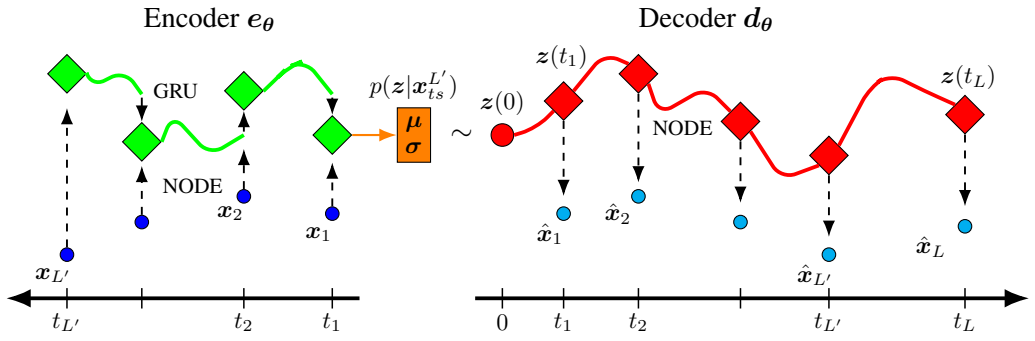


Figure 9: Visualization of the latent ODE with ODE-RNN encoder. Due to the NODE layer in the decoder the model is able to estimate the data point of the time-series at any desired time. Figure inspired by (Chen et al., 2018; Rubanova et al., 2019).

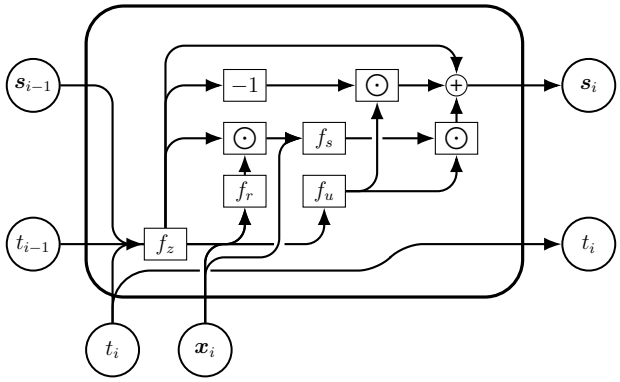


Figure 10: GRU-update for the ODE-RNN architecture, where $\odot$ denotes the hadamard product (component-wise multiplication) of two vectors and $f_z,\ f_u,\ f_r,\ f_s$ are auxiliary NNs.

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. 9. 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:

$$\mathcal{L}_f(\boldsymbol{x}_{ts}^L, L') = \gamma \cdot \|\hat{\boldsymbol{x}}_L - \boldsymbol{x}_L\|_1 - \text{ELBO}(\boldsymbol{x}_{ts}^L, L') \tag{9}$$

$$\text{ELBO}(\boldsymbol{x}_{ts}^L, L') = \mathbb{E}_{\boldsymbol{z}' \sim p_\mathcal{N}}\left[\log\left(\boldsymbol{d_\theta}\left(\boldsymbol{z}', t_L\right)\right)\right] - D_{\text{KL}}\left[p_\mathcal{N} || p\right]. \tag{10}$$

A.1 GRU update

In Fig. 10 we show the update of the hidden state $\boldsymbol{s}_{i-1}$ of the 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. We consider the setting with data distribution $(\boldsymbol{x}, y) \sim \mathcal{D}$ and we compute the NODE input $\boldsymbol{z}_0$ (either $\boldsymbol{z}_0 := \boldsymbol{x}$ or via some encoder) with the corresponding bounds $\mathcal{Z}$. Standard provable training aims to optimize a loss based on the over-approximation (Eq. (5)). However, in the case of NODE it is intractable to compute the full over-approximation of the trajectory graph (discussed in §5) for each sample in training. Thus, we only sample up to κ selected trajectories from $\mathcal{G}(\mathcal{Z})$.

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. 11.

We select trajectories as follows: We start the propagation of $\mathcal{Z}$ through the NODE layer. 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 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 $\mathcal{C}$. Afterward, we check whether or not we have reached T_{end}, where if T_{end} is reached, we save the resulting trajectory to a set $\mathcal{S}$. Moreover, 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. $|\mathcal{S}| < \kappa$.

