typically binary. The state of a cell at the next time step is determined by the current states of a surrounding neighborhood of cells according to a rule. These rules are applied iteratively for as many time steps as desired.

To make cellular automata more similar to other dynamical systems, which often involve control inputs and noise, we introduce modifications to the standard cellular automaton evolution rule. Specifically, we incorporate a control input u_t into the evolution rule and allow for random bit flips with some probability p to introduce noise into the dynamics.

Mathematically, a cellular automaton with control inputs and noise can be defined as follows. Let S be a finite set of states, typically $\{0, 1\}$. The cellular automaton is defined over a d-dimensional lattice, $L = \mathbb{Z}^d$, where each cell $x \in L$ has a state $s(x) \in S$. The neighborhood of a cell is defined as $N(x) = \{y \in L \mid ||y - x|| \leq r\}$, where r is the radius of the neighborhood and $\|\cdot\|$ is a norm on L.

The evolution of the cellular automaton is governed by a local rule $f : S^{N(x)} \to S$, which updates the state of a cell based on the states of the cells in its neighborhood. We modify this rule to include a control input u_t, leading to a new state based on $s_{t+1} = g(f(s_t(N(x))) + u_t)$, where g is the step function centered around 0.5.

In addition, we introduce noise into the dynamics by allowing for random bit flips with some probability p. This can be modeled by a noise function $n : S \to S$ that flips the state of a cell with probability p. The state of the cellular automaton at time $t + 1$ is then given by $s_{t+1}(x) = n(g(f(s_t(N(x))) + u_t))$.

B.10 Linear dynamical systems

Linear dynamical systems (LDS) are a fundamental mathematical framework for modeling and understanding time series data. They are particularly useful for representing systems where the state changes over time as a linear combination of its current state and various input signals.

In the continuous-time setting, an LDS can be described by a first-order differential equation:

$$\dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{u}(t) \tag{1}$$

Here, $\mathbf{x}(t) \in \mathbb{R}^D$ is the state vector at time t, $\mathbf{u}(t) \in \mathbb{R}^{\mathbb{D}}$ is the input vector, $A \in \mathbb{R}^{D \times D}$ is the state transition matrix, and $B \in \mathbb{R}^{D \times D}$ is the input matrix.

One of the key advantages of LDS is their tractability. They are directly solvable, meaning that given the system parameters and initial conditions, we can compute the state of the system at any future time. This makes LDS a powerful tool for simulation and prediction, and numerous tools and techniques have been developed for working with them that we can use to establish baselines for our challenges.

B.11 Threshold linear networks

Threshold linear networks (TLNs) are a class of dynamical systems that have been widely used to model various phenomena, particularly in neuroscience to model the activity of neural networks. They are a natural extension of linear dynamical systems, introducing nonlinearity through a threshold function to better capture the behaviors of real-world systems.

A TLN consists of n units, each associated with a state variable $x_i(t)$ for $i = 1, \ldots, n$, which evolves over time according to the following differential equation:

$$\frac{dx_i}{dt} = -x_i + \left[\sum_{j=1}^{n} W_{ij}x_j + b_i\right]_+, \quad i = 1, \ldots, n, \tag{2}$$

where W_{ij} represents the synaptic weight from unit j to unit i, b_i is an external input to unit i, and $[\cdot]_+ = \max\{\cdot, 0\}$ is the threshold nonlinearity. This is a common form of nonlinearity in models of neural networks, where the firing rate of a neuron (which corresponds to the state variable in the dynamical system) cannot be negative.

TLN dynamics can be represented in matrix form as:

$$\dot{\mathbf{x}}(t) = -\mathbf{x}(t) + [W\mathbf{x}(t) + \mathbf{b}]_+ \tag{3}$$

B.12 Spiking neural networks

Spiking neural networks (SNNs) represent a further step in the progression towards more complex and realistic models of neural systems. They introduce discontinuous dynamics (spikes) into the framework of linear dynamical systems, making them a type of hybrid dynamical system. This section provides an overview of the SNN model, with more a detailed explanation relegated to the Appendix.

