Efficient Robustness Verification of Neural Ordinary Differential Equations

Anonymous Author(s)
Affiliation
Address
email

Abstract

Neural Ordinary Differential Equations (NODEs) are a novel neural architecture, built around initial value problems with learned dynamics. Thought to be inherently more robust against adversarial perturbations, they were recently shown to be vulnerable to strong adversarial attacks, highlighting the need for formal guarantees. In this work, we tackle this challenge and propose GAINS, an analysis framework for NODEs based on three key ideas: (i) a novel class of ODE solvers, based on variable but discrete time steps, (ii) an efficient graph representation of solver trajectories, and (iii) a bound propagation algorithm operating on this graph representation. Together, these advances enable the efficient analysis and certified training of high-dimensional NODEs, which we demonstrate in an extensive evaluation on computer vision and time-series forecasting problems.

1 Introduction

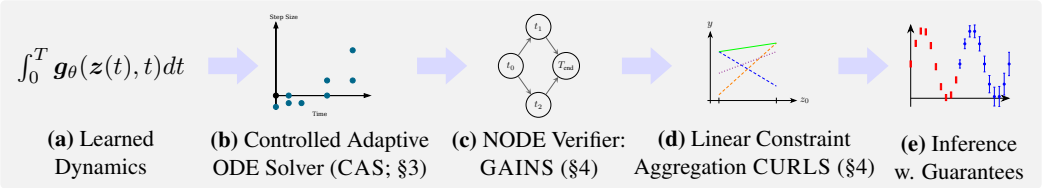
As deep learning enabled systems are increasingly deployed in safety-critical domains, developing neural architectures and specialized training methods that increase their robustness against adversarial examples (Szegedy et al., 2014; Biggio et al., 2013) is more important than ever.

Neural Ordinary Differential Equations (NODEs) (Chen et al., 2018) are built around initial value problems with learned dynamics and have been observed to inherently exhibit such robustness properties against adversarial attacks (Yan et al., 2020; Kang et al., 2021; Rodriguez et al., 2022). However, recently Huang et al. (2020) have found this robustness to be greatly diminished against stronger attacks, tracking it back to a gradient obfuscation effect of adaptive ODE solvers, thus highlighting the need for formal robustness guarantees.

Robustness Verification has been explored extensively for standard neural networks (Katz et al., 2017; Tjeng et al., 2019; Singh et al., 2018). However, methods successful in that setting can not be applied to NODEs as they can not handle the continuous step-size ranges arising for adaptive solvers. Despite first efforts towards NODE verification (Lopez et al., 2022), both scaling to high-dimensional problems and taking the effect of ODE solvers into account remain open problems.

This Work (illustrated in Fig. 1) tackles both of these problems, thereby enabling the systematic verification and study of NODE robustness as follows: (i) We introduce controlled adaptive ODE solvers (CAS) with step-sizes restricted to an exponentially spaced grid, yielding a finite number of time/step-size trajectories with minimal impact on solver efficiency. (ii) We introduce an efficient graph representation of these trajectories, allowing them to be merged and reducing their number from exponentially to quadratically many in the integration time. (iii) We extend the popular DEEPPOLY verifier (Singh et al., 2019) to efficiently operate on this trajectory graph by handling trajectory splitting in linear instead of exponential time. Combining these core ideas, we propose GAINS, a novel framework for the certified training and verification of NODEs.

Submitted to the DLDE Workshop in the 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.



(a) Learned Dynamics **(b)** Controlled Adaptive ODE Solver (CAS; §3) **(c)** NODE Verifier: GAINS (§4) **(d)** Linear Constraint Aggregation CURLS (§4) **(e)** Inference w. Guarantees

Figure 1: Given a learned NODE (a), we introduce controlled adaptive solvers (CAS) with discrete step-sizes (b). This enables us to construct the discrete trajectory graph (c), our NODE analysis framework GAINS is based on. To operate on this graph representation, we introduce CURLS to efficiently solve the arising linear constraint aggregation problem (d). For example, in the time-series forecasting setting, GAINS computes all possible outputs (blue error bars), for inputs in the red input ranges (e).

2 Background

Adversarial Robustness We consider time series forecasting models $\boldsymbol{f} \colon \mathbb{R}^{(d+1)\times L'+1} \mapsto \mathbb{R}^d$ that, given a time-series $\boldsymbol{x}^{L'} = \{(\boldsymbol{x}_j, t_j)\}_{j=1}^{L'}$ of data points $\boldsymbol{x}_j \in \mathbb{R}^d$ and a time t_L predict the values of the last data point $\boldsymbol{x}_L$. We call $\boldsymbol{f}$ ν-δ-robust on an ℓ_p-norm ball $\mathcal{B}_p^{\epsilon_p}(\boldsymbol{x})$ of radius ϵ_p around the original data points $\boldsymbol{x}^{L'}$ if the worst-case mean absolute error $\mathrm{MAE}_{\text{rob}}(\boldsymbol{x}')$ for $\boldsymbol{x}' \in \mathcal{B}_p^{\epsilon_p}(\boldsymbol{x}^{L'})$ is linearly bounded by the original input's $\mathrm{MAE}(\boldsymbol{x}^{L'})$ as

$$\mathrm{MAE}_{\text{rob}}(\boldsymbol{x}) < (1+\nu)\,\mathrm{MAE}(\boldsymbol{x}) + \delta, \quad \text{with} \quad \mathrm{MAE}_{\text{rob}}(\boldsymbol{x}^{L'}) = \max_{\boldsymbol{x}' \in \mathcal{B}_p^{\epsilon_p}(\boldsymbol{x}^{L'})} \mathrm{MAE}(\boldsymbol{x}'). \tag{1}$$

Neural Network Verification aims to decide whether a robustness property holds. Here, we consider bound propagation methods, i.e., methods determining a lower and upper bound l, u for each neuron $l \leq x \leq u$, either by propagating hyper-boxes (dimensionwise intervals) (Gehr et al., 2018; Mirman et al., 2018) or linear constraints (Singh et al., 2019; Zhang et al., 2018), where every layer's neurons $\boldsymbol{x}_i$ are lower- and upper-bounded depending only on the previous layer's neurons:

$$\boldsymbol{A}_i^- \boldsymbol{x}_{i-1} + \boldsymbol{c}_i^- =: \boldsymbol{l}_i \leq \boldsymbol{x}_i, \quad \boldsymbol{x}_i \leq \boldsymbol{u}_i := \boldsymbol{A}_i^+ \boldsymbol{x}_{i-1} + \boldsymbol{c}_i^+. \tag{2}$$

Given these linear constraints, we can recursively substitute $\boldsymbol{x}_{i-1}$ with its linear bounds in terms of $\boldsymbol{x}^{i-2}$ until we have obtained bounds depending only on the input $\boldsymbol{x}^0$. This allows us to compute concrete bounds $\boldsymbol{l}$ and $\boldsymbol{u}$ on any linear expression over network neurons.

Neural Ordinary Differential Equations are built around an initial value problem (IVP), defined by an input state $\boldsymbol{z}(0) = \boldsymbol{z}_0$ and a neural network $\boldsymbol{g}_\theta$, defining the dynamics of an ordinary differential equation (ODE) $\nabla_t \boldsymbol{z}(t) = \boldsymbol{g}(\boldsymbol{z}(t), t)$. We obtain its solution at time T_{end} by evaluating $\boldsymbol{z}(T_{\text{end}}) = \boldsymbol{z}(0) + \int_0^{T_{\text{end}}} \boldsymbol{g}(\boldsymbol{z}(t), t)\, dt$ with an ODE solver at the predefined integration time T_{end}.

3 Controlled Adaptive ODE Solvers

Adaptive step-size solvers (AS) control their step-size to efficiently solve an ODE up to a predefined error τ (Dormand & Prince, 1980; Bogacki & Shampine, 1989). They use two methods of different order p to compute the proposal solutions $\boldsymbol{z}^1$ and $\boldsymbol{z}^2$, derive a normalized error estimate $\delta = \|\frac{\boldsymbol{z}^1 - \boldsymbol{z}^2}{\tau}\|_1$ and update their step size to $h \leftarrow h\delta^{-1/p}$. However, this dependence of the step-size h on the error estimate δ, yields infinitely many trajectories for continuous input regions, making their abstraction intractable. To obtain solvers amenable to certification, we propose controlled adaptive solvers (CAS) by restricting possible step-sizes to a discrete set. To implement this change, we modify the step-size update rule of any AS to

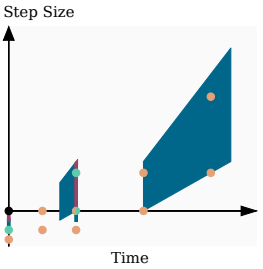


Figure 2: Reachable time/step-size tuples for CAS and adaptive solvers after one (●, ■) and two solver steps (●, ■).

$$h \leftarrow \begin{cases} h \cdot \alpha, & \text{if } \delta \leq \tau_\alpha, \\ h, & \text{if } \tau_\alpha < \delta \leq 1, \\ h/\alpha, & \text{otherwise.} \end{cases} \qquad \delta = \left\| \frac{\hat{\boldsymbol{z}}^1(t+h) - \hat{\boldsymbol{z}}^2(t+h)}{\tau} \right\|_1,$$

with update factor $\alpha \in \mathbb{R}^{>1}$, and the α-induced decision threshold $\tau_\alpha = \alpha^{-p}$. Intuitively, we increase the step size by a factor α if we expect the error after this increase to still be acceptable, i.e., $\delta \leq \alpha^{-p}$, we decrease the step size by a factor α and repeat the step if the error exceeds our tolerance, i.e., $\delta > 1$, and we keep the same step size otherwise. For more details, see App. D.1.

We contrast the reachable time/step-size states of CAS and AS solvers in Fig. 2. Initialized with the same state (●), the CAS solver can reach exactly three states after one (●) and nine states after two steps (●). The AS solver, in contrast, can reach increasingly large continuous state spaces after one (■) and two (■) steps.