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.

$$v' = (\tilde{t}',\ \tilde{h}') = \underset{(\hat{t},\hat{h})\in\Gamma(\boldsymbol{z}_0)}{\arg\min}\ |\tilde{t} - \hat{t}| + |\tilde{h} - \hat{h}|. \tag{11}$$

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 $g_n : \{d, a, i\} \to \{0,\ 1,\ 2\}$, 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 4.

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 $\mathcal{C}$

$$C^* = \underset{C=\{V_C\}\in\mathcal{C}}{\arg\max}\ \frac{|n(V_C) - n_{\mathcal{S}}|}{2} - |V_C| - \sigma_{out}[V_C], \tag{12}$$

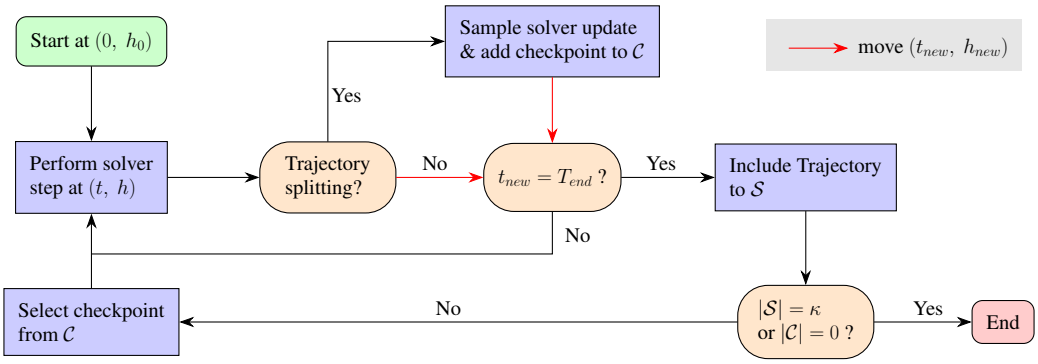


Figure 11: Selection process of $\mathcal{S}$, which contains at most κ trajectories starting with initial step size h_0 and final integration time T_{end} and the branching point set $\mathcal{C}$.

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. (12), whereas the remaining two terms capture the latter statement.

Table 4: The definition of the probabilities $p_d,\ p_a$ and p_i depending on the location index $n(V)$, reference update u' and hyperparameters $q_1,\ q_2$.

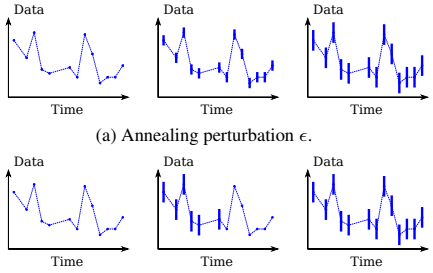
$n(V)$	u'	p_d	p_a	p_i
$n = 0$	a	$\frac{q_1+q_2}{2}$	$1 - q_1 - q_2$	$\frac{q_1+q_2}{2}$
$n = 0$ $n > 0$	d $\{d, a, i\}$	$1 - q_1 - q_2$	q_2	q_1
$n = 0$ $n < 0$	i $\{d, a, i\}$	q_1	q_2	$1 - q_1 - q_2$

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:

$$\mathcal{L}(\boldsymbol{z}_0, \mathcal{Z}, y) = (1 - \omega_1 \epsilon'/\epsilon_t)\mathcal{L}_{\text{std}}(\boldsymbol{z}_0, y) + \omega_1 \epsilon'/\epsilon_t \mathcal{L}_{\text{rob}}(\mathcal{Z}, y) + \omega_2 \|\boldsymbol{u}_{\text{out}} - \boldsymbol{l}_{\text{out}}\|_1, \tag{13}$$

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. (9).

Stabilizing Training In the time-series forecasting setting, the long integration times involving many solver calls lead to very large effective model depths. There, ϵ-annealing alone is insufficient to stabilize the training in the face of an exponential accumulation of approximation errors. To combat this, we additionally anneal the abstract ratio ρ from 0 to 1 and only use non-zero perturbation magnitudes for the first ρL data points in every time series, i.e., for an input with time index j, we set $\epsilon' \leftarrow \epsilon' \mathbb{1}_{j \leq \rho L}$. We visualize this annealing process in Fig. 12 and highlight, that it is independent of ϵ-annealing.



