

Automated Formal Synthesis of Lyapunov Neural Networks

Alessandro Abate, Daniele Ahmed, Mirco Giacobbe, and Andrea Peruffo

Abstract—We propose an automated and sound technique to synthesize provably correct Lyapunov functions. We exploit a counterexample-guided approach composed of two parts: a *learner* provides candidate Lyapunov functions, and a *verifier* either guarantees the correctness of the candidate or offers counterexamples, which are used incrementally to further guide the synthesis of Lyapunov functions. Whilst the verifier employs a formal SMT solver, thus ensuring the overall soundness of the procedure, a neural network is used to learn and synthesize candidates over a domain of interest. Our approach flexibly supports neural networks of arbitrary size and depth, thus displaying interesting learning capabilities. In particular, we test our methodology over non-linear models that do not admit global polynomial Lyapunov functions, and compare the results against a cognate δ -complete approach, and against an approach based on convex (SOS) optimization. The proposed technique outperforms these alternatives, synthesizing Lyapunov functions faster and over wider spatial domains.

I. INTRODUCTION

Stability analysis determines whether a continuous dynamical system, given a domain of interest around an equilibrium point, never escapes it and, possibly, asymptotically converges towards the point. Stability properties constitute a primary objective for control engineering. Think, to make a few examples, about the cruise controller of a car, which stabilises its speed around a target value; about the controller of a self-balancing scooter, which stabilises its axis in an upright position; or about the autopilot of an airplane, which closely follows a given direction. Automatic control problems consist of the composition of a controller with a physical dynamical system, are typically modelled using differential equations, and broadly comprise issues related to stability. In this work we address the stability analysis of autonomous systems described by non-linear ordinary differential equations (ODEs), presenting a novel method for the automated and formal synthesis of Lyapunov functions.

Lyapunov functions are formal certificates of (asymptotic) stability for ODEs. Specifically, for an n -dimensional system of (possibly non-linear) ODEs

$$\dot{x} = f(x), \quad x \in \mathbb{R}^n, \quad (1)$$

having an equilibrium point at x_e , and a domain of interest $\mathcal{D} \subseteq \mathbb{R}^n$ containing x_e , a Lyapunov function is a real-valued function $V : \mathbb{R}^n \rightarrow \mathbb{R}$ such that $V(x_e) = 0$ and, for all states $x \in \mathcal{D}$ other than x_e , satisfies the two conditions

$$\dot{V}(x) = \nabla V(x) \cdot f(x) < 0, \quad V(x) > 0. \quad (2)$$

A Lyapunov function maps the system states x into energy-like values which, by the first condition, decrease over time along model's trajectories and, by the second conditions, are bounded from below. If one such function exists, then the system is asymptotically stable within \mathcal{D} .

Finding a Lyapunov function is in general a hard problem and has been the objective of numerous studies [5]. In standard literature Lyapunov functions have been constructed via analytical methods, which are mathematically sound but require substantial expertise and manual effort [10], [13], [14], [37]. Algorithmically, for linear ODEs it is sufficient to use quadratic programming, as Lyapunov functions are necessarily quadratic polynomials. However, for non-linear ODEs no general method to automatically construct Lyapunov functions exists. Numerical methods for non-linear autonomous systems include techniques that reduce the problem to solving partial differential equations (PDEs) [17], [12], partition and linearise the vector field f and then reformulate the problem as a linear programming (LP) one [9], [8], [18], [6], or restrict V to be a sum-of-squares (SOS) function and relax the synthesis problem into a Linear Matrix Inequality (LMI) program [22], [21], [16], [20]. Despite their analytical exactness, PDE-based methods rely on numerical integrators which are bound to machine precision, LP-based methods linearise f with finite accuracy, and LMI-based methods employ numerical convex optimisation—in other words, they are all numerically unsound. Conversely, we deal with constructing a Lyapunov function as a problem of formal synthesis, which is not only *automatic*, but also formally *sound*.