Comparison to Adaptive Solvers CAS solvers can be seen as adaptive solvers with discretized step-sizes of the same order. Due to the exponentially spaced step-sizes, CAS will (under mild assumptions) need at most α-times as many steps as an adaptive solver and empirically sometimes actually fewer. We illustrate this on a conventional non-linear ODE in Fig. 3.

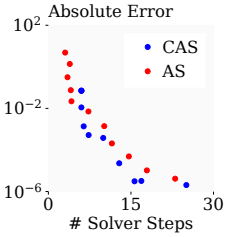


Figure 3: AS and CAS error over solver steps. (Details in App. D.2)

4 Verification of Neural Ordinary Differential Equations

A solver step corresponds to the weighted sum of network evaluations $\boldsymbol{g}(\boldsymbol{z})$ and can thus be abstracted using standard bound propagation methods. We call this an *abstract solver step*. While we could thus compute bounds for each of the exponentially many discrete trajectories obtained for CAS solvers, this would be intractably slow. We instead introduce the analysis framework GAINS, short for **G**raph based **A**bstract **I**nterpretation for **N**ODE**s**, which leverages a trajectory graph representation to compute bounds efficiently. Given a NODE with input $\mathcal{Z}$, we represent all trajectories $\Gamma(\boldsymbol{z}_0')$ for $\boldsymbol{z}_0' \in \mathcal{Z}$ in the trajectory graph $\mathcal{G}(\mathcal{Z}) = (\mathcal{V}, \mathcal{E})$ as follows. Each node $v \in \mathcal{V}$ represents a solver states (t, h) with time t, step-size h and interval bounds on $\boldsymbol{z}(t)$. Each directed edge $e \in \mathcal{E}$ connects two states appearing consecutively in a possible solver trajectory. This representation allows states $\boldsymbol{z}(t)$ with identical time and step-size to be merged, leaving only quadratically many ($\mathcal{O}(T_{\text{end}}^2 \log^2(T_{\text{end}}))$) solver steps to be considered, thus making the analysis tractable.

Trajectory Graph Construction We initialize the trajectory graph with a node for the initial state $(0, h_0)$ and proceed as follows: Among all nodes without outgoing edges, we chose the one with the smallest time t and largest step-size h (in that order). We apply an abstract solver step to this state, computing interval bounds on the error estimate $\delta_{(t,h)}$ and determining possible step-size updates (increase, accept, or decrease). If multiple updates are possible, we call this trajectory splitting. For each possible update, we obtain a new state (t', h') and add a corresponding node to $\mathcal{V}$ and an edge from (t, h) to (t', h') to $\mathcal{E}$. If the node (t', h') already existed, we update the bounds on state $\boldsymbol{z}(t)$ to contain the newly computed ones. We repeat this procedure until all trajectories have reached the termination node $(T_{\text{end}}, 0)$, yielding a complete trajectory graph and interval bounds for $\boldsymbol{z}(T_{\text{end}})$.

Verification with Linear Bounds We can derive more precise, linear bounds on $\boldsymbol{z}(T_{\text{end}})$ in terms of the NODE input $\boldsymbol{z}_0$ by recursively substituting bounds through the trajectory graph. Starting with the bounds for $(T_{\text{end}}, 0)$, we backsubstitute them along the solver step represented by every incoming edge using standard DEEPPOLY primitives and yielding a set of bounds in every preceding node. We recursively repeat this backsubstitution procedure until we arrive at the input node. We illustrate this in Fig. 4, where we backsubstitute $\boldsymbol{y}$ to t_1 and t_2, obtaining bounds in terms of $\boldsymbol{z}_1$ and $\boldsymbol{z}_2$, respectively. After an additional backsubstitution step to t_0, we obtain two bounds, $\boldsymbol{u}^1$ and $\boldsymbol{u}^2$, on $\boldsymbol{y}$, both in terms of $\boldsymbol{z}_0$. We now have to merge them into a single linear bound. We call this the linear constraint aggregation problem (LCAP) and observe that it arises for every node where trajectory splitting occurred.

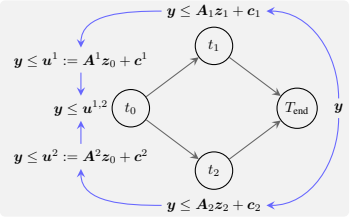


Figure 4: Backsubstitution (blue arrows) of upper bounds on $\boldsymbol{y} = \boldsymbol{z}(T_{\text{end}})$ via GAINS, resulting in an LCAP at t_0.

Linear Constraint Aggregation Problem Consider m upper bounds $\{\boldsymbol{u}^j\}_{j=1}^m$ on $y = \boldsymbol{z}(T_{\text{end}})$, that are linear in $\boldsymbol{z}_0 \in \mathcal{Z}$. We now wish to aggregate them into a single linear upper bound $\boldsymbol{y} \leq \boldsymbol{u}^* := \boldsymbol{a}\boldsymbol{z}_0 + c$ minimizing the volume between $\boldsymbol{u}^*$ and the $\boldsymbol{y} = 0$ plane over $\boldsymbol{z}_0 \in \mathcal{Z}$. Solving this problem optimally with an LP by considering the constraints induced by all 2^d corners of $\mathcal{Z}$ becomes intractable even in modest dimensions. We, thus, propose **C**onstraint **U**nification via **R**e**L**U **S**implification (CURLS), translating the $\max\{\boldsymbol{u}^j\}_{j=1}^m$ into a composition of ReLUs, efficiently handled by DEEPPOLY (Singh et al., 2019). For a pair of constraints $\boldsymbol{u}_i^1, \boldsymbol{u}_i^2$ we rewrite their maximum as $\max_{j \in 1,2} \boldsymbol{u}_i^j = \boldsymbol{u}_i^1 + \max(0, \boldsymbol{u}_i^2 - \boldsymbol{u}_i^1) = \boldsymbol{u}_i^1 + \text{ReLU}(\boldsymbol{u}_i^2 - \boldsymbol{u}_i^1)$. For $m > 2$ constraints, we apply this rewrite multiple times. We note that lower bounds can be merged analogously.

Table 1: Mean absolute errors for the unperturbed samples (Std. MAE) and certifiable (Cert.) ν-δ-robustness with $\nu = 0.1$ and $\delta = 0.01$.

Setting	Training Method	ϵ_t	Std. MAE [$\times$ 100]	$\epsilon = 0.05$		$\epsilon = 0.10$		$\epsilon = 0.20$	
				Adv. [%]	Cert. [%]	Adv. [%]	Cert. [%]	Adv. [%]	Cert. [%]
12h	Standard		**48.8**$^{\pm 0.2}$	72.9$^{\pm 1.3}$	0.0$^{\pm 0.0}$	23.4$^{\pm 3.6}$	0.0$^{\pm 0.0}$	5.6$^{\pm 1.0}$	0.0$^{\pm 0.0}$
	GAINS	0.1	53.6$^{\pm 2.2}$	98.2$^{\pm 0.6}$	94.1$^{\pm 0.2}$	78.5$^{\pm 2.1}$	57.6$^{\pm 4.2}$	33.2$^{\pm 4.1}$	20.5$^{\pm 2.9}$
		0.2	54.6$^{\pm 2.1}$	**98.8**$^{\pm 0.3}$	**97.6**$^{\pm 0.5}$	**88.6**$^{\pm 0.3}$	**80.5**$^{\pm 2.9}$	**48.9**$^{\pm 1.9}$	**36.8**$^{\pm 1.9}$

5 Experimental Evaluation

See App. F for the experimental setup and App. C for a significantly expanded empirical evaluation.

Time-Series Forecasting We consider PHYSIO-NET (see App. F) and predict the last measurement L, without having access to the preceding 12 hours of data (typically leaving 36 hours). In Table 1, we report the resulting mean absolute prediction error (MAE) for the unperturbed samples and ν-δ-robustness (see Eq. (1)) for a relative and absolute error tolerance of $\nu = 0.1$ and $\delta = 0.01$, respectively, at perturbation magnitudes $\epsilon = \{0.05, 0.1, 0.2\}$. With a normally trained latent ODE, we obtain an MAE of 0.49, but observe that it is vulnerable to adversarial attacks and robustness can not be certified even for the smallest perturbation radii ($\epsilon = 0.05$). To improve robustness, we use GAINS to compute a bound on the worst-case loss and optimize that instead of the standard loss. This so-called certified training with radius ϵ_t results in a small drop in standard MAE, but a significant increase in adversarial robustness, certifying as much as 97.6% of the samples. We thus conclude that GAINS can successfully train provably robust NODEs and prove their robustness.

Trajectory Sensitivity We investigate whether the solver trajectory, i.e. the chosen step-sizes, of CAS solvers are still susceptible to adversarial attacks, reporting the portion of trajectories that could be manipulated in Table 2. We observe that for normally trained NODEs, almost all trajectories can be manipulated, even with moderate perturbations ($\epsilon = 0.1$). While training with GAINS reduces this susceptibility notably, it remains significant. This highlights the need to consider solver effects on robustness, motivating both the use of CAS solvers and the trajectory graph-based approach of GAINS.

Table 2: Portion of successfully manipulated trajectories for 1000 MNIST image classification tasks.

Training	ϵ_t	Attack Success [%]	
		$\epsilon = 0.1$	$\epsilon = 0.2$
Standard		98.9$^{\pm 0.3}$	100.0$^{\pm 0.0}$
GAINS	0.11	73.4$^{\pm 3.5}$	95.5$^{\pm 1.8}$
	0.22	65.2$^{\pm 7.5}$	82.2$^{\pm 5.0}$

Linear Constraint Aggregation To evaluate CURLS, we compare it against an LP-based approach (see App. H) for solving the Linear Constraint Aggregation Problem (LCAP). In Fig. 5, we illustrate normalized abstraction volumes $\mathrm{vol}^{\mathrm{LP}} / \mathrm{vol}^{\mathrm{CURLS}}$ and runtimes for the two methods and randomly generated constraint sets in $d = [5, 100]$ dimensions. While the LP-based solutions are more precise for up to 75-dimensional problems, they take around 5 orders of magnitude longer to compute. For higher dimensional problems, CURLS is both faster and more precise. As certification of a single input involves multiple hundred high-dimensional LCAP problems, CURLS is essential to make verification with GAINS tractable.