(a) Annealing perturbation ϵ.

(b) Annealing abstract ratio ρ.

Figure 12: ϵ-annealing (top) and ρ-annealing (bottom) for time-series input. Dots indicate concrete inputs and error bars abstract regions.

Complexity Derivation The time complexity is derived via the maximum number of vertices in the trajectory graph $\mathcal{G}(\mathcal{Z})$. Note that the graph is constructed using a CAS with update factor α that enforces a minimum step size h_{min} (described in App. C.1). The complexity does depend on h_{min} and α, but we consider both to be constant and have thus dropped the dependence. We organize the graph into rows corresponding to the step sizes and observe that for integer α each step size contains at most T_{end}/h_{min} vertices. Further, the largest possible step size is T_{end} and the smallest step size h_{min}. Due to the exponentially spaced grid of possible step sizes with growth rate α, it follows that the graph has at most $(\log(T_{end}) - log(h_{min}))/log(\alpha)$ different step sizes and hence rows. Consequently there are at most $T_{end}/h_{min}(\log(T_{end}) - \log(h_{min}))/log(\alpha)$ or after dropping the constants $\mathcal{O}(T_{end}\log(T_{end}))$ vertices in $\mathcal{G}(\mathcal{Z})$.

For the final result, note that a simple graph with v vertices has at most $v(v-1)/2$ edges. Therefore, since all edges in the trajectory graph $\mathcal{G}(\mathcal{Z})$ represent a solver step, it follows that at most $\mathcal{O}(T_{end}^2 log^2(T_{end}))$ solver steps need to be considered by GAINS.

C Experimental Details

We have used the ODE solvers from the torchdiffeq package[3] (Chen et al., 2018), where we have extended the package to contain controlled adaptive ODE solvers. Moreover, we have used the PGD adversarial attack from the torchattacks package[4] (Kim, 2020). The annealing processes of the perturbation ϵ use the implementation of the smooth scheduler from[5] Xu et al. (2020), which we denote as 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 mid $= 0.6$ and anything else is used unaltered.

Moreover, we use the annealing process 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 $\mathcal{S}$ in App. B. The value q of the annealing process Sin$(q_{start}, q_{end}, e_1, e_2)$ in epoch e is given by

$$q \leftarrow \begin{cases} q_{start}, & \text{if } e \leq e_1, \\ \sin\left(\pi \frac{e - e_{mid}}{e_2 - e_1}\right) \cdot \frac{q_{end} - q_{start}}{2} + \frac{q_{end} + q_{start}}{2}, & \text{else if } e_1 < e \leq e_2, \\ q_{end}, & \text{otherwise,} \end{cases} \tag{14}$$

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

C.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

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

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

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

the dopri5 (Dormand & Prince, 1980) solver. Additionally, we have introduced a minimal allowed step size constraint and a maximal number of allowed rejections after clipping for the CAS, where the minimum step size is fixed to $h_{min} = 0.02$ 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

$$\tilde{h}_0 = \begin{cases} \frac{\|\boldsymbol{z}_0\|_1}{100 * \|\boldsymbol{g}_\theta(0, \boldsymbol{z}_0)\|_1}, & \text{if } \|\boldsymbol{g}_0\|_1 \geq 10^{-5} * \gamma \text{ and } \|\boldsymbol{g}_\theta\left(0, \boldsymbol{z}_0\right)\|_1 \geq 10^{-5} * \gamma, \\ 10^{-5}, & \text{otherwise,} \end{cases} \tag{15}$$

where $\gamma = b * \tau$ is determined by the batch size b and the absolute error tolerance τ. Afterward, a solver step is performed using the proposal step size $\tilde{h}_0$, and the step size update rule of standard 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. (15) 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.

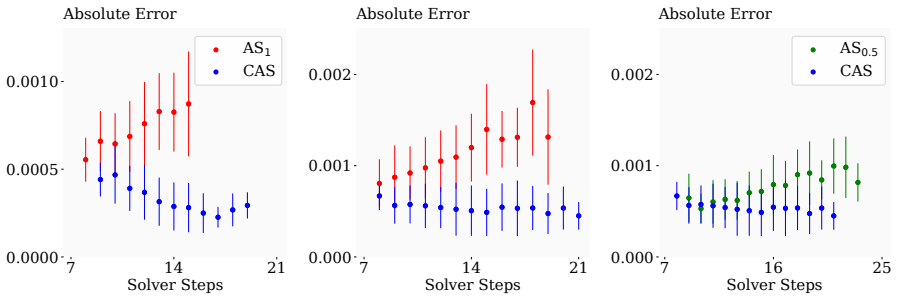
C.2 CAS COMPARISON

In Fig. 3 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. 4 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 $\tau_{d8} = 10^{-7}$.

