Discovering Lyapunov functions with transformers

Alberto Alfarano* Meta AI
François Charton* Meta AI
Amaury Hayat* CERMICS - Ecole des Ponts Paristech

Abstract

We consider a long-standing open problem in mathematics: discovering the Lyapunov functions that control the global stability of dynamical systems. We propose a method for generating training data, and train sequence-to-sequence transformers to predict the Lyapunov functions of polynomial and non-polynomial systems with high accuracy. We also introduce a new baseline for this problem, and show that our models achieve state-of-the-art results, and outperform approximation based techniques and sum-of-square algorithmic routines.

1 Introduction

Recent applications of AI, and especially Transformers [34], to mathematics mostly focus on two directions. One aims to improve the ability of Large Language Models in reasoning, performing arithmetic, or solving elementary problems written in natural or formal language [22, 23, 15, 26, 37, 41, 21, 38, 16, 28, 36, 39, 13]. The other one studies the capability of transformers to solve undergraduate or graduate level problems [33, 20, 6, 7, 8, 5] that play an important role in many fields of science, but that can already be solved via non-AI means. There were comparatively few attempts to use AI to solve open research problems in mathematics [35, 10], and most of them do not use transformers.

We believe that two factors account for the lack of application of transformers to research problems. First, the formalization of the underlying problem may require specialized work by mathematicians [3]. Second, math transformers are typically trained on large supervised sets of problems and solutions, generated using an external solver to compute the solutions of random problems. For open problems, such an approach is unfeasible, and the only way to create supervised training data is to sample solutions and generate problems associated to them (the "backward method" from [20]). This creates new difficulties [40]. First, we no longer control the distribution of the problems we train the model to solve – i.e. we might be solving a specific, and easier, version of the problem we claim to be working on. Second, the technique for generating problems from solutions must be chosen with care, or the model might just learn to 'ìnverse" our naive data generator, instead of solving the actual problem.

In this paper, we focus on a long-standing, yet easy to formalize, open problem in mathematics: discovering the Lyapunov functions that control the global stability of dynamical systems – whether their solutions always remain bounded when time goes to infinity. We propose a new technique for generating training data for supervised learning, and show that transformers trained on this dataset achieve very high accuracies on held-out test sets. We also show that our models outperform deep-learning based approximation techniques and classical algorithmic routines that provide partial solutions to this problem in specific cases. Finally, we introduce an out-of-distribution benchmark of hard (from a human perspective) but algorithmically solvable cases, and show that our models achieve good performance, and compute several hundred times faster than state-of-the-art methods.

We believe our work can be useful to both mathematicians and AI researchers. Our models can be used to discover new stable dynamical systems, and our approach may serve as a blueprint for future work on open problems in mathematics.

*Equal contribution

37th Conference on Neural Information Processing Systems (NeurIPS 2023) Workshop on MATH-AI.

2 Lyapunov functions

The stability of a dynamical system, i.e. whether its solutions remain bounded when time goes to infinity, is a problem that intrigued mathematicians for centuries. It was studied by Newton, Lagrange in the 18th century, and then Poincaré at the beginning of the 20th century, in the context of the *three body problem*. A turning point happened at the turn of the 20th century, when Lyapunov showed, among other things, that a system is stable if one could find for this system a kind of decreasing entropy function, the *Lyapunov function* [18, 9, 24]. Existence of a Lyapunov function was later shown to a necessary condition for stability for many classes of system [29, 25, 17]. Lyapunov's theorem is a very strong result, but unfortunately, it provides no clue about finding such a Lyapunov functions, or even showing that one exists. Today, more than a century later, systematic ways of finding Lyapunov functions are only known in a few special cases, and their derivation remains an open problem in mathematics.

The mathematical problem can be presented this way: consider the dynamical system

$$\dot{x} = f(x), \tag{2.1}$$

where $x \in \mathbb{R}^n$ and $f \in C^1(\mathbb{R}^n)$ is the dynamics of x and satisfies $f(0) = 0$. Here $C^1(\mathbb{R}^n)$ refers to the functions that are continuously differentiable on $\mathbb{R}^n$ and $\dot{x}$ refers to the derivative with time of x. The goal is to know if the system has a stable equilibrium x^* that can be taken without loss of generality as $x^* = 0$. Mathematically speaking, this means,