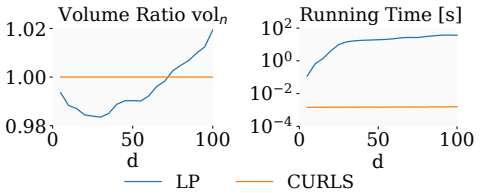


Figure 5: Comparison of th CURLS and LP solution to the LCAP with respect to normalized volume (left) and runtime (right).

6 Conclusion

In this work, we propose the analysis framework GAINS, **G**raph based **A**bstract **I**nterpretation for **N**ODE**s**, which, for the first time, allows the verification and certified training of high dimensional NODEs based on the following key ideas: i) We introduce CAS solvers which retain the efficiency of adaptive solvers but are restricted to discrete instead of continuous step-sizes. ii) We leverage CAS solvers to construct efficient graph representations of all possible solver trajectories given an input region. iii) We build on linear bound propagation based neural network analysis and propose new algorithms to efficiently operate on these graph representations. Combined, these advances enable GAINS to analyze NODEs under consideration of solver effects in polynomial time.

References

Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Srndic, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part III*, volume 8190, 2013. doi: 10.1007/978-3-642-40994-3_25.

Przemyslaw Bogacki and Lawrence F Shampine. A 3 (2) pair of runge-kutta formulas. *Applied Mathematics Letters*, 2(4), 1989.

Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, 2018.

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder–decoder approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*, 2014. doi: 10.3115/v1/W14-4012.

John R Dormand and Peter J Prince. A family of embedded runge-kutta formulae. *Journal of computational and applied mathematics*, 6(1), 1980.

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. AI2: safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, 2018. doi: 10.1109/SP.2018.00058.

Yifei Huang, Yaodong Yu, Hongyang Zhang, Yi Ma, and Yuan Yao. Adversarial robustness of stabilized neuralodes might be from obfuscated gradients. *ArXiv preprint*, abs/2009.13145, 2020.

Qiyu Kang, Yang Song, Qinxu Ding, and Wee Peng Tay. Stable neural ode with lyapunov-stable equilibrium points for defending against adversarial attacks. *Advances in Neural Information Processing Systems*, 34, 2021.

Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. *ArXiv preprint*, abs/1702.01135, 2017.

Hoki Kim. Torchattacks: A pytorch repository for adversarial attacks. *ArXiv preprint*, abs/2010.01950, 2020.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proc. of ICLR*, 2015.

Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In *Proc. of ICLR*, 2014.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11), 1998. doi: 10.1109/5.726791.

Diego Manzanas Lopez, Patrick Musau, Nathaniel Hamilton, and Taylor T Johnson. Reachability analysis of a general class of neural ordinary differential equations. *ArXiv preprint*, abs/2207.06531, 2022.

Matthew Mirman, Timon Gehr, and Martin T. Vechev. Differentiable abstract interpretation for provably robust neural networks. In *Proc. of ICML*, volume 80, 2018.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, 2019.

Ivan Dario Jimenez Rodriguez, Aaron Ames, and Yisong Yue. Lyanet: A lyapunov framework for training neural odes. In *International Conference on Machine Learning*. PMLR, 2022.

Yulia Rubanova, Ricky TQ Chen, and David Duvenaud. Latent odes for irregularly-sampled time series. arxiv. *Search in*, 2019.

Ikaro Silva, George Moody, Daniel J Scott, Leo A Celi, and Roger G Mark. Predicting in-hospital mortality of icu patients: The physionet/computing in cardiology challenge 2012. In *2012 Computing in Cardiology*. IEEE, 2012.

Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin T. Vechev. Fast and effective robustness certification. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, 2018.

Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL), 2019.

Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian J. Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *Proc. of ICLR*, 2014.

Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. In *Proc. of ICLR*, 2019.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *ArXiv preprint*, abs/1708.07747, 2017.

Kaidi Xu, Zhouxing Shi, Huan Zhang, Yihan Wang, Kai-Wei Chang, Minlie Huang, Bhavya Kailkhura, Xue Lin, and Cho-Jui Hsieh. Automatic perturbation analysis for scalable certified robustness and beyond. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

Hanshu Yan, Jiawei Du, Vincent Y. F. Tan, and Jiashi Feng. On robustness of neural ordinary differential equations. In *Proc. of ICLR*, 2020.

Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. Efficient neural network robustness certification with general activation functions. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, 2018.

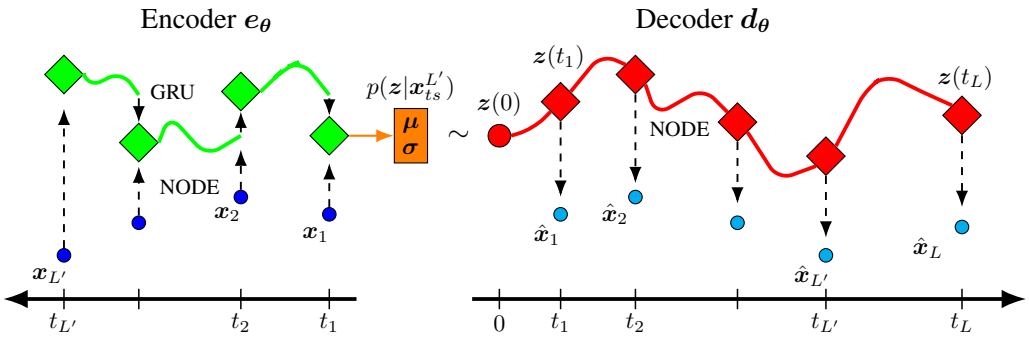


Figure 6: Visualization of the latent ODE with ODE-RNN encoder. Due to the NODE layer in the decoder the model is able to estimate the data point of the time-series at any desired time. Figure inspired by (Chen et al., 2018; Rubanova et al., 2019).

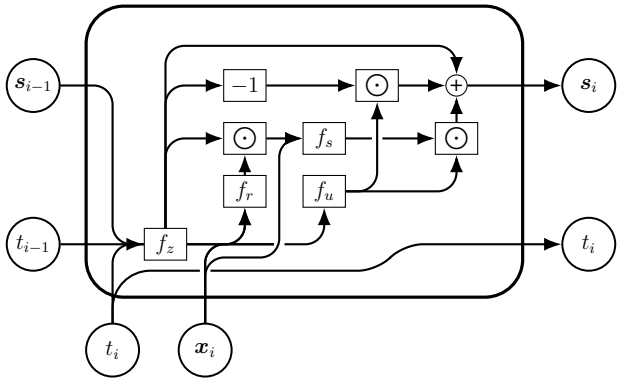


Figure 7: GRU-update for the ODE-RNN architecture, where $\odot$ denotes the hadamard product (componentwise multiplication) of two vectors and $f_z,\ f_u,\ f_r,\ f_s$ are auxiliary NNs.

A Latent ODEs for Time-Series Forcasting

For time-series forecasting, we use an encoder-decoder architecture called latent ODE (Rubanova et al., 2019) and illustrated in Fig. 6. The encoder $\boldsymbol{e_\theta}$ is an ODE-RNN, yielding an embedding $\boldsymbol{s}_{L'}$ of the data points observed until $t_{L'}$, where the series is processed in reversed time order. The core idea is to describe the evolution of a hidden state with a NODE and update it using a GRU unit (Cho et al., 2014) (described in App. A.1) to account for new observations. This embedding is then passed through a one layer MLP to yield the posterior distribution $p(\boldsymbol{z}|\boldsymbol{x}_{ts}^{L'}) = \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma})$ over the initial state of the decoder $\boldsymbol{z}(0)$. The decoder $\boldsymbol{d_\theta}$ then estimates $\hat{\boldsymbol{x}}_L$ as a linear transform of the solution $\boldsymbol{z}(t_L)$ of the IVP with initial state $\boldsymbol{z}(0)$ at time t_L. Note that in testing we use $\boldsymbol{z}(0) = \boldsymbol{\mu}$ and omit the sampling.

The latent ODE is trained to maximize the evidence lower bound (ELBO) (Kingma & Welling, 2014) an minimize the absolute error of the final predictions weighted with γ:

$$\mathcal{L}_f(\boldsymbol{x}_{ts}^{L}, L') = \gamma \cdot \|\hat{\boldsymbol{x}}_L - \boldsymbol{x}_L\|_1 - \mathrm{ELBO}(\boldsymbol{x}_{ts}^{L}, L') \tag{3}$$

$$\mathrm{ELBO}(\boldsymbol{x}_{ts}^{L}, L') = \mathbb{E}_{\boldsymbol{z}' \sim p_{\mathcal{N}}}\left[\log\left(\boldsymbol{d_\theta}\left(\boldsymbol{z}', t_L\right)\right)\right] - D_{\mathrm{KL}}\left[p_{\mathcal{N}} || p\right]. \tag{4}$$

A.1 GRU update

In Fig. 7 we show the update of the hidden state $\boldsymbol{s}_{i-1}$ for ODE-RNN (Rubanova et al., 2019) architecture after feeding the i-th entry $(\boldsymbol{x}_i, t_i)$ as input. The update uses a NODE layer to represent f_z, where the integration domain of the NODE layer is $[t_{i-1}, t_i]$.

B Provable NODE Training

In this section, we describe our GAINS-based training procedure. Standard provable training aims to optimize a loss based on the over-approximation (**??**). However, in the case of NODE it is intractable to compute the full over-approximation of the trajectory graph as discussed in §4 for each sample in training. Thus, we only sample up to κ selected trajectories from $\mathcal{G}(\mathcal{Z})$.

We consider data $\boldsymbol{x}, y$ we compute the NODE input $\boldsymbol{z}_0$ (either $\boldsymbol{z}_0 := \boldsymbol{x}$ or via some encoder) and the corresponding bounds $\mathcal{Z}$. If $\mathcal{G}(\mathcal{Z})$ exhibits trajectory splitting, i.e. different step sizes could be taken, we sample κ different trajectories.