The SNN model [29] we include consists of D dynamical variables, denoted as $\mathbf{x}(t)$, a state transition matrix A, time-varying external inputs $\mathbf{c}(t)$, output spike trains $\mathbf{o}(t)$, and a decoding weight matrix $\mathbf{D}$. The model can be described by the following components:

1. The linear dynamical system: $\dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{u}(t)$
2. The estimate of the dynamical variable: $\hat{\mathbf{x}}(t) = \mathbf{D}\mathbf{r}(t)$
3. The time-varying firing rates of the neurons: $\dot{\mathbf{r}}(t) = -\mathbf{r}(t) + \mathbf{o}(t)$
4. The cost function: $E(t) = \int_0^t \left(\frac{1}{2}||\mathbf{x}(u) - \hat{\mathbf{x}}(u)||_2^2\right) du$

The firing rule and membrane potential are governed by the following conditions:

1. $V_i(t) \geq T_i$
2. $V_i(t) = \mathbf{D}_i^T(\mathbf{x}(t) - \hat{\mathbf{x}}(t))$
3. $T_i = \frac{1}{2}||\mathbf{D}_i||_2^2$

The membrane potential and connectivity filters are described by:

1. $\dot{\mathbf{v}}(t) = \mathbf{D}^T A\mathbf{D}\mathbf{v}(t) + \mathbf{V}_s\mathbf{r}(t) - \mathbf{V}_f\mathbf{o}(t) + \mathbf{D}_i^T\mathbf{c}(t) + s_V g(t)$
2. $\mathbf{V}_f = \mathbf{D}^T\mathbf{D}$
3. $\mathbf{V}_s = \mathbf{D}^T(A + \mathbf{I})\mathbf{D}$

The key variables in this model are $\mathbf{x}(t)$ (dynamical variables), A (state transition matrix), $\mathbf{c}(t)$ (external inputs), $\mathbf{o}(t)$ (output spike trains), $\mathbf{D}$ (decoding weight matrix), and $g(t)$ corresponds to a white "background noise" with unit-variance.

C Implemented models

C.1 Sparse identification of nonlinear dynamics

Sparse identification of nonlinear dynamics (SINDy) works by looking for sparse representations of the dynamics of a system in terms of potential functions and their derivatives. It searches for the simplest set of functions that can describe the observed behavior of the system. This approach is particularly useful when dealing with high-dimensional and noisy data, where traditional methods might be computationally intensive or difficult to apply.

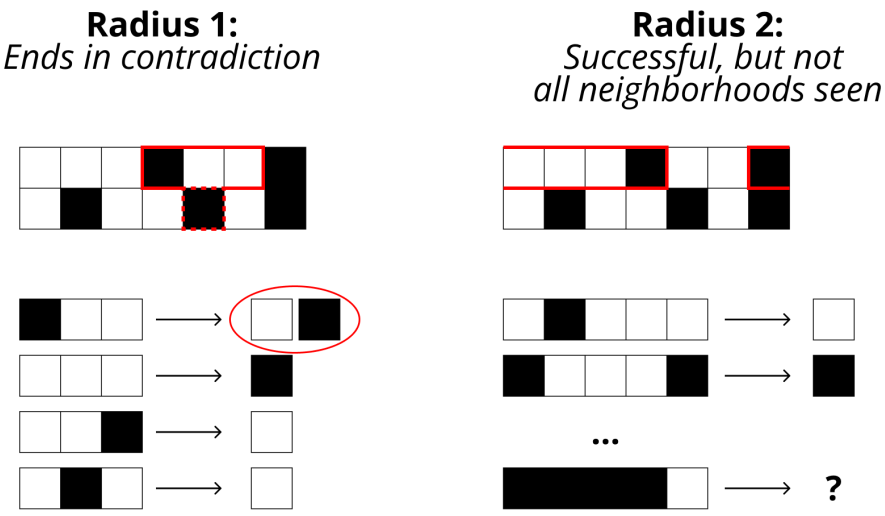


Figure 20: An example of the Lowest Possible Radius model, which sweeps across the training data and finds a contradiction for radius 1, but where radius 2 is successful. Not all neighborhoods are seen for radius 2 though, lowering prediction accuracy.