Definition 2.1. The system (2.1) is said *stable* when, for any $\varepsilon > 0$, there exists $\eta > 0$ such that, if $\|x(0)\| < \eta$, the system (2.1) with initial condition $x(0)$ has a unique solution $x \in C^1([0, +\infty))$ and

$$\|x(t)\| \leq \varepsilon, \quad \forall\, t \in [0, +\infty). \tag{2.2}$$

In short, a system is stable if we can guarantee that the solution remains bounded and as small as we like (this is represented by ε), provided that the initial condition is sufficiently small (this is represented by η). What Lyapunov showed is that this is linked to the notion of Lyapunov functions

Definition 2.2. The function $V \in C^1(\mathbb{R}^n, \mathbb{R}_+)$ is said to be a *Lyapunov function* if the following condition are satisfied

$$V(0) = 0, \quad V(x) > 0 \ \text{ for } x \neq 0, \quad \nabla V(x) \cdot f(x) \leq 0. \tag{2.3}$$

The two first conditions can be replaced without loss of generality by $V(x) > V(0)$ for any $x \neq 0$. Lyapunov's main theorem is

Theorem 2.1 (Lyapunov 1892)**.** *If there exists a Lyapunov function to the system* (2.1)*, then this system is stable.*

Most dynamical systems are unstable, for instance the solutions of the simple system $\dot{x}(t) = x(t)$ grow exponentially with time. The solutions, for $x \in \mathbb{R}$, of $\dot{x}(t) = 1 + x(t)^2$, always blow up before $t = \pi$. No Lyapunov functions can be found for these systems. On the other hand, a stable system can have an infinite number of Lyapunov functions. For the system $\dot{x}_0(t) = -x_0(t)$ and $\dot{x}_1(t) = -x_1(t)$, $V = a_0 x_0^2 + a_1 x_1^2$ is a valid Lyapunov function for any choice of $a_0 > 0, a_1 > 0$.

There is no systematic way of finding a Lyapunov function. Mathematical approaches typically use empircal techniques such as backstepping or forwarding ([9, Chap. 12]) on simple parameterized candidates, to derive sufficient conditions on the parameters [12]. Computational techniques, such as those implemented in SOSTOOLS [31, 32] can be used in special cases, like polynomial systems of small degrees. See Appendix A for a discussion of other approaches (including neural networks).

3 Generating datasets

Creating a reliable dataset of problems and solutions that can be used to train transformer to discover Lyapunov functions is one of the main challenges of this work. Since there is no systematic method in general for finding a Lyapunov function for a given system, or even testing whether one exists, we cannot create training datasets by randomly sampling instances of the problem and calculating their solutions, as in previous works on problems with known solutions [8]. Instead, we introduce a new technique, sampling a function V, and then finding a system that has V as a Lyapunov function. The procedure is detailed in Appendix C, and can be summarized as follows.

(step 1) Generate a function V at random, satisfying $V(x) > V(0), \ \ \forall x \in \mathbb{R}^n \setminus \{0\}$.
(step 2) Compute the gradient $\nabla V(x)$ and denote $\mathcal{H}_x = \{z \in \mathbb{R}^n \mid z \cdot \nabla V(x) = 0\}$ the hyperplane[2] orthogonal to $\nabla V(x)$, for any $x \in \mathbb{R}^n$.
(step 3) Select $p \in \{1, ..., n\}$ at random and generate p vectors $\{e^i\}_{i \in \{1,...,p\}}$ of this hyperplane.
(step 4) Generate at random $k_1 > 1$ real-valued functions $(f_i)_{i \in \{1,...,k_1\}}$ and $k_2 \geq 0$ real-valued functions $(g_i)_{i \in \{1,...,k_2\}}$ (k_1 and k_2 chosen at random) and denote $f_i = 0$ for $i \geq k_1 + 1$.
(step 5) Build the final system $f(x) = -(f^2_{\tau_3(i)}(x)(\nabla V(x))_i)_{i \in \{1,...,n\}} + \sum_{i=1}^{k_2} g_i(x) e^{\tau(i)}(x)$, with τ_3 a random permutation of $\{1, ..., n\}$ and τ a random function from $\mathbb{N}$ to $\{1, ..., n\}$.
(step 6) Simplify the final system, therefore masking obvious patterns from the generative process.

By selecting the functions V, f_i and g_i from particular classes, we can constrain this technique to generate particular systems. We will use this technique to generate datasets of polynomial and non-polynomial systems.