Trajectory Exploration During the sampling we balance exploration of the full trajectory graph and staying close to the reference trajectory, the trajectory $\Gamma(\boldsymbol{z}_0)$ of the solver with unperturbed input $\boldsymbol{z}_0$. A visualization of the selection process is depicted in Fig. 8.

We select trajectories as follows: We start the propagation of $\mathcal{Z}$ through the NODE solver. Recall that, for a concrete input at each step the CAS solver will either *(i) increase*, *(d) decrease* or *(a) accept*, i.e., keep, the current step size h. For an abstract solver step we may need to keep track of multiple decisions (trajectory splitting). Thus, for each abstract solver step we check whether or not trajectory splitting occurs and as long as no trajectory split occurs, we are following the reference trajectory. If, however, multiple time step updates are possible, i.e., we encounter trajectory splitting, we choose a single path u via random sampling (details below), and add the corresponding state to the branching point set $\mathcal{C}$. Afterwards, we check whether or not we have reached T_{end}, where if T_{end} is reached, we save the resulting trajectory to a set $\mathcal{S}$. After having reached T_{end}, we repeat the process with a checkpoint $C \in \mathcal{C}$, as long as there is still a checkpoint in $\mathcal{C}$, i.e. $|\mathcal{C}| > 0$, and we have not already collected κ trajectories, i.e. $|\mathcal{S}| < \kappa$.

Loss Computation Finally, we compute the BOX output of the NODE layer as the over-approximation of the final states form all saved trajectories $\mathcal{S}$. Then, for provable training we use a loss term of the following form:

$$\mathcal{L}(\boldsymbol{z}_0, \mathcal{Z}, y) = (1 - {}^{\omega_1 \epsilon'}/_{\epsilon_t})\mathcal{L}_{\text{std}}(\boldsymbol{z}_0, y) + {}^{\omega_1 \epsilon'}/_{\epsilon_t}\mathcal{L}_{\text{rob}}(\mathcal{Z}, y) + \omega_2 \|\boldsymbol{u}_{\text{out}} - \boldsymbol{l}_{\text{out}}\|_1, \tag{5}$$

where $\mathcal{L}_{\text{std}}$ is the standard loss (depending on the task) evaluated on the unperturbed sample, and $\mathcal{L}_{\text{rob}}$ is an over-approximation of $\mathcal{L}_{\text{std}}$ based on the abstraction obtained from $\mathcal{S}$. The term $\boldsymbol{u}_{\text{out}} - \boldsymbol{l}_{\text{out}}$ regularizes the bound width of the corresponding output region. During training, we anneal ϵ, gradually increasing ϵ' from 0 to ϵ, thereby shifting focus from the standard to the robust loss term. In the classification setting, we use the cross entropy loss and in time series forecasting we use a latent ODE specific loss, combining a MAE error and ELBO term, defined in Eq. (3).

Sampling Updates For a state (t, h) we let $V_{(t,h)}$ denote the set of vertices which where traversed from initial vertex $(0,\ h_0)$ to $(t,\ h)$. Moreover, for any vertex $v = (\tilde{t},\ \tilde{h})$ we define its reference vertex $v' = (\tilde{t}',\ \tilde{h}')$ as the vertex with the smallest ℓ_1-distance to the vertex v among the vertices in the reference trajectory $\Gamma(\boldsymbol{z}_0)$, i.e.

$$v' = (\tilde{t}',\ \tilde{h}') = \underset{(\hat{t},\hat{h}) \in \Gamma(\boldsymbol{z}_0)}{\arg\min}\ |\tilde{t} - \hat{t}| + |\tilde{h} - \hat{h}|. \tag{6}$$

Furthermore, for any vertex $v \in V_{(t,h)}$ we let $u(v)$ denote the update (*(i) increase*, *(d) decrease* or *(a) accept*) taken to leave state v in the given trajectory. Analogously, we define for any $v' \in \Gamma(\boldsymbol{z}_0)$ $u'(v')$ as the performed update in $\Gamma(\boldsymbol{z}_0)$ after vertex v'.

Additionally, we define the auxiliary mapping $g_n : \{d, a, i\} \rightarrow \{0,\ 1,\ 2\}$, where $g_n(d) = 0,\ g_n(a) = 1$ and $g_n(i) = 2$. Using the previous definitions we define the location index of $V_{(t,h)}$ as $n(V_{(t,h)}) = \sum_{v \in V_{(t,h)}} g_n(u(v)) - g_n(u'(v'))$. If the location index is bigger than zero, we assume to be traversing a trajectory that has performed steps with bigger step sizes than the reference trajectory $\Gamma(\boldsymbol{z}_0)$. On the other hand, for a location index smaller than zero the opposite is true, whereas if the location index is zero we are close to the reference trajectory $\Gamma(\boldsymbol{z}_0)$.

Finally, when sampling an update u we choose from the categorical distribution $P_u(p_d, p_a, p_i)$ depending on $n(V_{(t,h)}), u'(v')$ for the current state (t, h) and hyperparameters q_1 and q_2. The definition of the probabilities $p_d,\ p_a$ and p_i can be seen in Table 3.

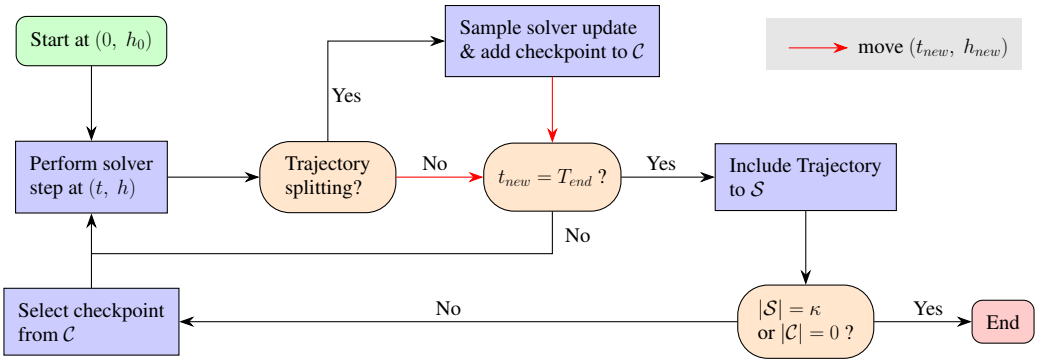


Figure 8: Selection process of $\mathcal{S}$, which contains at most κ trajectories starting with initial step size h_0 and final integration time T_{end} and the branching point set $\mathcal{C}$.

In the definition of the sample probabilities the update that pushes the location index the most towards zero occurs always with probability $1 - q_1 - q_2$, whereas the event occurring with probability q_1 pushes the location index away from zero. Hence, depending on which probability is higher, we either prefer to select trajectories close to the reference trajectory or trajectories that are distributed over the entire trajectory graph. In order to have a combination of both, we use an annealing process for the hyperparameters q_1 and q_2. In the early stages of training, we choose selection hyperparameters such that $1 - q_1 - q_2 \geq q_2 \geq q_1$, i.e. stay close to the reference trajectory, and towards the end of the training the chain of inequalities should be reversed, i.e. cover the entire trajectory graph and not just a region.

Checkpoint Selection Criterion We use the following decision criterion to select C^* from $\mathcal{C}$

$$C^* = \underset{C=\{V_C\}\in\mathcal{C}}{\arg\max} \frac{|n(V_C) - n_{\mathcal{S}}|}{2} - |V_C| - \sigma_{out}[V_C], \tag{7}$$

where the vertex set V_C contains all traversed vertices until the creation of the checkpoint C and we denote by $n_{\mathcal{S}}$ the average location index of the already stored trajectories in $\mathcal{S}$. Observe, that the decision criterion is designed such that checkpoints in under-explored regions of the trajectory graph and checkpoints arising early in the trajectory graph are favored, where the former statement is captured by the first term in Eq. (7), whereas the remaining two terms capture the latter statement.

Table 3: The definition of the probabilities $p_d,\ p_a$ and p_i depending on the location index $n(V)$, reference update u' and hyperparameters $q_1,\ q_2$.

$n(V)$	u'	p_d	p_a	p_i
$n = 0$	a	$\frac{q_1+q_2}{2}$	$1 - q_1 - q_2$	$\frac{q_1+q_2}{2}$
$n = 0$ $n > 0$	d $\{d, a, i\}$	$1 - q_1 - q_2$	q_2	q_1
$n = 0$ $n < 0$	i $\{d, a, i\}$	q_1	q_2	$1 - q_1 - q_2$

C Additional Experiments

Experimental Setup We implement GAINS in PyTorch (Paszke et al., 2019) and evaluate all benchmarks using single NVIDIA RTX 2080Ti. We conduct experiments on MNIST (LeCun et al., 1998), FMNIST (Xiao et al., 2017), and PHYSIO-NET (Silva et al., 2012). For image classification we use an architecture consisting of two convolutional and one NODE layer (see Table 6 in App. E for more details). For time-series forecasting we use a latent ODE (see Table 7 in App. F for more details). We provide detailed hyperparameter choices in App. E and F.

Table 4: Mean and standard deviations of the standard (Std.), adversarial (Adv.), and certified (Cert.) accuracy obtained with GAINS depending on the training method and evaluated on the first 1000 test set samples.