SINDy has found applications in various fields, including physics, biology, engineering, and economics, where it can help reveal the underlying dynamics of complex systems from limited and noisy data. It's a powerful tool for model discovery and hypothesis generation based on empirical observations.[30]

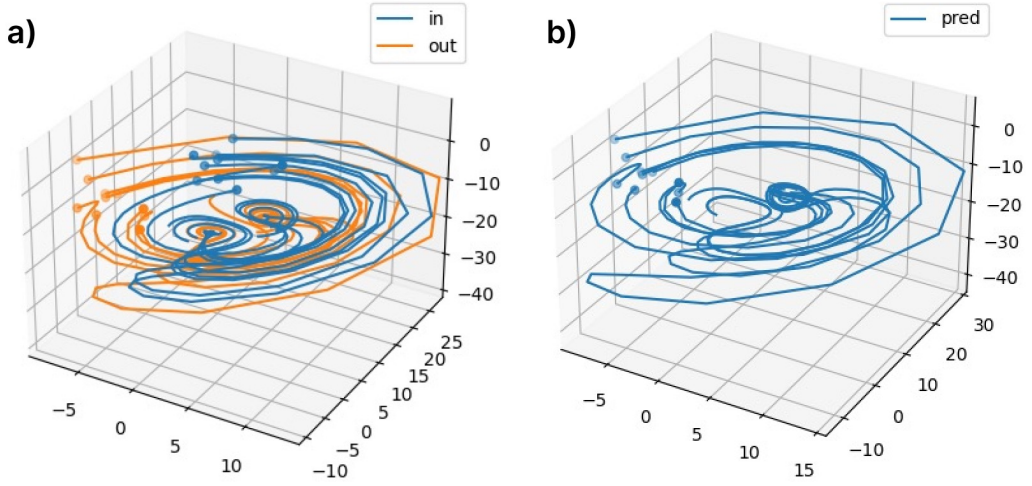


Figure 21: a) The in- and out-of-distribution training trajectories on a 3-dimensional Lorenz system; b) the predicted trajectories using SINDy through DynaDojo.

C.2 Dynamic mode decomposition

Dynamic mode decomposition (DMD) is a powerful data-driven technique that unveils the intricate dynamics of complex systems through observed data. By processing time-series data collected from the system, DMD breaks down the information into snapshot matrices using singular value decomposition. These matrices allow the identification of key modes and patterns within the data, enabling the extraction of coherent structures. DMD then calculates eigenvalues and associated eigenvectors, shedding light on the frequency, growth rates, and behavior of these modes. With these

insights, DMD facilitates the reconstruction of the system's behavior over time, enabling predictions and a deeper understanding of its underlying dynamics.

DMD's applicability spans various domains, from fluid dynamics to neuroscience and beyond. Its ability to analyze high-dimensional and noisy data makes it a valuable tool for uncovering hidden structures and trends within complex datasets. By providing a data-driven approach to deciphering the evolution of systems, DMD contributes to advancements in fields such as physics, engineering, and economics, fostering discoveries and predictions that enhance our comprehension of intricate processes.