In Fig. 13 we compare CAS and AS solvers with respect to their absolute errors depending on the number of performed solver steps for higher-dimensional, NODEs trained on the MNIST and FMNIST datasets, using standard training with the dopri5 AS solver as described in App. D. We compare dopri5-based CAS with absolute error tolerance $\tau = 0.005$ and a dopri5 AS_β with absolute error tolerance $\tau_\beta = \tau \cdot \beta$ and compute a 'ground truth' solution as reference for error computation using an AS with a 100-times smaller error tolerance, i.e. $\beta = 0.01$. We report the mean and standard deviation of the resulting absolute error $|z(1) - z_{GT}(1)|$ as a function of the number of solver steps over the first 1000 test set samples.

Using the same error tolerance for CAS and AS solvers, i.e. $\beta = 1$, we observe for both datasets, that while CAS solvers tend to perform more solver steps than AS_1, they have significantly smaller absolute errors at the same number of solver steps. We track this back to the conservative step-size update rule of CAS solvers. When decreasing the absolute error tolerance of the AS by factor 2, i.e. $\beta = 0.5$, we observe that the AS solver tends to performs more solver steps while still yielding larger absolute errors (see Fig. 13c). We thus conclude that CAS solvers are generally competitive with AS solvers.



(a) MNIST with $\beta = 1$. (b) FMNIST with $\beta = 1$. (c) FMNIST with $\beta = 0.5$.

Figure 13: Comparison of CAS and AS solvers on learned NODEs.

C.3 Bound Calculation

We introduce three different approaches to compute the bounds of a neuron, namely GAINS, GAINS-Box, and GAINS-Linear. GAINS-Box computes the bounds by only considering interval bound propagation techniques, whereas GAINS-Linear uses linear bound propagation methods (as described in §5). However, observe that when using the ReLU activation function, the selection of the slope λ of the lower bounding function (see Fig. 2) allows some design choice, because each $\lambda \in [0, 1]$ is valid (Singh et al., 2019a). GAINS-Linear selects λ such that the area between the upper and lower bound is minimized, i.e. $\lambda = 1$ if $u_x \geq -l_x$ and $\lambda = 0$ otherwise. Finally, GAINS is a combination of GAINS-Box and GAINS-Linear, where we compute the bounds for each neuron using both methods and use the tightest bounds to proceed. In order to further tighten the bounds, GAINS additionally instantiates GAINS-Linear with $\lambda = 0$ for each ReLU and GAINS-Linear with $\lambda = 1$ for each ReLU.

D Classification Experiments

In this section, we extend the experimental details from App. C 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 5, the neural network architecture we use in classification is shown. The four arguments of the Conv2d layer in Table 5 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.

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. (13).

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 used the scheduler Smooth($\frac{1}{255}$, 10, 40) for the annealing of the perturbation ϵ. Afterward, in the actual training session, the NODE layer uses a CAS as described in App. C. 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

Table 5: 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

$\text{Sin}(0.15, 0.33, 10, 80)$ in order to increase the value of q_1. The neural network is trained using the loss function defined in Eq. (13) 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. (13) 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 $\text{PGD}(\epsilon, N = 10, \alpha = \frac{\epsilon}{5}, \mathcal{L}_{CE})$ 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. (13) 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 1, we have used the $\text{PGD}(\epsilon, N = 200, \alpha = \frac{1}{40}, \mathcal{L}_{CE})$ 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.

E Further Details for Time-Series Forecasting Experiments

In this section, we extend the experimental details from App. C 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)[6] 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 in the length of the time-series being 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.

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

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.

Furthermore, we 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 6, 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. C 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 6: 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. (13) with $\mathcal{L}_f$ defined in Eq. (9) 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

$$\text{MAE}(\boldsymbol{x}_L, \hat{\boldsymbol{x}}_L) = \frac{1}{|M|} \sum_{j \in M} |x_{L,j} - \hat{x}_{L,j}|. \tag{16}$$

Additionally, as our validation metric, we use the MAE with concrete inputs in all experiments in order to evaluate the performance of the model on the validation set. We have trained the models on the random seeds 100, 101, and 102[7].