Dataset	Training Method	ϵ_t	Std. [%]	$\epsilon = 0.10$		$\epsilon = 0.15$		$\epsilon = 0.20$	
				Adv. [%]	Cert. [%]	Adv. [%]	Cert. [%]	Adv. [%]	Cert. [%]
MNIST	Standard		$98.8^{\pm 0.4}$	$23.2^{\pm 3.5}$	$0.0^{\pm 0.0}$	$2.5^{\pm 1.6}$	$0.0^{\pm 0.0}$	$0.3^{\pm 0.2}$	$0.0^{\pm 0.0}$
	Adv.	0.11	$\mathbf{99.2}^{\pm 0.1}$	$\mathbf{95.4}^{\pm 0.4}$	$0.0^{\pm 0.0}$	$\mathbf{88.3}^{\pm 0.6}$	$0.0^{\pm 0.0}$	$59.4^{\pm 3.2}$	$0.0^{\pm 0.0}$
	GAINS	0.11	$95.5^{\pm 0.1}$	$91.5^{\pm 0.6}$	$\mathbf{83.9}^{\pm 1.3}$	$84.0^{\pm 2.7}$	$17.7^{\pm 9.0}$	$21.4^{\pm 1.8}$	$0.0^{\pm 0.0}$
		0.22	$91.8^{\pm 1.3}$	$88.5^{\pm 1.8}$	$82.7^{\pm 3.4}$	$86.8^{\pm 2.1}$	$\mathbf{69.4}^{\pm 6.5}$	$\mathbf{84.5}^{\pm 3.2}$	$\mathbf{50.9}^{\pm 9.1}$
FMNIST	Standard		$\mathbf{88.6}^{\pm 1.2}$	$0.1^{\pm 0.1}$	$0.0^{\pm 0.0}$	$0.0^{\pm 0.0}$	$0.0^{\pm 0.0}$		
	Adv.	0.11	$80.9^{\pm 0.7}$	$\mathbf{70.2}^{\pm 0.5}$	$0.0^{\pm 0.0}$	$47.1^{\pm 3.7}$	$0.0^{\pm 0.0}$		
	GAINS	0.11	$75.1^{\pm 1.2}$	$65.7^{\pm 1.0}$	$\mathbf{56.3}^{\pm 1.4}$	$21.1^{\pm 5.9}$	$8.4^{\pm 2.3}$		
		0.16	$71.5^{\pm 1.7}$	$64.0^{\pm 2.7}$	$54.7^{\pm 2.5}$	$\mathbf{60.1}^{\pm 3.5}$	$\mathbf{42.7}^{\pm 1.4}$		

Adversarial Robustness in Classification We consider classification models $\boldsymbol{f} \colon \mathbb{R}^d \mapsto \mathbb{R}^c$ that, given an input $\boldsymbol{x} \in \mathcal{X} \subseteq \mathbb{R}^d$, predict c numerical values $\boldsymbol{y} := \boldsymbol{f}(\boldsymbol{x})$, interpreted as class confidences. We call $\boldsymbol{f}$ adversarially robust on an ℓ_p-norm ball $\mathcal{B}_p^{\epsilon_p}(\boldsymbol{x})$ of radius ϵ_p, if it predicts target class t for all perturbed inputs $\boldsymbol{x}' \in \mathcal{B}_p^{\epsilon_p}(\boldsymbol{x})$. More formally, we define *adversarial robustness* as:

$$\arg\max_j h(\boldsymbol{x}')_j = t, \quad \forall \boldsymbol{x}' \in \mathcal{B}_p^{\epsilon_p}(\boldsymbol{x}) := \{x' \in \mathcal{X} \mid \|\boldsymbol{x} - \boldsymbol{x}'\|_p \leq \epsilon_p\}. \tag{8}$$

C.1 Classification

We train NODE based networks with standard, adversarial, and provable training ($\epsilon_t \in \{0.11, 0.22\}$) and certify robustness to ℓ_∞-norm bounded perturbations of radius ϵ as defined in Eq. (8). We report means and standard deviations across three runs at different perturbation levels ($\epsilon \in \{0.1, 0.15, 0.2\}$) depending on the training method in Table 4. Both for MNIST and FMNIST, adversarial accuracies are low (0.0% to 23.2%) for standard trained NODEs, agreeing well with recent observations showing vulnerabilities to strong attacks (Huang et al., 2020). While adversarial training can significantly improve robustness even against these stronger attacks, we can not certify any robustness. Using provable training with GAINS significantly improves certifiable accuracy (to up to 84% depending on the setting) while reducing standard accuracy only moderately. This trade-off becomes more pronounced as we consider increasing perturbation magnitudes for training and certification.

C.2 Time-Series Forecasting

For time-series forecasting, we consider the PHYSIO-NET Silva et al. (2012) dataset, containing $8\,000$ time-series of up to 48 hours of 35 irregularly sampled features. We rescaling most features to mean $\mu = 0$ and standard deviation $\sigma = 1$ (before applying perturbations) and refer to App. F for more details. We consider three settings, where we predict the last measurement L, without having access to the preceding 6, 12, or 24 hours of data. In Table 5, we report the mean absolute prediction error (MAE) for the unperturbed samples and ν-δ-robustness (see Eq. (1)) for a relative and absolute error tolerances of $\nu = 0.1$ and $\delta = 0.01$ at perturbation magnitudes $\epsilon = \{0.05, 0.1, 0.2\}$. We observe only a minimal drop in standard precision, when certifiably training with GAINS at moderate perturbation magnitudes ($\epsilon_t = 0.1$) while increasing both adversarial and certified accuracies substantially. Further, while we can again not verify any robustness for standard trained NODEs, they exhibit non-vacuous empirical robustness. However, without guarantees it remains unclear whether this is due to adversarial examples being harder to find or NODEs being inherently more robust in the time-series forecasting setting. Interestingly, MAEs are smallest for the $12h$ setting, despite a longer forecast horizon than in the $6h$ setting. We hypothesize that this is due to the larger number of input points and thus abstracted embedding steps leading to increased approximation errors. Across settings, we observe that training with larger perturbation magnitudes leads to slightly worse performance on unperturbed data, but significantly improves robustness.

Table 5: Mean absolute errors for the unperturbed samples (Std. MAE) and certifiable (Cert.) ν-δ-robustness with $\nu = 0.1$ and $\delta = 0.01$.

Setting	Training Method	ϵ_t	Std. MAE [$\times$ 100]	$\epsilon = 0.05$		$\epsilon = 0.10$		$\epsilon = 0.20$	
				Adv. [%]	Cert. [%]	Adv. [%]	Cert. [%]	Adv. [%]	Cert. [%]
6h	Standard		$51.9^{\pm 0.7}$	$61.0^{\pm 5.0}$	$0.0^{\pm 0.0}$	$17.7^{\pm 2.6}$	$0.0^{\pm 0.0}$	$4.1^{\pm 1.2}$	$0.0^{\pm 0.0}$
	GAINS	0.1	$\mathbf{51.6}^{\pm 1.9}$	$96.1^{\pm 1.0}$	$90.8^{\pm 1.7}$	$70.4^{\pm 4.2}$	$50.2^{\pm 3.1}$	$28.4^{\pm 1.3}$	$16.9^{\pm 1.1}$
		0.2	$63.2^{\pm 1.6}$	$\mathbf{99.9}^{\pm 0.1}$	$\mathbf{99.9}^{\pm 0.1}$	$\mathbf{99.8}^{\pm 0.2}$	$\mathbf{99.6}^{\pm 0.4}$	$\mathbf{97.2}^{\pm 0.2}$	$\mathbf{97.0}^{\pm 0.2}$
12h	Standard		$\mathbf{48.8}^{\pm 0.2}$	$72.9^{\pm 1.3}$	$0.0^{\pm 0.0}$	$23.4^{\pm 3.6}$	$0.0^{\pm 0.0}$	$5.6^{\pm 1.0}$	$0.0^{\pm 0.0}$
	GAINS	0.1	$53.6^{\pm 2.2}$	$98.2^{\pm 0.6}$	$94.1^{\pm 0.2}$	$78.5^{\pm 2.1}$	$57.6^{\pm 4.2}$	$33.2^{\pm 4.1}$	$20.5^{\pm 2.9}$
		0.2	$54.6^{\pm 2.1}$	$\mathbf{98.8}^{\pm 0.3}$	$\mathbf{97.6}^{\pm 0.5}$	$\mathbf{88.6}^{\pm 0.3}$	$\mathbf{80.5}^{\pm 2.9}$	$\mathbf{48.9}^{\pm 1.9}$	$\mathbf{36.8}^{\pm 1.9}$
24h	Standard		$\mathbf{54.8}^{\pm 0.3}$	$83.3^{\pm 0.6}$	$0.0^{\pm 0.0}$	$38.7^{\pm 0.8}$	$0.0^{\pm 0.0}$	$10.1^{\pm 1.4}$	$0.0^{\pm 0.0}$
	GAINS	0.1	$54.9^{\pm 0.4}$	$97.8^{\pm 0.0}$	$95.3^{\pm 0.7}$	$78.6^{\pm 1.9}$	$68.3^{\pm 3.5}$	$34.2^{\pm 3.4}$	$24.2^{\pm 2.0}$
		0.2	$56.3^{\pm 0.5}$	$\mathbf{99.4}^{\pm 0.2}$	$\mathbf{99.0}^{\pm 0.4}$	$\mathbf{89.3}^{\pm 1.4}$	$\mathbf{84.9}^{\pm 1.1}$	$\mathbf{57.6}^{\pm 5.5}$	$\mathbf{47.2}^{\pm 2.9}$

D Experimental Details

All of our models is implemented in PyTorch (Paszke et al., 2017). We have used the ODE solvers from the torchdiffeq package[1] (Chen et al., 2018), which we have extended the package to contain controlled adaptive ODE solvers. Moreover, we have used the PGD adversarial attack form the torchattacks package[2] (Kim, 2020). The annealing processes of the perturbation ϵ use the implementation of the smooth scheduler from[3] Xu et al. (2020), which we denote as $\text{Smooth}(\epsilon_t, e_{start}, e_{end}, \text{mid})$. The first three arguments of the Smooth scheduler represent the target perturbation, the starting epoch of the scheduler and the epoch in which the process reaches the target perturbation. The additional mid parameter of the schedule is fixed to $\text{mid} = 0.6$ and anything else is used unaltered.

Moreover, we use the annealing process $\text{Sin}(q_{start}, q_{end}, e_1, e_2)$, for the hyperparameters $q_1,\ q_2$ occurring in the sampling process of the construction of the selection set $\mathcal{S}(\mathcal{Z})$ in App. B. The value q of the annealing process $\text{Sin}(q_{start}, q_{end}, e_1, e_2)$ in epoch e is given by

$$q \leftarrow \begin{cases} q_{start}, & \text{if } e \leq e_1, \\ \sin\left(\pi \frac{e - e_{mid}}{e_2 - e_1}\right) \cdot \frac{q_{end} - q_{start}}{2} + \frac{q_{end} + q_{start}}{2}, & \text{else if } e_1 < e \leq e_2, \\ q_{end}, & \text{otherwise,} \end{cases} \tag{9}$$

where we use $e_{mid} = \frac{e_2 - e_1}{2}$.