A key consideration when generating training data using backward techniques (i.e. generating problems from solutions) is that the process should not be easily inverted. For example, when training a model to predict the roots of polynomial, if the problem is presented in factorized form, e.g. $2(X-3)(X-5)(X-7)$, the language model can learn to "read" the solutions directly in the problem, a much easier task than learning to solve from the developed polynomial $2X^3 - 30X^2 + 142X - 210$.

In the case of Lyapunov functions, a key step in preventing V from being easily inferred from the expression of f, is the choice of the vectors e^i. If we naively select for our e^i an orthonormal basis of $\mathcal{H}_x$, calculated from $\nabla V(x)$ (e.g. using Gram-Schmidt orthogonalization), coefficients like $1/|\nabla V(x)|$ will appear in the expression at step 3, with close to no chance of being simplified at step 5, allowing the transformer to read V from the system, without learning any mathematics. On the other hand, if the e^i are simple enough to allow the $1/|\nabla V(x)|$ terms to simplify away, but not diverse enough to span $\mathcal{H}_x$, the systems generated will be less diverse, and models trained on the data set will fail to generalize out of the training distribution, for lack of diversity in their training data.

Baselines for the polynomial case. In the particular case of polynomial systems, there are routine methods to find polynomial sum-of-square Lyapunov functions with high accuracy, when such functions exist (see Section A). We can use these techniques to generate a forward dataset, as follows:

(step 1) Generate a polynomial system at random
(step 2) Use a routine to find a potential polynomial and sum-of-square Lyapunov function.
(step 3) Keep the system if such function exists, restart from step 1 otherwise.

Because most random polynomial systems are not globally stable, this method is computationally costly, and could not be used to generate training data.

Since a globally stable polynomial system does not necessarily have a polynomial sum-of-square Lyapunov function, this dataset does not represent all globally stable polynomial systems. Yet, the polynomial systems in this dataset have, *a priori*, a very different distribution from those generated using the backward method described previously. This makes it a good evaluation benchmark.

4 Models and experiments

Models. We frame all problems as translation tasks [20, 6], and train sequence-to-sequence transformers [34] to translate systems –represented as a sequence of mathematical symbols– into Lyapunov functions, also represented as sequences, by minimizing the cross-entropy between model predictions and correct solutions. We train models with 6 to 10 layers, 8 to 16 attention heads and embedding dimension between 512 and 1024 (see Appendix B).

Datasets Models are trained on three datasets, with around 30,000,000 of different globally stable systems each, generated using the "backward" technique from Section 3 (see Appendix D for details):

- small systems of polynomials, 2-3 equations, Lyapunov functions of degree up to 8
- large systems of polynomials, 3-6 equations, Lyapunov functions of degree up to 12
- non polynomial systems, 2-3 equations

[2] if $\nabla V(x) = 0$ this is the whole space instead, but this does not change the method.

For evaluation, we introduce the "forward" dataset: polynomial systems in 2 or 3 variables, generated using the method described at the end of Section 3. $3,715$ test examples were generated, running SOSTOOLS for up to $2,000$ seconds.

Model evaluation. For any stable system, an infinite number of Lyapunov functions exist. At evaluation, we check that the model predicts a valid sequence, representing a function V that V satisfies equation 2.3. Model predictions use beam search with early stopping, normalizing log-likelihood scores by their sequence length. We report results with for beam 1 (greedy decoding) and 50, and consider that the model is correct when one correct Lyapunov function is found.

Main results Our models can find a correct Lyapunov function in more than 95% of examples in held-out test sets (Table 1). On these test sets, we compare our results with three prior works: SOSTOOLS [31, 32], Fossil [1], Augmented NL [14]. Neither Fossil nor Augmented NL can predict any correct solutions. This could be explained by the fact that Fossil and Augmented NL are solving a harder problem: they are designed to find a stabilizing control and a Lyapunov function on the system, whereas we only focus on finding a Lyapunov function. SOSTOOLS achieves 78% accuracy on the small polynomial test set, but its performance drops when systems become larger, and it cannot address non-polynomial systems.

On the forward test set (Table 2), our best model, trained on the non-polynomial dataset, achieves 83.1% accuracy with beam search. This demonstrates that models trained on "backward" distributions (i.e. problems generated from their solutions) can generalize out of their training distribution. On the forward test set, Fossil and Augmented NL achieve respectively 0 and 3% accuracy. SOSTOOLS achieve 100%, this is by design: the dataset was constructed from systems that SOSTOOL can solve.