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 differs. 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 the 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. (13).

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 the best performance on the validation set in verification.

Evaluation Details In order to obtain the adversarial accuracies reported in Table 2, we have used the PGD($\epsilon, N = 200, \alpha = \frac{1}{40}$, MAE) attack with $\epsilon \in \{0.05, 0.1, 0.2\}$ on all data modes of the PHYSIO-NET dataset.

F 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 afterward how to combine them. 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

[7] Some models were trained with seed 103.

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 mod $m = 0$ and otherwise $\mathcal{L}_{att,i}(\boldsymbol{z}_0) = \mathcal{L}_1(\boldsymbol{z}_0)$.

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.

G DEEPPOLY TOY DATASET & LP BASELINE

In this section, we describe the generation of the DEEPPOLY toy dataset and the used LP baseline in the LCAP experiments in §6.3. 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 $\text{sim}(\boldsymbol{a}, \boldsymbol{b}) = \frac{\sum_{i=1}^d a_i \cdot b_i}{\|\boldsymbol{a}\|_2 \|\boldsymbol{b}\|_2}$ 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 differences 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. $w_{d+1}^0 \leftarrow w_{d+1}^0 - \min_{\boldsymbol{x'} \in \mathcal{X}} g_{\boldsymbol{w^0 - a}}(\boldsymbol{x'})$ if $\min_{\boldsymbol{x'} \in \mathcal{X}} g_{\boldsymbol{w^0 - a}}(\boldsymbol{x'}) < 0$. The proposal upper bound is accepted as the upper bound if $|w_{d+1}^0| \leq 2 \cdot \beta$ and otherwise we repeat the procedure until we have an accepted upper bound.

Afterward, 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 $|w_{d+1}^1| \leq 2 \cdot \beta$, 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 $|w_{d+1}^i| \leq 2 \cdot \beta$ and $\frac{1}{|U|} \sum_{k=1}^{|U|} \text{sim}(\boldsymbol{w^i}, \boldsymbol{w^k}) \geq 0.975$, 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. $\frac{1}{(m-1)\cdot(m-2)} \sum_{i=1}^m \sum_{k=i+1}^m \text{sim}(\boldsymbol{w^i}, \boldsymbol{w^k}) \leq 0.99$. 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. (7). We denote the optimal solution of the simplified optimization problem by $u^{\mathcal{Z}'}$, which is obtained by using a commercial linear program solver (GUROBI (Gurobi Optimization, LLC, 2022)). 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. (7).

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 $z_j^{i,k} = \hat{z}_j^i$ with probability 0.75 and else $z_j^{i,k} = l_{z_j} + u_{z_j} - \hat{z}_j^i\ \forall k \in [40-1], \forall j \in [d]$.

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.

H Additional Experiments

H.1 Comparison CAS and AS

To further compare CAS and AS solvers, we train and evaluate NODEs of the same architecture (see App. D) with either CAS or AS using both standard and adversarial training ($\epsilon_t = 0.11$). We report mean and standard deviation of the resulting standard and adversarial accuracy on MNIST and FMNIST across three runs in Table 7. We observe that while the mean performance with AS is better than that with CAS solvers in more settings than vice-versa, across both datasets and all perturbation magnitudes, there is not a single setting, where the ± 1 standard deviation ranges do not overlap. Further, we observe the same trends regardless which solver we use. We thus conclude that any performance difference between CAS and AS solvers is statistically insignificant.

Table 7: Means and standard deviations of the standard (Std.) and adversarial (Adv.) accuracy evaluated using CAS or AS on the first 1000 test set samples.