D.1 CAS Details

When using a CAS, we have used in all experiments update factor $\alpha = 2$, momentum factor $\beta = 0.1$, absolute error tolerance $\tau = 0.005$ and the individual ODE solver steps where performed using the dopri5(Dormand & Prince, 1980) solver. Additionally, we have introduced a minimal allowed step size constraint and maximal number of allowed rejections after clipping for the CAS, where the minimal step size is fixed to $0.02 \cdot T_{end}$ and the maximal number of allowed rejections after clipping is 2. In our experiments on the MNIST, FMNIST and PHYSIO-NET datasets the constraints only became active in early stages of training. Note that only after rejecting a step with step size h the aforementioned events can occur, in which case the solver indicates that the desired error tolerance will not be satisfied and terminates the integration by fixing the step size to h and accepting each following step without performing any step size updates anymore.

Initial Step-Size The initial step size h_0 is obtained differently in the training and testing setting. In training, a proposal initial step size $\tilde{h}_0$ is calculated using

$$\tilde{h}_0 = \begin{cases} \frac{\|\boldsymbol{z}_0\|_1}{100 * \|\boldsymbol{g}_\theta(0, \boldsymbol{z}_0)\|_1}, & \text{if } \|\boldsymbol{g}_0\|_1 \geq 10^{-5} * \gamma \text{ and } \|\boldsymbol{g}_\theta(0, \boldsymbol{z}_0)\|_1 \geq 10^{-5} * \gamma, \\ 10^{-5}, & \text{otherwise,} \end{cases} \tag{10}$$

where $\gamma = b * \tau$ is determined by the batch size b and the absolute error tolerance τ. Afterwards, a solver step is performed using the proposal step size $\tilde{h}_0$ and the step size update rule of standard

[1] https://github.com/rtqichen/torchdiffeq

[2] https://github.com/Harry24k/adversarial-attacks-pytorch

[3] https://github.com/KaidiXu/auto_LiRPA/blob/master/auto_LiRPA/eps_scheduler.py

adaptive step size solvers is used in order to produce the initial step size h_0. Note that by applying the standard update rule, the solver starts the integration process with a step size for which a step acceptance is expected. Moreover, during training the solver keeps track of an exponentially weighted average η of the initial step sizes, where it is updated using momentum factor β, i.e. $\eta \leftarrow (1-\beta)\eta + \beta * h_0$.

During testing, the current η is set as the initial step size, i.e. $h_0 = \eta$. Observe, that in NN verification the division in Eq. (10) is avoided, for which there exists only loose abstract transformations in the DEEPPOLY abstract domain. Therefore, the proposed initial step size scheme decreases the approximation error in the DEEPPOLY abstract domain at the cost of storing and keeping track of η.

D.2 CAS Comparison

In Fig. 2 we compare the reachable states, e.g. $(t,\ h)$-pairs, of the unmodifiied dopri5 (Dormand & Prince, 1980) adaptive solver (AS) and the dopri5-based CAS (as described in the previous paragraph) after at most two steps. In order to simplfy the computation of the reachable states, we have assumed that $\delta_{(t,\ h)} \in \left[2^{-6}, 2^{2}\right]\ \forall t, h$.

In Fig. 3 we compare the dopri5 AS and dopri5-based CAS with eleven different absolute error tolerances $\tau \in \{10^{-6}, 4.7{\cdot}10^{-6}, 2.2{\cdot}10^{-5}, 10^{-4}, 5{\cdot}10^{-4}, 2.3{\cdot}10^{-3}, 0.01, 0.05, 0.24, 1, 2.42\}$ on the one-dimensional nonlinear ODE $\nabla_t z = z \cdot cos\left(0.8 \cdot cos(t)^2 + t\right)$. For each absolute error tolerance value, we sample 2000 initial states $z(0) \sim \mathcal{U}(-2.5, 2.5)$ (continuous uniform distribution) and solve the resulting IVP until $T = 5$, where we report the average number of performed solver steps and the absolute error of the solver. The absolute error is calculated via $|z(5) - z_{d8}(5)|$, where $z(5)$ is the solution of either the considered AS or CAS and $z_{d8}(5)$ is the solution of the high-order adaptive solver dopri8 with absolute error tolerance $\tau_{d8} = 10^{-7}$.

E Classification Experiments

In this section, we extend the experimental details from App. D with emphasize on the classification experiments on the MNIST and FMNIST datasets.

Preprocessing We have rescaled the data in both datasets such that the values are in $[0, 1]$. Afterwards, we have standardized the data using $\mu = 0.1307,\ \sigma = 0.3081$ on the MNIST dataset and $\mu = 0.286,\ \sigma = 0.353$ on the FMNIST dataset, e.g. for input x we have $x \leftarrow \frac{x-\mu}{\sigma}$.

Neural Network Architecture In Table 6, the neural network architecture we use in classification is shown. The four arguments of the Conv2d layer in Table 6 represent the input channel, output channel, kernel size, and the stride. The two arguments of the Linear layer represents the input dimension and the output dimension. The NODE layer has $T_{end} = 1$ and ODE dynamics $\boldsymbol{g}_\theta$. Moreover, the ConcatConv2d layer takes as input a state x and time t, where it concatenates t along the channel dimension of x before applying a standard Conv2d layer. The five arguments of the ConcatConv2d layer represent the input channel, output channel, kernel size, stride and the padding.

Table 6: The neural network architecture used in classification on the MNIST and FMNIST datasets.

Classification neural network $\boldsymbol{f_\theta}$
Conv2d(1, 32, 5, 2) + ReLU
Conv2d(32, 32, 5, 2) + ReLU
NODE ($\boldsymbol{g}_\theta$, 1)
AdaptiveAvgPool2d
Linear(32,10)
ODE dynamics $\boldsymbol{g}_\theta$
[ConcatConv2d(33, 32, 3, 1, 1) + ReLU] x2

Training Details We used the ADAM (Kingma & Ba, 2015) optimizer with learning rate 1e-3 and weight decay 1e-4 as well as batch size $b = 512$ and all the training samples in training and we have used $\mathcal{L}_{std} = \mathcal{L}_{CE}$ in Eq. (5).

In provable training we have used a warm-up training session, in which we have trained the model for 50 epochs using the fixed step size ODE solver euler with $h = \frac{1}{2}$. Moreover, in the warm-up training session we have used the scheduler Smooth($\frac{1}{255}$, 10, 40) for the annealing of the perturbation ϵ.
Afterwards, in the actual training session the NODE layer uses a CAS as described in App. D. Furthermore, we train for 100 epochs using the Smooth(ϵ_t, 0, 60) schedule with $\epsilon_t \in \{0.11, 0.22\}$ on the MNIST dataset and $\epsilon_t \in \{0.11, 0.16\}$ on the FMNIST dataset. The approximation of the abstract transformer of the NODE layer uses $\kappa = 2$ in epochs 1 until 25, $\kappa = 8$ in epochs 51 until 65 and $\kappa = 4$ in all the other epochs. Moreover, we set $q_1 = q_2$ and use the annealing process $\text{Sin}(0.15, 0.33, 10, 80)$ in order to increase the value of q_1. The neural network is trained using the loss function defined in Eq. (5) with $\omega_1 = \frac{2}{3}$ and $\omega_2 = 0.01$.

In the standard training baseline we have trained the neural network for 100 epochs using the loss function defined in Eq. (5) with $\omega_1 = \omega_2 = 0$.

In the adversarial training baseline we have trained the neural network for 100 epochs, where the samples from the dataset are attacked using PGD($\epsilon, N = 10, \alpha = \frac{\epsilon}{5}, \mathcal{L}_{CE}$) prior to being fed into the model as input. Moreover, we use Smooth(ϵ_t, 5, 65) for the annealing of ϵ and $\epsilon_t = 0.11$ on both datasets. We use the loss function in Eq. (5) with $\omega_1 = \omega_2 = 0$ in training.

Furthermore, we want to emphasize that whenever we are considering abstract input regions, e.g. in provable training and adversarial training, we do not allow perturbations outside of the [0,1] interval.

Evaluation Details In order to obtain the adversarial accuracies reported in Table 4, we have used the PGD($\epsilon, N = 200, \alpha = \frac{1}{40}, \mathcal{L}_{CE}$) attack with $\epsilon \in \{0.1, 0.15, 0.2\}$ on the MNIST dataset and $\epsilon \in \{0.1, 0.15\}$ on the FMNIST dataset.

F Further Details for Time-Series Forecasting Experiments

In this section, we extend the experimental details from App. D with emphasize on the time-series forecasting task on the PHYSIO-NET dataset. Moreover, we have made use of the code provided by Rubanova et al. (2019)[4] for the fetching of the dataset and parts of the latent ODE architecture.

PHYSIO-NET Preprocessing The PHYSIO-NET dataset contains data from the first 48 hours of a patients stay in intensive care unit (ICU). The dataset consists of 41 possible features per observed measurement, where the measurements are made at irregular times and not all possible features are measured. We round up the time steps to three minutes, which results the length of the time-series to be at most $48 \cdot 20 + 1 = 961$.

Moreover, we remove four time-invariant features and additionally two categorical features from the series, namely the Gender, Age, Height, ICUType, GCS and MechVent. The removed features are inserted in an initial state $\boldsymbol{x}_0 \in \mathbb{R}^6$ of the time-series, which is used to initialize the hidden state of the encoder. Note that there is exactly one measurement for the features Gender, Age, Height and ICUType, which we used unaltered as the first four entries of the initial state $\boldsymbol{x}_0$. On the other hand, in the case where we want to predict a value in the future while only using the first L' entries of an input series, there can be multiple or no measurements for the GCS and MechVent features among the first L' entries of the series. If there are measurements made for the GCS feature, we use the average of the observed values as the fifth entry of $\boldsymbol{x}_0$, whereas if there are measurements for the MechVent feature we set the sixth entry of $\boldsymbol{x}_0$ to 1. Otherwise, if there are no measurements for the two aforementioned features their corresponding entry in $\boldsymbol{x}_0$ is set to zero.