Formal methods for the synthesis of Lyapunov functions assume V to be given in some parametrised form, i.e., a *template*, and either relax the entire problem into a computationally tractable abstraction, or incrementally construct and check candidates through the interaction of a learner and a verifier in a counterexample-guided inductive synthesis (CEGIS) fashion [35]. Relaxation-based methods typically assume polynomial templates and reformulate the problem as a semi-algebraic one [34], [33] or as a linear program [25], [30], [31], and solve either using exact algorithms; notably, Darboux-based semi-algebraic methods can also relax polynomial templates over transcendental models [7]. Alternatively, incremental methods construct, from polynomial templates, candidates for V using linear relaxations [26], [27], [28], genetic algorithms [36], fitting simulations or, more directly, spatial samples [11], [2]; then, they verify the candidates exactly and, whenever necessary, refine the search space by learning from generated counterexamples. Notably, all methods require the user to provide a template,

A. Abate, M. Giacobbe, and A. Peruffo are with the Department of Computer Science, University of Oxford, OX1 3QD, Oxford, UK
name.surname@cs.ox.ac.uk

D. Ahmed is with Amazon Inc, London, UK.

and most techniques restrict it to a polynomial form, which in general may be insufficient [1]. We overcome these limits using, instead of fixed expressions, general templates based on neural networks.

Neural Networks are widely used in a variety of applications, such as in image classification and in natural language processing. In general, neural networks are powerful regressors, and thus lend themselves to the approximation of Lyapunov functions [15], [24]. The construction of *Lyapunov Neural Networks* (LNN) has been previously studied by approaches based on simulations and on numerical computations [32], [23], [19], [29], all of which are inherently unsound.

We introduce a method that exploits efficient machine learning algorithms, while guaranteeing formal soundness. Our method, inspired by the CEGIS architecture [2], [3], trains a candidate LNN from samples by solving an optimisation problem and attempts to falsify the candidate by solving a Satisfiability Modulo Theory (SMT) problem; upon an affirmative solution, it adds one counterexample to the samples set, re-trains the LNN, and repeats in a loop, whereas upon a negative answer (no counterexamples exist), it terminates successfully proving the soundness of the candidate. We employ a complete SMT-solver [4]: it always reports a counterexample if one exists, thus the result is provably correct. A similar CEGIS method for LNN based on δ -complete decision procedures provides weaker formal guarantees for Lagrange (practical) stability [3]; conversely, our method can guarantee full asymptotic stability at the equilibrium point, while covering wider domains of attraction. On the technical side, we employ polynomial activation functions in the NN for efficient and complete verification, and a simpler loss function, which only accounts for the $\dot{V}(x) < 0$ constraint of Eq. 2, for efficient training.

We build a prototype software and compare our method against a numerical LMI-based method (SOSTOOLS) [20], a formal template-based CEGIS method [2], and the cognate δ -complete CEGIS approach for LNN [3]. We evaluated their performance over four systems of polynomial ODEs, which are challenging as do not admit polynomial Lyapunov functions over the entire \mathbb{R}^n . We have thus measured the widest domain for which each of the methods succeeded to find a Lyapunov function. Our method attained comparable or wider domains than the other approaches, in shorter or comparable time. Notably, our method gives the strongest guarantees within the alternatives (asymptotic stability) and does not require any user hints.

Altogether, we present a synthesis method for LNN that (1) accounts for the asymptotic stability of systems of ODEs, that (2) is formal and automatic, and that (3) is faster and cover wider domains than other state-of-the-art tools.

II. COUNTEREXAMPLE-GUIDED INDUCTIVE SYNTHESIS OF LYAPUNOV NEURAL NETWORKS

We introduce a CEGIS procedure for the construction of Lyapunov functions in the form of feed-forward neural networks. We consider a network with a number n of input

neurons that corresponds with the dimension of the dynamical system, followed by k hidden layers with respectively h_1, \dots, h_k neurons, and finally followed by one output neuron. Nodes of adjacent layers are fully interconnected: a matrix $W_1 \in \mathbb{R}^{h_1 \times n}$ encompasses the weights from input to first hidden layer, a matrix $W_i \in \mathbb{R}^{h_i \times h_{i-1}}$ the weights from any other $(i-1)$ -th to i -th hidden layer, and a matrix $W_{k+1} \in \mathbb{R}^{1 \times h_k}$ the weights from k -th layer to the last neuron. Every i -th hidden layer comes with an activation function $\sigma_i: \mathbb{R} \rightarrow \mathbb{R}$, hence the output of the i -th layer can be generally written as $z_i = \sigma_i(W_i z_{i-1} + b_i)$. However, in addition to conditions in Eq. (2), a requirement as $V(x_e) = 0$ must be verified, where w.l.o.g. we consider the origin as the equilibrium point of interest. Therefore, to analytically guarantee this last condition to hold, we consider neurons with no additive bias and activations satisfying $\sigma(0) = 0$. Upon an assignment to the input neurons $x \in \mathbb{R}^n$, the neural network evaluates, layer by layer, a linear map through the matrix and the activation function (element-wise) in alternation, realizing the function