Dataset	Training Method	ODE Solver	Std. [%]	Adv. [%]		
				$\epsilon = 0.10$	$\epsilon = 0.15$	$\epsilon = 0.20$
MNIST	Standard	AS	**99.2**$^{\pm 0.1}$	**24.5**$^{\pm 2.0}$	1.9$^{\pm 0.7}$	0.0$^{\pm 0.2}$
		CAS	98.8$^{\pm 0.4}$	23.2$^{\pm 3.5}$	**2.5**$^{\pm 1.6}$	**0.3**$^{\pm 0.2}$
	Adv.	AS	**99.2**$^{\pm 0.2}$	**95.9**$^{\pm 0.2}$	**88.5**$^{\pm 0.6}$	54.6$^{\pm 2.4}$
		CAS	**99.2**$^{\pm 0.1}$	95.4$^{\pm 0.4}$	88.3$^{\pm 0.6}$	**59.4**$^{\pm 3.2}$
FMNIST	Standard	AS	**90.3**$^{\pm 0.4}$	**1.3**$^{\pm 1.6}$	**0.5**$^{\pm 0.7}$	
		CAS	88.6$^{\pm 1.2}$	0.1$^{\pm 0.1}$	0.0$^{\pm 0.0}$	
	Adv.	AS	80.8$^{\pm 0.5}$	**70.3**$^{\pm 0.3}$	**53.6**$^{\pm 3.1}$	
		CAS	**80.9**$^{\pm 0.7}$	70.2$^{\pm 0.5}$	47.1$^{\pm 3.7}$	

H.2 Comparison GAINS and TisODE

We compare our certified training via GAINS to the heuristic defence of Yan et al. (2020), which introduce time-invariant steady neural ODEs (TisODEs) using a pre-trained TisODE model from

Table 8: Comparison of GAINS-trained and TisODEs (Yan et al., 2020) with respect to standard (Std.) and adversarial (Adv.) accuracy on the first 1000 test set samples of the MNIST dataset.

Training Method	Std. [%]	Adv. [%]		
		$\epsilon = 0.10$	$\epsilon = 0.15$	$\epsilon = 0.20$
TisODE (Yan et al., 2020)	**99.3**	**93.1**	78.6	55.5
GAINS ($\epsilon_t = 0.22$)	91.8	88.5	**86.8**	**84.5**

Table 9: Means and standard deviations of the standard (Std.) and certified (Cert.) accuracy obtained using GAINS, GAINS-Linear and GAINS-Box evaluated on the first 1000 FMNIST test set samples.

ϵ_t	Std. [%]	$\epsilon = 0.10$			$\epsilon = 0.15$		
		GAINS-Box Cert. [%]	GAINS-Linear Cert. [%]	GAINS Cert. [%]	GAINS-Box Cert. [%]	GAINS-Linear Cert. [%]	GAINS Cert. [%]
0.11	$75.1^{\pm 1.2}$	$44.2^{\pm 5.5}$	$56.3^{\pm 1.4}$	$62.5^{\pm 1.1}$	$3.5^{\pm 1.4}$	$8.4^{\pm 2.3}$	$13.3^{\pm 3.1}$
0.16	$71.5^{\pm 1.7}$	$47.0^{\pm 5.7}$	$54.7^{\pm 2.5}$	$61.3^{\pm 2.7}$	$36.8^{\pm 5.2}$	$42.7^{\pm 1.4}$	$55.0^{\pm 4.3}$

Yan et al. (2020)[8] with 141 130 trainable parameters and a GAINS-trained NODE with 45 866 parameters. Reporting standard and adversarial accuracies for MNIST in Table 8, we observe that while the TisODE has a higher standard accuracy, its adversarial accuracy quickly decreases with perturbation size, falling to 55.5% at $\epsilon = 0.2$, where the GAINS-trained NODE still has 84.5% adversarial accuracy. We highlight that TisODEs are not trained with future certification in mind, explaining the gap in standard accuracy.

H.3 Ablation GAINS Verification

To analyse the effect of combining linear-bound propagation with interval bound propagation, discussed in App. C.3, we conduct two experiments: First, we compare the certified accuracies obtained with GAINS to GAINS-Linear, a version only using linear-bound propagation, and GAINS-Box, a version only using interval bound propagation (both use our trajectory graph construction). Second, we compare the bounds on output logit differences obtained with GAINS, GAINS-Linear and GAINS-Box to those obtained via an adversarial attack using PGD.

In Table 9, we report the certified accuracies obtained with GAINS, GAINS-Linear and GAINS-Box on the FMNIST dataset and observe that GAINS outperforms the other methods in every setting, showcasing that GAINS inherits benefits from both linear- and interval bound propagation. Moreover, we additionally observe that using GAINS-Linear results in higher accuracies than using GAINS-Box, demonstrating the importance of linear bound propagation and thus CURLS for our method GAINS.