Additionally, we clip the measurements for features with high noise or atypical values. Concretely, we clip the Temp feature to the [32,45] interval, the Urine feature to the [0,2000] interval, the WBC feature to the [0,60] interval and the pH feature to the [0,14] interval.

[4]https://github.com/YuliaRubanova/latent_ode

Furthermore, we randomly split the dataset into a training set containing 7200 time-series, validation set containing 400 time-series and testing set containing 400 time-series.

We normalize the features to be normally distributed, where we estimate the mean and standard deviation of each feature using the training set. The normalization is used for all features except the categorical features (Gender, ICUType, GCS, MechVent) and the features Fi02 and Sa02, which represent a ratio. The categorical features are used unaltered, whereas the ratios are rescaled in order to be in the [0,1] interval.

Finally, we introduce three different data modes *6h*, *12h* and *24h*, which we consider for the time-series forecasting task. The data modes differ in the number of entries L' which are used as input in order to estimate the final data point of a series. When considering the time-series $\boldsymbol{x}_{ts}^{L} = \{(\boldsymbol{x}_{(i)}, t_{(i)})\}_{i=1}^{L}$ and the data mode *6h*, the number of entries used as input is $L'_6 = \max_{i \in [L]} i$ such that $t_{(i)} \leq t_{(L)} - 6$, i.e. we try to predict at least six hours into the future. The data modes *12h* and *24h* are defined in the same way, where we try to predict at least 12 or 24 hours into the future. Furthermore, for a fixed time-series it follows that $L'_6 \geq L'_{12} \geq L'_{24}$.

Time-Series Forecasting Architecture In Table 7, we show the main components of the latent ODE architecture, which we use for the time-series forecasting task on the PHYSIO-NET dataset. In the NODE layer of the encoder $\boldsymbol{e_\theta}$ we use a one-step euler ODE solver, where the step size h depends on the measured time points in the input time-series. On the other hand, the NODE layer in the decoder $\boldsymbol{d_\theta}$ uses the CAS as specified in App. D and the final integration time depends on the time-series point we want to estimate, e.g. if we estimate $\boldsymbol{x}_{(L)}$ we use $T_{end} = t_{(L)}$.

Table 7: The main components of the latent ODE architecture used in time-series forecasting on the PHYSIO-NET dataset.

Encoder $\boldsymbol{e_\theta}$	
Linear(6,80) + ReLU GRU-Unit $\boldsymbol{f_\theta^{\text{GRU}}}$ Linear(80,100) + ReLU Linear(100,40)	
GRU-Unit $\boldsymbol{f_\theta^{\text{GRU}}}$	
f_z	NODE ($\boldsymbol{g_\theta^e}$)
f_u, f_r	Linear(115,50) + ReLU Linear(50,40) + Sigmoid
f_s	Linear(115,50) + ReLU Linear(50,80)
ODE dynamics $\boldsymbol{g_\theta^e}$	
[Linear(40,40) + ReLU] x3 Linear(40,40)	
Decoder $\boldsymbol{d_\theta}$	
NODE($\boldsymbol{g_\theta^d}$) Linear(20,35)	
ODE dynamics $\boldsymbol{g_\theta^d}$	
Linear(20,40) + ReLU [Linear(40,40) + ReLU] x2 Linear(40,20)	

Training Details We have used batch size $b = 128$ and $\mathcal{L}_{std} = \mathcal{L}_f$ in Eq. (5) with $\mathcal{L}_f$ defined in Eq. (3) and $\gamma = 30000$. Moreover, we assume that the initial state of the generative model of the time series has prior distribution $\mathcal{N}(0, 1)$. What is more, since not all feature values are observed in each measurement, we want to emphasize that only the observed features are used to evaluate

any metric. For example, if the final data point $\boldsymbol{x}_L$ has measured features at the entries in the set $M \subseteq [35]$ and we obtain the estimate $\hat{\boldsymbol{x}}_L$, the MAE is given by

$$\text{MAE}(\boldsymbol{x}_L, \hat{\boldsymbol{x}}_L) = \frac{1}{|M|} \sum_{j \in M} |x_{L,j} - \hat{x}_{L,j}|. \tag{11}$$

Additionally, as our validation metric we use the MAE with concrete inputs in all experiments in order to evalue the performance of the model on the validation set. We have trained the models on the random seeds 100, 101 and 102 [5].

Moreover, observe that in a batched input setting the sequence length of the individual time-series can be different and also the time in which measurements are made differ. In order to circumvent this issue and allow batched training, we take the union of the time points and extend each individual series to contain all time points observed in the batch, where we add data points with no measured features to each series. Furthermore, in batched training the GRU-unit of latent ODE only performs an update to the hidden state to those inputs in the batch, for which at least one feature was observed in the data point at the currently considered time.

In standard training we have trained the latent ODE for at most 120 epochs, where after each epoch we evaluate the performance of the model on the validation set and use the model with best performance on the validation set in testing. Note, that if the performance on the validation does not improve for 10 epochs we apply early stopping. Furthermore, ADAM (Kingma & Ba, 2015) was used as optimizer with learning rate 1e-3 and weight decay 1e-4 and we have used $\omega_1 = \omega_2 = 0$ in Eq. (5).

In provable training we have trained the latent ODE for 120 epochs, where we have used the scheduler Smooth(ϵ_t, 5, 65) for the perturbation with $\epsilon_t \in \{0.1, 0.2\}$. The approximation of the abstract transformer of the NODE layer in the decoder $\boldsymbol{d_\theta}$ uses $\kappa = 1$ in all epochs, whereas the NODE layer in the encoder $\boldsymbol{e_\theta}$ has due to the chosen ODE solver always only one possible trajectory. Moreover, in the NODE layer of $\boldsymbol{d_\theta}$, we set $q_1 = q_2$ and use the annealing process $\text{Sin}(0.15, 0.33, 10, 80)$ in order to increase the value of q_1. Furthermore, the abstract ratio ρ is initialized as $\rho = 0.1$ and we increase its value by 0.05 at the end of epochs $\{10, 15\}$ and by 0.1 at the end of epochs $\{10 + 5 \cdot i\}_{i=2}^{9}$. Moreover, ADAM was used as optimizer with learning rate 1e-3 and weight decay 1. Furthermore, as soon as the target perturbation is reached ($\epsilon' = \epsilon_t$), we evaluate the performance of the model on the validation set after each epoch and use the model with best performance on the validation set in verification.

Evaluation Details In order to obtain the adversarial accuracies reported in Table 1, we have used the $\text{PGD}(\epsilon, N = 200, \alpha = \frac{1}{40}, \text{MAE})$ attack with $\epsilon \in \{0.05, 0.1, 0.2\}$ on all data modes of the PHYSIO-NET dataset.

G Trajectory Attacks

In order to describe the used attacking procedure, let us denote by δ_1 the local error estimate of the solver in the first step, e.g. $\delta_1 = \delta_{(0,\ h_0)}$, and by δ_2 the local error estimate from the second step. Moreover, assume that we use a CAS with update factor α.

We describe the attack for a single δ_i with $i = 1, 2$ first and afterwards how to combine them together. The loss function $\mathcal{L}_{att}(\boldsymbol{z}_0)$ we try to maximize during the attack, depends on the value of δ_i, where in the case that $\delta_i \in [0, \tau_\alpha] \cup [\frac{\tau_\alpha + 1}{2}, 1]$, we have $\mathcal{L}_i(\boldsymbol{z}_0) = \delta_i$, whereas otherwise $\mathcal{L}_i(\boldsymbol{z}_0) = -\delta_i$ is used. Hence, we try to decrease or increase the error estimate δ_i depending on the closest decision boundary, such that a different update is performed.

The attacks are performed by using the $\{\text{PGD}(\epsilon, 100, \frac{1}{40}, \mathcal{L}_{att,m})\}_{i=-1}^{5}$ attacks with $\epsilon \in \{0.1, 0.15, 0.2\}$ and we define $\mathcal{L}_{att,m}$ next. The parameter m specifies how to combine the loss functions for the individual local error estimates δ_1 and δ_2, where for $m = -1$ we use $\mathcal{L}_{att,-1}(\boldsymbol{z}_0) = \mathcal{L}_1(\boldsymbol{z}_0)$, for $m = 0$ we use $\mathcal{L}_{att,0}(\boldsymbol{z}_0) = \mathcal{L}_1(\boldsymbol{z}_0) + \mathcal{L}_2(\boldsymbol{z}_0)$ and for $m \geq 1$ we use in PGD iteration j the loss $\mathcal{L}_{att,i}(\boldsymbol{z}_0) = \mathcal{L}_2(\boldsymbol{z}_0)$ if $j \bmod m = 0$ and otherwise $\mathcal{L}_{att,i}(\boldsymbol{z}_0) = \mathcal{L}_1(\boldsymbol{z}_0)$.

[5] GAINS with $\epsilon_t = 0.2$ in Table 1 was only trained with seed 100 and 101 due to time constraints.

In our experiments, we use the attacks with $-1 \leq m \leq 5$ for the same input $\boldsymbol{z}_0$ and as soon as we have successfully found $\boldsymbol{z}_0' \in \mathcal{B}^\epsilon(\boldsymbol{z}_0)$ such that $\Gamma(\boldsymbol{z}_0) \neq \Gamma(\boldsymbol{z}_0')$ holds, the attack is stopped and considered to be successful.