$$V(x) = W_{k+1} \sigma_k W_k \dots \sigma_2 W_2 \sigma_1 W_1 x. \quad (3)$$

Figure 1 depicts a neural network of this kind with $k = 1$, $n = h_1 = 2$, and weights as parameters w_1, \dots, w_6 .

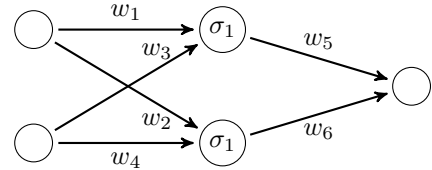


Fig. 1: A feed-forward neural network with one hidden layer.

Our procedure takes as input a n -dimensional vector field $f: \mathbb{R}^n \rightarrow \mathbb{R}^n$ (with an equilibrium assumed w.l.o.g. in the origin), a domain $\mathcal{D} \subseteq \mathbb{R}^n$, and the desired depth k and width h_1, \dots, h_k for the hidden neurons. Upon termination, the procedure returns a neural network $V: \mathbb{R}^n \rightarrow \mathbb{R}$ satisfying the conditions in Eq. (2), yielding a Lyapunov Neural Network for the asymptotic stability of f within region \mathcal{D} .

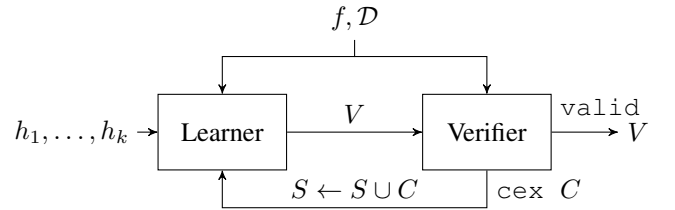


Fig. 2: CEGIS architecture for the synthesis of LNN.

The procedure, outlined in Fig. 2, consists of a learner and a formal verifier interacting in a CEGIS loop. The learner trains a candidate neural network V to satisfy the conditions in Eq. (2) over a discrete set of samples $S \subset \mathcal{D}$, which is initialised randomly. The outcome from the learner satisfies $V(0) = 0$, $\dot{V}(s) < 0$, and $V(s) > 0$ over all samples $s \in S$, but not necessarily over the entire dense domain \mathcal{D} . Thus the