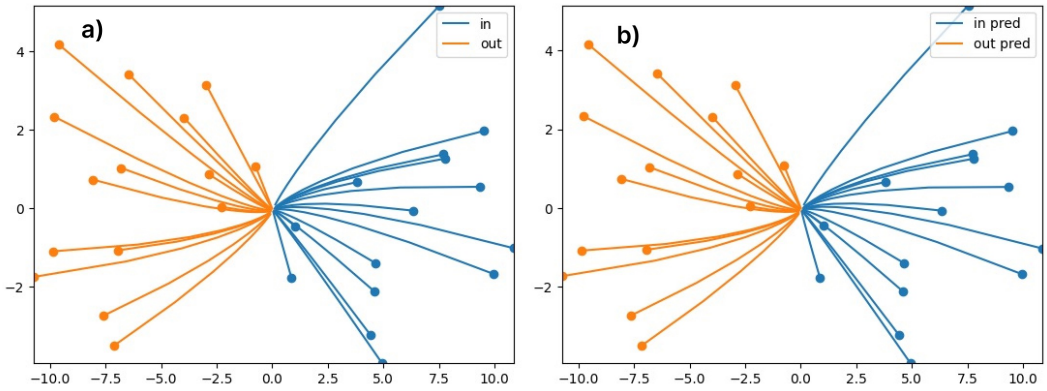


Figure 22: a) The in- and out-of-distribution training trajectories on a 2-dimensional LDS embedded in three dimensions; b) the predicted trajectories using DMD through DynaDojo.

C.3 Lowest possible radius

The lowest possible radius (LPR) model, which works exclusively on the cellular automata (CA) system without any noise, is based on a simple principle of remembering every neighborhood seen in a training set. LPR starts with an assumed radius of **1** (neighborhood size of 3). It uses a sweeping window for each row in each sample, and logs in the `radiiTables dict` the value that each neighborhood maps to. As an example, at radius 1, after sweeping through a small part of the training set, `radiiTables` could look like `{"1":{"100":0, "000":1, "001":0, "010":0, "100":None,...}}` (see Figure 20). When sweeping, if LPR finds an instance of a neighboorhood that maps to a value *different than what it has stored in* `radiiTables` — in Figure 20 the logged value for "100" is `0`, but the model then sees an example of `1` — it will bump up the radius by one, in this case to **2**. This is because in a CA system without noise, the mappings should be deterministic; if "100" maps to both `0` and `1`, this contradiction means "100" cannot be a valid neighborhood and the radius of the system must be larger.

When a radius is bumped, LPR restarts its sweeping from the first row of the first sample, and starts logging the value that each neighborhood of the new radius maps to. For the lowest radius that has no contradictions on the entire sweeped training set, LPR saves that radius to use at prediction. At test time, to predict the evolution of a CA system, LPR uses the `radiiTables` to map each neighborhood to its correct value if it has seen it in training–and equally randomly between `0` and `1` if it has *not* seen it. Thus, more training data is often helpful for LPR as it allows it to see more neighborhoods. Once all neighborhoods are seen though for the actual radius of the system (which can happen easily at lower radii), LPR saturates at an error of **0** and adding more training samples as no effect: All neighborhoods for the correct radius are already seen.

For controlled LPR, for every control horizon, the model will create a `U` control matrix that edits the last observation in the prior horizon. This edit is smart in the sense that it will transform that observation by adding `1`, `-1`, or `0` to change the observation to include only neighborhoods it has not seen. That is, if the last observation of a sample in the previous horizon is `"0,0,1,1,1,0"` and the model has not seen the neighborhoods keys "000" or "101", the `u` for that sample will be `"0,0,-1,0,-1,1"` so that the observation, added with this u, becomes the desired `"0,0,0,1,0,1"` and the model can learn how the unseen keys evolve in the system.

This allows the controlled version of LPR to sometimes require less training samples than the autonomous version of LPR. Notably, controlled LPR outperforms autonomous at samller radii and adequate embedded dimension and control budget, as this provides enough runway to see a substantial number of neighborhoods. In an embedded dimension of 12, the `U` can contain 4 unseen neighborhoods for radius 1 (50% of all possible neighborhoods), but only 1 unseen neighborhood of the 2048 possible neighborhoods for radius 5; lower radii are much amenable to this form of control.

C.4 Deep neural network

The research paper presents a standard Deep neural network (DNN) model. The DNN comprises a stack of seven dense layers. The outer six hidden layers have 30 units and customizable hidden activations. The middle layer has 10 units and a linear activation. For our results, we used linear activations in between our hidden layers with L_2 regularization on the weights, but DNN may perform better on different systems with nonlinear activations. Models were evaluated after training for 30 epochs with the Adam optimizer and mean squared error loss.

C.5 Linear regression

We implement a standard linear regression (LR) baseline using `sklearn`.

For the controlled LR model, it adds a random `u` to each sample. Then, when fitting, it uses knowledge of this `u`.