An interesting feature of our results on the forward test set is that the best model was trained on the non-polynomial dataset (Table 2). Since all test examples are small polynomials, one would expect models trained on polynomials systems to perform better. This suggests that training from more diverse datasets, and harder instances, helps improve out-of-distribution generalization, even on easy and specific problems. Additional results can be found in Appendix E.

	Previous work			Our work	
Test set	Fossil	Augmented NL	SOSTOOLS	Beam 1	Beam 50
Small polynomial	0%	0%	78%	**99.2%**	**99.3%**
Large polynomial	0%	0%	16%	**93.8%**	**95.1%**
Non-polynomial	0%	0%	N/A	**97.1%**	**97.8%**

Table 1: **Model accuracy on backward (held-out) test sets**. For our models, training and test distributions are the same.

Training distribution	Beam 1	Beam 50
Small polynomial	42.4%	77.1%
Large polynomial	35.8%	63.1%
Non-polynomial	**38.6%**	**83.1%**

Table 2: **Model accuracy on the forward test set**. 8-layer models.

Finally, we note that on transformer-based models, Lyapunov function prediction is much faster than in alternative models. To generate examples in the forward dataset, we needed to run SOSTOOLS with a timeout of 2000s. Average evaluation time across all datasets was 2,471s for Fossil and 1,394s for Augmented NL. On average, inference tine with our models is 0.4s per example with greedy decoding, and 8.9s with a beam search of 50.

5 Discussion

These experiments demonstrate how an open mathematical problem, like discovering Lyapunov functions, can be solved by transformers using supervised learning, so long an external method for verifying model predictions is available. We show how training sets can be generated "backwards" (problems from solutions). Yet, our results on the forward test set indicate that models trained from backward sets can generalize to "regular" systems, and predict correct solutions with high accuracy. This suggests that our models are learning the underlying mathematics, and not the generating procedure. (In other words, it does not learn to invert the backward procedure we use for generating our training data since examples in the forward dataset are not generated from solutions). To our knowledge, this is the first attempt to recover a symbolic Lyapunov function in the general case without resorting to numerical methods. It also proves to be much faster than previous computational techniques.

From a mathematical point of view, this article proposes a new way of finding Lyapunov functions, in a much larger framework than was possible so far, using current mathematical theories. While this

systematic procedure remains a black box, its solutions are explicit and can be checked to ensure their mathematical validity. This shows that language models can be used to solve problems from research level mathematics. Although only a small minority of mathematicians currently utilize deep-learning tools, it suggests that AI can help mathematicians make tremendous progress and may become a central component in the future landscape of mathematical practice.

References

[1] Alessandro Abate, Daniele Ahmed, Alec Edwards, Mirco Giacobbe, and Andrea Peruffo. FOSSIL: A Software Tool for the Formal Synthesis of Lyapunov Functions and Barrier Certificates using Neural Networks. In *Proceedings of the 24th International Conference on Hybrid Systems: Computation and Control*, HSCC '21, page 11. Association for Computing Machinery, May 2021.

[2] Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. Neural symbolic regression that scales, 2021.

[3] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, jan 2020.

[4] Ya-Chien Chang, Nima Roohi, and Sicun Gao. Neural Lyapunov Control. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[5] François Charton. Can transformers learn the greatest common divisor? *arXiv preprint arXiv:2308.15594*, 2023.

[6] François Charton, Amaury Hayat, and Guillaume Lample. Learning advanced mathematical computations from examples. *arXiv preprint arXiv:2006.06462*, 2020.

[7] François Charton, Amaury Hayat, Sean T McQuade, Nathaniel J Merrill, and Benedetto Piccoli. A deep language model to predict metabolic network equilibria. *arXiv preprint arXiv:2112.03588*, 2021.

[8] François Charton. Linear algebra with transformers, 2022.

[9] Jean-Michel Coron. *Control and nonlinearity*. American Mathematical Soc., 2007.

[10] A Davies, P Velickovic, L Buesing, S Blackwell, D Zheng, N Tomasev, R Tanburn, P Battaglia, C Blundell, A Juhasz, et al. Advancing mathematics by guiding human intuition with AI. *Nature*, 2021.

[11] Peter Giesl. *Construction of global Lyapunov functions using radial basis functions*, volume 1904. Springer, 2007.

[12] Peter Giesl and Sigurdur Hafstein. Review on computational methods for lyapunov functions. *Discrete and Continuous Dynamical Systems-B*, 20(8):2291–2331, 2015.

[13] Fabian Gloeckle, Baptiste Roziere, Amaury Hayat, and Gabriel Synnaeve. Temperature-scaled large language models for Lean proofstep prediction. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS'23*, 2023.