In Fig. 14, we compare the tightness of the certified bounds computed with GAINS, GAINS-Linear and GAINS-Box to empirical bounds obtained via an adversarial attack on a GAINS-trained NODE ($\epsilon_t = 0.16$) for FMNIST. We illustrate both the certified over adversarial bounds (left) and the frequency of different tightness-gap sizes depending on the verification method (right) for a perturbation magnitude of $\epsilon = 0.15$. In both settings, we evaluate the first 1000 test-set images and compute the empirical bounds with a strong PGD attack using 200 steps. We clearly observe that using GAINS significantly improves bound tightness.

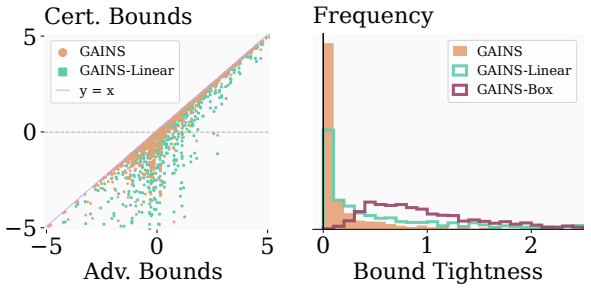


Figure 14: Comparison of certified (GAINS and GAINS-Linear) and empirical (Adv.) bounds on the worst case logit-difference (left) and illustration of the frequency of the bound tightness depending on the verification method (right).

[8] https://github.com/HanshuYAN/TisODE

Table 10: The standard (Std.), adversarial (Adv.), and certified (Cert.) accuracy obtained with GAINS evaluated on the first 1000 CIFAR-10 test set samples .

Training Method	ϵ_t	Std. [%]	$\epsilon = 0.001$	
			Adv. [%]	Cert. [%]
GAINS	0.001	60.8	57.6	57.1

H.4 SCALABILITY TO CIFAR-10

In this section, we evaluate the scalability of GAINS to the CIFAR-10 dataset(Krizhevsky et al., 2009). Training a NODE with GAINS as described below, we report standard, adversarial and certified accuracies in Table 10. We observe that for most perturbation magnitudes ($\epsilon_t = \epsilon = 0.001$), we achieve a standard accuracy of over 60% and and a certified accuracy of 57.1%, demonstrating the scalability of our approach to CIFAR-10.

Experimental Setup We modify the experimental details from App. D such that they are applicable to the CIFAR-10 dataset. We use $\boldsymbol{\mu} = [0.4914, 0.4822, 0.4465]$ and $\boldsymbol{\sigma} = [0.2023, 0.1994, 0.2010]$ for standardization.

During warm-up, we use the scheduler Smooth($\frac{0.1}{255}$, 10, 40) for ϵ-annealing. During the main training, we use $\kappa = 2$ in epochs 1-25, and $\kappa = 4$ in otherwise. For evaluation, we have used a strong PGD attack with 200 steps.

H.5 HYPERPARAMETER SELECTION

In this section, we investigate the effects of different hyperparameter selections in provable NODE training, with emphasis on the trajectory exploration and update sampling described in App. B. All experiments in this section were conducted on the FMNIST dataset using provable training with $\epsilon_t = 0.16$ and the remaining hyperparameters are as described in App. D, except when explicitly stated otherwise.

Aggregation Method As described in App. B in training we sample κ trajectories from the trajectory graph $\mathcal{G}(\mathcal{Z})$ in order to approximate the bounds of the NODE output $\boldsymbol{z}(T_{end})$. We compare three approaches on how to combine the κ trajectories in training, which we call stack, average and worst case. The stack approach considers the bounds from each sampled trajectory individually and can be interpreted as increasing the effective batchsize by factor κ, since we stack all obtained bounds along the batch dimension and propagate the resulting output through the remainder of the architecture. On the other hand, the average approach uses the mean of all obtained bounds, whereas the worst case approach uses the loosest bounds for each neuron. The results are reported in Table 11, where we see that the stack approach performs the best. We assume that this follows from the interpretation that this can be seen as increasing the effective batchsize and results in better gradient estimation. On the other hand, using the worst case approach suffers from gradient information loss, due to the usage of the maximum and minimum operations.