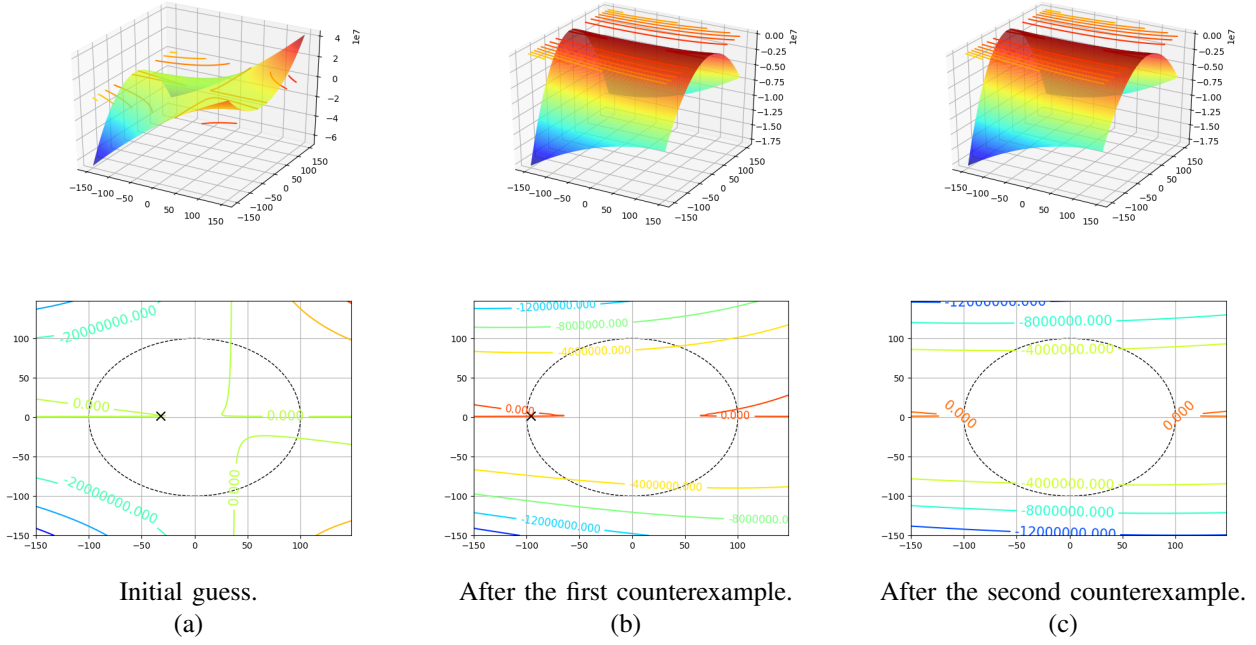


Fig. 3: The evolution of $\dot{V}(x, y)$ with the corresponding level sets through three CEGIS iterations for certifying the asymptotic stability of the system in Eq. 4 within a circle of radius $\gamma = 100$, using the neural network in Fig. 1. The synthesis loop finds two counterexamples, shown as crosses, and succeeds after three iterations.

formal verifier checks whether the resulting V violates the same conditions within the whole \mathcal{D} and, if so, produces a set of samples $C \subset \mathcal{D}$ containing one or more counterexamples c which violate either $\dot{V}(c) < 0$ or $V(c) > 0$. We add C to the samples set S , hence forcing the learner to newly produce a different candidate function, which will later be passed again to the verifier. The loop repeats indefinitely, until the verifier fails at finding a counterexample: this outcome proves that V is a LNN over the entire \mathcal{D} . We do however not guarantee termination of this procedure in general, rather we are interested in its performance in practice.

Example: We demonstrate the workflow of our procedure through the following example. Consider the planar dynamical system described by the system of polynomial ODEs

$$\begin{cases} \dot{x} = -x + xy \\ \dot{y} = -y, \end{cases} \quad (4)$$

which is asymptotically stable at the origin [1]. We aim at proving asymptotic stability within the circle of radius 100 centred at the origin, hence we take $\mathcal{D} = \{x: \|x\|_2 \leq \gamma\}$ for $\gamma = 100$. Second, we select a neural network with $k = 1$ and $h_k = 2$, as in Fig. 1, and use a quadratic polynomial as activation function $\sigma_1(x) = x^2$, hence $V(0) = 0$ is always satisfied. Moreover, we impose $w_5 = w_6 = 1$, making V positive semi-definite, i.e., $V(x, y) \geq 0$, for every remaining parameter w_1, \dots, w_4 . Therefore, we exclusively need to find a set of parameters w_1, \dots, w_4 , for which $\dot{V}(x, y) < 0$ for all $(x, y) \in \mathcal{D}$. Our CEGIS procedure computes the correct LNN after three iterations of learning and verifying the network; we show the \dot{V} at each iteration in Fig. 3. The procedure

begins by sampling a set S of random points from the domain \mathcal{D} ; then, it invokes the learner. The learner trains a network V that satisfies $\dot{V}(x, y) < 0$ over all samples in S . Specifically, the learner fixes the sample points in state space $(x, y) \in S$ and searches over the parameters space w_1, \dots, w_4 using a numerical gradient descent; the result is shown in Fig. 3a. Next, the verifier encodes \dot{V} as a first-order logic term, fixes the current instance of the parameters w_1, \dots, w_4 as constants, and computes a variables assignment $(x, y) \in \mathcal{D}$ such that $\dot{V}(x, y) > 0$, using an SMT solver. The solver produces the counterexample depicted in Fig. 3a as a cross, which indicates that \dot{V} violates the Lyapunov condition. The counterexample point is added to the samples set S and the learner retrain the parameters over the extended batch. The derivative \dot{V} of the new neural network, depicted in Fig. 3b, satisfies the negativity constraint over all initial samples plus the newly added point; yet, it violates it over a different counterexample, also depicted in Fig. 3b, as generated by the verifier, which is again added to S . The learner retrain the neural network and produces the \dot{V} of Fig. 3c. The verifier rechecks it, but this time fails at producing any counterexamples, thus proving their absence. As a consequence, the neural network satisfies both Lyapunov conditions over the entire \mathcal{D} , and the loop terminates. \square

In conclusion, the formal synthesis of LNN consists of finding an instance of parameters w such that, for all states $x \in \mathcal{D}$, the neural network satisfies the Lyapunov conditions of Eq. (2). Our CEGIS loop tackles this general problem by solving two problems interactively, the first is learning and the second a verification problem. We capitalise on the

power of neural networks for learning from data (Sec. III) and on the power of SMT-solving for verifying, and produce counterexamples accordingly (Sec. IV).