[14] Davide Grande, Andrea Peruffo, Enrico Anderlini, and Georgios Salavasidis. Augmented Neural Lyapunov Control. *IEEE Access*, 11:67979–67986, 2023.

[15] Kaden Griffith and Jugal Kalita. Solving arithmetic word problems with transformers and preprocessing of problem text. *arXiv preprint arXiv:2106.00893*, 2021.

[16] Albert Qiaochu Jiang, Sean Welleck, Jin Peng Zhou, Timothee Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *11th International Conference on Learning Representations*, 2022.

[17] Christopher M Kellett. Classical converse theorems in lyapunov's second method. *Discrete & Continuous Dynamical Systems-Series B*, 20(8), 2015.

[18] Hassan K. Khalil. *Nonlinear systems*. Macmillan Publishing Company, New York, 1992.

[19] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*, 2019.

[21] Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. HyperTree Proof Search for Neural Theorem Proving. *Advances in neural information processing systems*, 2022.

[22] Nayoung Lee, Kartik Sreenivasan, Jason D. Lee, Kangwook Lee, and Dimitris Papailiopoulos. Teaching arithmetic to small transformers. *arXiv preprint arXiv:2307.03381*, 2023.

[23] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, Yuhuai Wu, Behnam Neyshabur, Guy Gur-Ari, and Vedant Misra. Solving quantitative reasoning problems with language models. *arXiv preprint arXiv2206.14858*, 2022.

[24] Aleksandr Mikhaĭlovich Lyapunov. *The general problem of the stability of motion*. PhD thesis, Kharkov University, 1892.

[25] Jose Luis Massera. On liapounoff's conditions of stability. *Annals of Mathematics*, pages 705–721, 1949.

[26] Yuanliang Meng and Anna Rumshisky. Solving math word problems with double-decoder transformer. *arXiv preprint arXiv:1908.10924*, 2019.

[27] Rodrigo Frassetto Nogueira, Zhiying Jiang, and Jimmy Lin. Investigating the limitations of the transformers with simple arithmetic tasks. *CoRR*, abs/2102.13019, 2021.

[28] Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

[29] KP Persidskii. On a theorem of liapunov. In *CR (Dokl.) Acad. Sci. URSS*, volume 14, pages 541–543, 1937.

[30] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving, 2020.

[31] Stephen Prajna, Antonis Papachristodoulou, and Pablo A Parrilo. Introducing sostools: A general purpose sum of squares programming solver. In *Proceedings of the 41st IEEE Conference on Decision and Control, 2002.*, volume 1, pages 741–746. IEEE, 2002.

[32] Stephen Prajna, Antonis Papachristodoulou, Peter Seiler, and Pablo A Parrilo. Sostools and its control applications. *Positive polynomials in control*, pages 273–292, 2005.

[33] Feng Shi, Chonghan Lee, Mohammad Khairul Bashar, Nikhil Shukla, Song-Chun Zhu, and Vijaykrishnan Narayanan. Transformer-based Machine Learning for Fast SAT Solvers and Logic Synthesis. *arXiv preprint arXiv:2107.07116*, 2021.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[35] Adam Zsolt Wagner. Constructions in combinatorics via neural networks. *arXiv preprint arXiv:2104.14516*, 2021.

[36] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[37] Yuhuai Wu, Albert Qiaochu Jiang, Jimmy Ba, and Roger Grosse. Int: An inequality benchmark for evaluating generalization in theorem proving. *arXiv preprint arXiv:2007.02924*, 2020.

[38] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35:32353–32368, 2022.

[39] Kaiyu Yang, Aidan M Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *arXiv preprint arXiv:2306.15626*, 2023.

[40] Gal Yehuda, Moshe Gabel, and Assaf Schuster. It's not what machines can learn, it's what we cannot teach. *arXiv preprint arXiv:2002.09398*, 2020.

[41] Hattie Zhou, Azade Nova, Hugo Larochelle, Aaron Courville, Behnam Neyshabur, and Hanie Sedghi. Teaching algorithmic reasoning via in-context learning. *arXiv preprint arXiv:2211.09066*, 2022.