Table 11: Means and standard deviations of the standard (Std.) and certified (Cert.) accuracy using different aggregation methods evaluated on the first 1000 test set samples of the FMNIST dataset.

Aggregation Method	Std. [%]	Cert. [%]	
		$\epsilon = 0.10$	$\epsilon = 0.15$
stack	**71.5**$^{\pm 1.7}$	**61.2**$^{\pm 2.7}$	**54.8**$^{\pm 4.1}$
average	71.0$^{\pm 0.4}$	60.0$^{\pm 1.4}$	52.8$^{\pm 0.9}$
worst case	69.0$^{\pm 1.5}$	57.9$^{\pm 2.1}$	50.9$^{\pm 2.1}$

Annealing Process In Table 12 we evaluate the influence of the used annealing process for the sample probability $q = q_1 = q_2$ during training. We observe that when using a fixed sample probability (last two processes in Table 12), GAINS achieves higher accuracies when the sampled trajectories are closer to the reference trajectory, i.e. use smaller q. We hypothesize that the process Sin$(0.33, 0.33, 10, 80)$ considers too many trajectories which occur only due to approximation errors

in the abstract domain. However, we observe the best performance in all settings, when annealing the sampling probability. We assume that staying close to the reference trajectory in the early stages of training stabilizes the network and reduces the number of vertices in the trajectory graph induced by approximation errors. On the other hand, it is important to refine the bounds in all parts of the trajectory graph, which is why the annealing works best, if in the end we have a uniform distribution, i.e. $q \approx \frac{1}{3}$.

Table 12: Means and standard deviations of the standard (Std.) and certified (Cert.) accuracy using different annealing processes evaluated on the first 1000 test set samples of the FMNIST dataset.

Annealing Process	Std. [%]	Cert. [%]	
		$\epsilon = 0.10$	$\epsilon = 0.15$
$\mathrm{Sin}(0.15, 0.33, 10, 80)$	**71.5**$^{\pm 1.7}$	**61.2**$^{\pm 2.7}$	**54.8**$^{\pm 4.1}$
$\mathrm{Sin}(0.15, 0.4, 10, 80)$	68.1$^{\pm 2.6}$	56.4$^{\pm 4.6}$	49.6$^{\pm 5.2}$
$\mathrm{Sin}(0.15, 0.15, 10, 80)$	70.8$^{\pm 1.2}$	60.1$^{\pm 0.7}$	53.6$^{\pm 0.6}$
$\mathrm{Sin}(0.33, 0.33, 10, 80)$	68.5$^{\pm 0.7}$	57.8$^{\pm 1.8}$	50.9$^{\pm 2.9}$

Number of Sampled Trajectories In Table 13 we evaluate the influence of the number of sampled trajectories κ, where we additionally investigate the effect of including the reference trajectory among the selected trajectories (*fixed* in Table 13). We consider three κ settings, in the first one we always use $\kappa = 1$, in the second one we use $\kappa \in [2, 4, 8]$ as described in App. D, and in the last setting, we always use $\kappa = 4$. We observe that in the $\kappa = 1$ setting it is better to always use the reference trajectory instead of sampling. When increasing κ, we note that the variant which does not always include the reference trajectory performs better.

Table 13: Means and standard deviations of the standard (Std.) and certified (Cert.) accuracy using different κ evaluated on the first 1000 test set samples of the FMNIST dataset.

κ	Selection Method	Std. [%]	Cert. [%]	
			$\epsilon = 0.10$	$\epsilon = 0.15$
1	sample κ	69.0$^{\pm 1.6}$	57.8$^{\pm 3.0}$	51.0$^{\pm 3.7}$
	fixed	71.5$^{\pm 1.5}$	60.7$^{\pm 1.2}$	54.1$^{\pm 1.0}$
[2,4,8]	sample κ	71.5$^{\pm 1.7}$	61.2$^{\pm 2.7}$	**54.8**$^{\pm 4.1}$
	fixed + sample $\kappa - 1$	70.7$^{\pm 1.5}$	60.2$^{\pm 1.8}$	53.2$^{\pm 1.6}$
4	sample κ	**71.8**$^{\pm 0.9}$	**62.2**$^{\pm 1.0}$	54.7$^{\pm 1.5}$
	fixed + sample $\kappa - 1$	69.9$^{\pm 2.0}$	59.2$^{\pm 2.6}$	53.4$^{\pm 3.9}$