H Additional Details on the LCAP Problem

Recall the LCAP problem of soundly merging a set of different linear constraints bounding the same variable. We consider a variable y for which we have m upper bounds $\{\boldsymbol{u}^j\}_{j=1}^m$ linear in $\boldsymbol{x}$, which in turn can take values in the hyper-box region $\mathcal{Z}$. In this case, we want to obtain a single linear upper bound $y \leq \boldsymbol{a}\boldsymbol{x} + c$ that minimizes the volume between the constraint and the $y = 0$ plane over $\boldsymbol{x} \in \mathcal{Z}$, while soundly over-approximating all constraints. More formally, we want to solve:

$$\underset{\boldsymbol{a},c}{\arg\min} \int_{\mathcal{Z}} \boldsymbol{a}\boldsymbol{x} + c \,\mathrm{d}\boldsymbol{x} \qquad s.t. \quad \boldsymbol{a}\boldsymbol{x} + c \geq \max_j \boldsymbol{a}^j \boldsymbol{x} + c^j, \quad \forall \boldsymbol{x} \in \mathcal{Z}. \tag{12}$$

or equivalently

$$\underset{[\boldsymbol{a}^+,c]\in\mathbb{R}^{d+1}}{\arg\min} \boldsymbol{a}^+ \frac{\boldsymbol{l}+\boldsymbol{u}}{2} + c \qquad s.t. \quad \boldsymbol{a}^+\boldsymbol{x} + c \geq \max_i \boldsymbol{a}_i^+ \boldsymbol{x} + c_i, \quad \forall \boldsymbol{x} \in V(\mathcal{Z}) \tag{13}$$

where $V(\mathcal{Z})$ are the 2^d corners of the hyper-box $\mathcal{Z}$ and showing soundness on $V(\mathcal{Z})$ is sufficient due to the linearity of all constraints. While we can thus encode the LCAP as an LP, we have to consider exponentially many soundness constraints (one per corner), making the exact solution intractable.

In this section, we provide additional details on the experiment comparing an LP-based solution to the LCAP to our CURLS approach. In particular, we will describe the generation of the LCAP toy dataset and the LP baseline used in §5. In order to do so, we define the discrete uniform distribution $\mathcal{U}(\mathcal{X})$ over a set $X = \{\boldsymbol{x}_i\}_{i=1}^n$ and the continuous uniform distribution $\mathcal{U}(a, b)$ on a bounded domain $[a, b]$, i.e. $-\infty < a < b < \infty$. The former distribution is a categorical distribution with $p_i = \frac{1}{n}$ $\forall\, i \in [n]$, whereas the latter distribution has probability density function $p_{\mathcal{U}}(x) = \frac{1}{b-a}$ $\forall x' \in [a, b]$ and $p_{\mathcal{U}}(x) = 0$ otherwise.

LCAP Toy Dataset To generate m different linear constraints in order to describe a random relation between activation $y \in \mathbb{R}$ and activations $\boldsymbol{x} \in \mathbb{R}^d$. We only describe the process for the upper bounds of the linear constraints, since the construction of the lower bounding constraint follows analogously. Additionally, we define the cosine similarity between two vectors as $\mathrm{sim}(\boldsymbol{a}, \boldsymbol{b}) = \frac{\sum_{i=1}^d a_i \cdot b_i}{\|\boldsymbol{a}\|_2 \|\boldsymbol{a}\|_2}$ with $\|\boldsymbol{a}\|_2 = \left(\sum_{i=1}^d a_i^2\right)^{\frac{1}{2}}$. We ensure that the average cosine similarity among the produced upper bounds is within $[0.975,\ 0.99]$. The lower bound on the similarity is included, since we assume that all linear constraints describe the same relation and therefore we expect them to be similar. On the other hand, the upper bound on the similarity is included such that there are at least some difference between the constraints and the LCAP is harder to solve.

Furthermore, we define the functions $g_1(d) = 5 \cdot \left(\min\left(1, \frac{20}{d+1}\right)\right)^2$, $g_2(d) = \beta \cdot \min\left(1, \frac{5}{d+1} \cdot \left\lceil \frac{d+1}{50} \right\rceil\right)$ with $\beta = 3$ and the ceiling function $\lceil z \rceil = \min\{n \in \mathbb{N} | n \geq z\}$, and $g_{\boldsymbol{\alpha}}(\boldsymbol{x}) = \sum_{j=1}^d \alpha_j \cdot x_j + \alpha_{d+1}$ for any $\boldsymbol{\alpha} \in \mathbb{R}^{d+1}$.

First, we construct the abstract input domain $\mathcal{X}$, where for each entry x_j we sample $z_1, z_2 \sim \mathcal{U}(-g_1(d), g_1(d))$ and set $l_{x_j} = \min(z_1, z_2)$ and $u_{x_j} = \max(z_1, z_2)$.

Afterwards, we sample the coefficients $a_j \sim \mathcal{U}\left(-\frac{\beta}{2}, \frac{\beta}{2}\right)$ $\forall j \in [d+1]$ and fix the relation between $\boldsymbol{x}$ and y as $y = g_{\boldsymbol{a}}(\boldsymbol{x})$. Next, we sample the coefficients $w_j^0 \sim \mathcal{U}(-\beta, \beta)$ $\forall j \in [d+1]$ and define the proposal upper bound $g_{\boldsymbol{w}^0}(\boldsymbol{x})$. We apply an upper bounding update to the bias term if it is not a proper upper bound, i.e. $w_{d+1}^0 \leftarrow w_{d+1}^0 - \min_{\boldsymbol{x}' \in \mathcal{X}} g_{\boldsymbol{w}^0 - \boldsymbol{a}}(\boldsymbol{x}')$ if $\min_{\boldsymbol{x}' \in \mathcal{X}} g_{\boldsymbol{w}^0 - \boldsymbol{a}}(\boldsymbol{x}') < 0$. The proposal upper bound is accepted as upper bound if $|w_{d+1}^0| \leq 2 \cdot \beta$ and otherwise we repeat the procedure until we have an accepted upper bound.

Afterwards, we initialize the upper bounding set $U = \{\}$, which is iteratively augmented until its cardinality is m. In the first iteration we sample $\Delta_j^1 \sim \mathcal{U}(-g_2(d), g_2(d))$ $\forall j \in [d+1]$ and define

$\boldsymbol{w^1} = \boldsymbol{w^0} + \Delta^1$. Moreover, the bias term of $g_{\boldsymbol{w^1}}$ is corrected using the upper bounding update, such that we have $g_{\boldsymbol{w^1}}(\boldsymbol{x'}) \geq g_{\boldsymbol{a}}(\boldsymbol{x'})\ \forall \boldsymbol{x'} \in \mathcal{X}$. We include $\boldsymbol{w^1}$ to U if $|w^1_{d+1}| \leq 2 \cdot \beta$, and otherwise repeat until the iteration is accepted.
In the i-th iteration, $\boldsymbol{w^i}$ is obtained by applying the same procedure as in the first iteration. However, $\boldsymbol{w^i}$ is only included to U if $|w^i_{d+1}| \leq 2 \cdot \beta$ and $\frac{1}{|U|}\sum_{k=1}^{|U|} \mathrm{sim}(\boldsymbol{w^i}, \boldsymbol{w^k}) \geq 0.975$, otherwise we repeat the calculation of $\boldsymbol{w^i}$.

As soon as the cardinality of U equals m, we calculate the average similarity of the vectors in U and accept the set U if the similarity is less than 0.99, i.e. $\frac{1}{(m-1)\cdot(m-2)}\sum_{i=1}^{m}\sum_{k=i+1}^{m} \mathrm{sim}(\boldsymbol{w^i}, \boldsymbol{w^k}) \leq 0.99$. Otherwise, the set is rejected and we reinitialize the process from the beginning. If the set is accepted, we define the linear upper bounding constraints using $u^i = g_{\boldsymbol{w^i}}$ for $i \in [m]$.

Observe that the generation process is probabilistic and we often reject proposal coefficients and sets. Hence, in order to avoid a non-terminating process we limit the number of sampled vectors to 35000.

LP Baseline We have used LP(8, 50, 40) as a baseline for the LCAP toy dataset experiment, where for a LCAP with m different constraints that describe the relation between $z \in \mathcal{Z} \subseteq \mathbb{R}^d$ and $y \in \mathbb{R}$ the baseline works as follows. The LP baseline initially defines the set $\mathcal{Z}' = \{\boldsymbol{z}'_{(j)}\}_{j=1}^{8\cdot d}$ with $\boldsymbol{z}'_{(j)} \sim \mathcal{U}(\partial\mathcal{Z})\ \forall j \in [8 \cdot d]$, where $\partial\mathcal{Z}$ are the corners of $\mathcal{Z}$, and solves the resulting optimization problem when replacing $\mathcal{Z}$ with $\mathcal{Z}'$ in Eq. (12). We denote the optimal solution of the simplified optimization problem by $u^{\mathcal{Z}'}$. Note that due to the linear form of all the constraints, it is enough to only consider the 2^d points in $\partial\mathcal{Z}$ in the optimization constraint of Eq. (12).

Observe that since we have loosened the restrictions, we may have that $u^{\mathcal{Z}'}$ is unsound in $\partial\mathcal{Z}$, i.e. it exists some $\boldsymbol{z'} \in \partial\mathcal{Z}$ and $i \in [m]$ such that $u^{\mathcal{Z}'}(\boldsymbol{z'}) < u^i(\boldsymbol{z'})$.
If $u^{\mathcal{Z}'}$ is sound it is used as the solution of the LP baseline, otherwise for all $i \in [m]$ that violate the soundness check, we add $\hat{\boldsymbol{z}}^i = \arg\min_{\boldsymbol{z'} \in \partial\mathcal{Z}} u^{\mathcal{Z}'}(\boldsymbol{z'}) - u^i(\boldsymbol{z'})$ to the current $\mathcal{Z}'$. Moreover, for each $\hat{\boldsymbol{z}}^i$ we produce the corner points $\{\boldsymbol{z}^{i,k}\}_{k=1}^{40-1}$ and add them to $\mathcal{Z}'$ as well, where we have $z^{i,k}_j = \hat{z}^i_j$ with probability 0.75 and else $z^{i,k}_j = l_{z_j} + u_{z_j} - \hat{z}^i_j\ \forall k \in [40-1], \forall j \in [d]$.

This process is repeated at most 50 times and if the solution $u^{\mathcal{Z}'}$ is still unsound after 50 iterations, we add $\max_{i\in[m], \boldsymbol{z'} \in \partial\mathcal{Z}} u^i(\boldsymbol{z'}) - u^{\mathcal{Z}'}(\boldsymbol{z'})$ as a correction bias.