Appendix

A Related works

The mathematical approaches to finding Lyapunov functions mainly consist in trying simple parametrized Lyapunov function candidates and to find conditions on the parameters [9, 11]. Several techniques such as backstepping or forwarding (see [9, Chap. 12] for more details) were developed to exploit the structure of the system is particular cases. These approaches used to be limited to a few special, or very simple, cases. However, with the increase in computing resources, more complex parameterized Lyapunov functions could be tackled. In particular, tools were developed for polynomial systems with a sum-of-squares polynomial function of a fixed degree, which belong to a finite-dimensional space (whereas generic C^1 and C^∞ functions belong to an infinite dimensional space). One well-known example of such tools is SOSTOOLS [31, 32]. This approach, however, is not applicable to more general settings.

Several neural network approaches have been proposed in recent years [4, 14, 1]. They learn Lyapunov functions using a vanilla 2-layer feed forward network. Models, trained on a dataset of state-space samples, generate candidate Lyapunov function, while a Satisfiability Modulo Theories (SMT) solver acts as a falsifier of the predicted Lyapunov function, by providing counter-examples. This approach demonstrated its potential by finding correct Lyapunov functions for some well-studied high dimensional systems. It has not been extensively studied on larger classes of systems.

Transformers [34] have been trained over synthetic datasets to solve various mathematical problems: arithmetic [27], linear algebra [8], symbolic integration [20], symbolic regression [2] and theorem proving [30]. The application closest to ours is [6], which focuses on the local stability of dynamical systems, i.e. applications of the spectral mapping theorem.

B Model settings

In all experiments, we train sequence-to-sequence transformers [34] with 6 to 10 layers, 8 to 16 attention heads, and embedding dimensions from 512 to 1024. During training, we minimize the cross-entropy between model predictions and correct solutions. The optimizer is Adam [19], with a learning rate of 10^{-4}. We use linear warmup over the 10000 first optimation steps, and inverse square root scheduling. Models are trained on batches of 16 examples, over 4 V100 GPU with 32 GB of memory.

After every epoch (300,000 examples), we evaluate the in-domain performance of our models on a held-out test set of 10,000 examples, created with the same generator as the training set, with greedy decoding (i.e. beam search of 1). We ensure that the training and test set do not overlap. As in previous works [20], we note that model predictions are almost always syntactically correct: predicted sequences that cannot be decoded as a function only account for 0.1% of the test set.

C Generation procedure

C.1 Function generation

Following [20, 6], we represent functions as trees. We generate random functions by sampling random trees with unary and binary internal nodes, and then randomly selecting operators for these nodes, and variables and integers for leaves. Our binary operators are the four operations and the power function. Unary operators are $\exp, \log, \mathrm{sqrt}, \sin, \cos, \tan$.

To generate polynomials, we randomly sample a given number of monomials, with integer or real coefficients. The number of monomials, range of the coefficients, and the powers and number of terms of each monomial, are randomly selected between bounds, provided as hyperparameters.

C.2 Backward generation

We build globally stable systems by first generating a Lyapunov function V at random, and then building a dynamic system which has V as a Lyapunov function. The procedure is:

Step 1: Generate two random functions V_{cross} and V_{proper}

$$V_{\text{cross}}(x) = \sum_{i=1}^{m} (P_i(x))^2, \tag{C.1}$$

with m a random integer, and P_i random functions verifying $P_i(x) = 0$. The nature of functions P_i depends on the systems we want to generate (polynomial or not).

$$V_{\text{proper}}(x) = \left(\sum_{i=1}^{n} \alpha_{i,j} x_i^{\beta_i} x_j^{\beta_j} \right), \tag{C.2}$$

with n a random integer, β_i random positive integers and $A = (\alpha_{i,j})_{(i,j)\in\{1,\dots,n\}^2}$ a random positive definite matrix. Note that this guarantees that V_{cross} and V_{proper} are minimal in $x = 0$.

Step 2: Transform, by multiplication and composition, the components V_{proper} and V_{cross}:

1. with probability $p_{1,c}$, replace
$$V_{\text{proper}}(x) \leftarrow f(V_{\text{proper}}(x)) \tag{C.3}$$
with f a random increasing function chosen from a pre-defined set of *positive-functions*
2. with probability $p_{1,m}$, replace
$$V_{\text{proper}}(x) \leftarrow (V_{\text{proper}}(x) - V_{\text{proper}}(0))g(h(x)), \tag{C.4}$$
with g a random positive function from a pre-defined set of *positive-functions* and h a sub-expression of V_{proper}.
3. for every $k \in \{1, ..., m\}$, with probability p_2, replace
$$P_k(x) \leftarrow f_k(x_k + P_k(x)), \tag{C.5}$$
where f_k is a function that is bounded by below with a minimum (not necessarily unique) in x_k and chosen at random from a pre-defined set of *bounded-functions*.

Step 3: Define the Lyapunov function $V(x) = V_{\text{cross}}(x) + V_{\text{proper}}(x)$. Overall, we have

$$V(x) = \left[f\left(\sum_{i=1}^{n} \alpha_{i,j} x_i^{\beta_i} x_j^{\beta_j} \right) - f(0) \right] g\left(\sum_{i=1}^{p} \alpha_{\sigma(i),\sigma(j)} x_{\sigma(i)}^{\beta_{\sigma(i)}} x_{\sigma(j)}^{\beta_{\sigma(j)}} \right) + \sum_{i=1}^{m} f_k(x_k + P_k(x)), \tag{C.6}$$

where $f = Id$ with probability $1 - p_{1,c}$, $g = 1$ with probability $1 - p_{1,m}$ and $f_k(x) = x^2$ with probability $1 - p_2$ and σ is a random permutation and $p \in \{1, ..., n\}$.

Such a Lyapunov function satisfies

$$V(x) > V(0), \quad \forall x \in \mathbb{R}^n \setminus \{0\}. \tag{C.7}$$

Indeed,

$$V(0) = \sum_{i=1}^{m} f_k(x_k)$$

and $V(x) > f_k(x_k)$ for any $x \in \mathbb{R}^n \setminus \{0\}$, since g is positive, f is increasing and $(\alpha_{i,j})_{i,j\in\{1,...,n\}}$ is positive definite.

Step 4: Taking advantage of (2.3), for any $x \in \mathbb{R}$, denote

$$\mathcal{H}_x = \{z \in \mathbb{R}^n \mid z \cdot \nabla V(x) = 0\}$$

the hyperplane orthogonal to $\nabla V(x)$.

Then, for a random $p \in \{1, ..., n\}$, generate p vectors $\{e^i\}_{i\in\{1,...,p\}}$ from this hyperplane as follows:

$$e^i_j = \begin{cases} (\nabla V(x))_{\tau_2(i)} \text{ if } j = \tau_1(i) \\ -(\nabla V(x))_{\tau_1(i)} \text{ if } j = \tau_2(i) \\ 0 \quad \text{otherwise,} \end{cases} \tag{C.8}$$

with τ_1 and τ_2 random functions from $\mathbb{N} \setminus \{0\}$ into $\{1, ..., n\}$, such that $\tau_1(i) \neq \tau_2(i)$. This implies that for any $i \in \{1, ..., n\}$

$$\nabla V(x) \cdot e^i = (\nabla V(x))_{\tau_1(i)}(\nabla V(x))_{\tau_2(i)} - (\nabla V(x))_{\tau_2(i)}(\nabla V(x))_{\tau_1(i)} = 0. \tag{C.9}$$

Note that, so long $\nabla V(x) \neq 0$, one can use this process to construct a generative family of $\mathcal{H}_x$.

Step 5: Generate at random k_1 real-valued functions $(f_i)_{i\in\{1,...,k_1\}}$ and k_2 real-valued functions $(g_i)_{i\in\{1,...,k_2\}}$, where $k_1 \geq 1$ and $k_2 \geq 1$ are chosen at random. Denote $f_i = 0$ for any $i \in \{k_1 + 1, ..., n\}$.

Step 6: Build the system

$$f(x) = -\left(f^2_{\tau_3(i)}(x)(\nabla V(x))_i\right)_{i\in\{1,...,n\}} + \sum_{i=1}^{k_2} g_i(x)e^{\tau(i)}(x), \tag{C.10}$$

where τ_3 is a random permutation of $\{1, ..., n\}$ to itself and τ is a random function from $\mathbb{N}$ to $\{1, ..., n\}$.

Overall, the function f satisfies

$$\nabla V(x) \cdot f(x) = -\left(\sum_{i=1}^{n} f^2_{\tau_3(i)}(x)(\nabla V(x))^2_i\right) \leq 0, \tag{C.11}$$

hence V is a Lyapunov function of the system

$$\dot{x}(t) = f(x(t)). \tag{C.12}$$

Step 7: Expand and simplify the equations of f (using Sympy), in order to eliminate obvious patterns due to the generation steps (that the model could recognize and leverage), avoid equivalent systems in the training set, and limit the length of training sequences. Polynomial systems are expanded into normal form.

We consider two generation modes:

Polynomial generation: we generate polynomial systems with sum-of-square Lyapunov functions to allow for easy comparison with existing methods such as SOSTOOLS [31, 32]. In this case, all P_i are polynomials with no zero-order term and $p_{1,c} = p_{1,m} = p_2 = 0$. Also, f_i and g_i are polynomials (Appendix C.1). We generate f_i with a degree lower or equal to half the maximal degree of g_i and a maximal value of coefficients of the order of the square root of the maximal value of g_i. Since the f_i are squared in the final system, this allows f_i^2 and g_i to have the same order, and prevents the transformer from inferring unwanted additional information by looking at the higher degree monomial.

Generic generation: P_i is generated as $P_i(x) = Q_i(x) - Q_i(0)$, where $Q_i(x)$ is a random function generated as per Appendix C.1 (and [20, 6]).

D Datasets and encodings

We use the techniques from the previous section to generate several datasets of $f(x)$ and $V(x)$, varying:

- whether or not the dataset contains only polynomial systems,
- the number of variables (and equations in the system, either *small* (2 to 3) or *large* (3 to 6)
- whether the system coefficients are integer or real.

Generated f and V, represented as trees, are encoded as sequences of tokens, using Polish notation to enumerate trees. For system coefficients, we use three encoding schemes for system coefficients, I10, I1000 for integers, and F10 for real numbers.

In I10 or I1000, integers as sequences of digits in base $b = 10$ or 1000, preceded by their sign (+ or -). For example, the number 12345 is encoded as $[+, 1, 2, 3, 4, 5]$ in I10 and as $[+, 12, 345]$ in I1000. Choosing $b = 10$ or $b = 1000$ is a tradeoff between the length of the encoded input and the size of the vocabulary (and of the input and output embeddings).

In F10, we encode a real number rounded to 4 significant digits, as a sequence of three tokens: the sign, the mantissa encoded as a sequence of digits in base 10 and the exponent, encoded as a symbolic token from `E-100` to `E100`. For instance, 3.14 is represented as $314 \cdot 10^{-2}$ and encoded as $[+, 3, 1, 4, \texttt{E-2}]$.

When generating datasets, we eliminate systems with input sequences longer than $T_x = 2048$, and ouput sequences longer than $T_y = 1024$, to avoid very long sequences that would exhaust GPU memory.

E Additional results

E.1 In-domain performance

We report the in-domain performance of our models, for different number of layers and attention heads (dimension 1024), by evaluating them on a fixed validation set of 10,000 examples, generated as per Section 3, using beam search of 1 (greedy decoding).

			layers=6				**layers=8**				**layers=10**			
dataset	equations	coefficients	hs=8	hs=10	hs=12	hs=16	hs=8	hs=10	hs=12	hs=16	hs=8	hs=10	hs=12	hs=16
Polynomial	2-3	Integer	98.4%	98.9%	99.0%	99.0%	99.0%	99.2%	98.8%	98.8%	99.3%	99.%	98.9%	99.0%
Polynomial	2-3	Float	87.0%	88.4%	87.3%	86.9%	87.7%	87.1%	86.0%	87.1%	87.7%	86.2%	87.8%	87.4%
Polynomial	3-6	Integer	95.4%	95.9%	95.9%	93.1%	93.9%	96.2%	95.7%	96.5%	93.1%	95.9%	96.5%	94.2%
Non-Polynomial	2-3	Integer	95.6%	96.8%	96.5%	96.1%	96.8%	97.2%	97.0%	96.9%	97.0%	96.8%	95.5%	95.0%

E.2 Out of domain generalization

We report the performance of our models (for various depths, 16 heads, and 1024 dimensions) and the impact of batch size, on the forward dataset, generated according to the method presented in Section 3. Note: results for the large polynomial training sets are limited to forward test set examples with 3 equations.

			layers=6			**layers=8**			**layers=10**		
dataset	equations	coefficients	bs=1	bs=10	bs=50	bs=1	bs=10	bs=50	bs=1	bs=10	bs=50
Polynomial	2-3	Integer	42.3%	60.1%	73.9%	42.4%	62%	77.1%	41.7%	59.8%	76.9%
Polynomial	2-3	Float	41.5%	52.6%	70.4%	39.8%	51.3%	67.5%	38.1%	52.1%	68.2%
Polynomial	3-6	Integer	38.1%	50.5%	61.1%	35.8%	50.8%	63.1%	37.4%	51.3%	65.7%
Non-Polynomial	2-3	Integer	43.1%	64.8%	82.1%	38.6%	64.7%	83.1%	36.4%	62.6%	78.9%