III. TRAINING OF LYAPUNOV NEURAL NETWORKS

The learner's task is to construct a candidate LNN from a discrete set of training samples $S \subset \mathcal{D}$, assuming as hyper-parameters k and h_1, \dots, h_k , the number of layers and corresponding hidden neurons. Ultimately, we expect the LNN to fulfil the conditions in Eq. (2), i.e. positivity of $V(x)$ and negativity of $\dot{V}(x)$. To ease the discussion, let us now focus solely on $\dot{V}(x)$. Our setting practically classifies the sample set into two partitions, comprising either the data points x_p such that $\dot{V}(x_p) > 0$ or the data points x_n such that $\dot{V}(x_n) < 0$. The loss function should penalise all data points in the x_p partition: we therefore select a binary classification loss function. We consider a function $L(q_1, q_2, l, \varepsilon)$ known as *Margin Ranking Loss*, that creates a criterion measuring the loss given inputs q_1, q_2 , and a label l (containing 1 or -1). If $l = 1$, the first input is ranked higher (have a larger value) than the second input, and viceversa for $l = -1$. Further, the quantity $\varepsilon > 0$ defines a threshold that guarantees the positive definiteness. Formally,

$$L(q_1, q_2, l, \varepsilon) = \max(0, -l(q_1 - q_2) + \varepsilon).$$

Recall that we aim at minimising $\dot{V}(x)$: hence we utilise $q_1 = -\dot{V}(x)$, $q_2 = \dot{V}(x)$, $l = 1$, as L becomes

$$L(-\dot{V}(x), \dot{V}(x), 1, \varepsilon) = ReLU(2\dot{V}(x) + \varepsilon),$$

where $ReLU$ represents the Rectifier Linear Unit function, where $ReLU(x) = \max(0, x)$. This approach computes the sum of data points x such that $\dot{V}(x) > -0.5\varepsilon$.

However, when $\dot{V}(x) \leq -0.5\varepsilon$ the network stops learning as the output is flat nil. Further, according to this loss function, a data point p_1 holding a very negative $\dot{V}(p_1)$ and a data point p_2 holding a negative but close to zero $\dot{V}(p_2)$ are equivalent. In order to tackle this issue, we swap the $ReLU$ function for a *Leaky ReLU* to enhance the NN learning. The Leaky ReLU $LR(a, p)$ is defined as

$$LR(a, p) = \begin{cases} p & \text{if } p \geq 0 \\ ap & \text{otherwise,} \end{cases}$$

where a is a (small) positive constant. In view of the (small) negative linear part, the network continues to learn also for $V(x) \leq -0.5\varepsilon$. Finally, the loss function results in

$$L(\dot{V}(x), \varepsilon, a) = \begin{cases} 2\dot{V}(x) + \varepsilon & \text{if } \dot{V}(x) \geq -0.5\varepsilon \\ a(2\dot{V}(x) + \varepsilon) & \text{otherwise,} \end{cases} \quad (5)$$

where a and ε are hyper-parameters defined at the beginning of the algorithm. Note that $\dot{V}(x)$ can be directly be computed from the matrices W_i , avoiding a symbolic differentiation of $V(x)$. Recall from Eq.(3) that $V(x)$ is the concatenation of terms $\sigma_i W_i x$ with derivative

$$\frac{d}{dx} \sigma_i(W_i x) = \text{diag}[\sigma'_i(W_i x)] \cdot W_i,$$

where σ'_i is the full derivative of the activation function σ_i , and $\text{diag}[v]$ represents a diagonal matrix whose entries are the elements of vector v . Let us define the auxiliary vector

$$\hat{z}_i = W_i \sigma_{i-1} W_{i-1} \dots \sigma_1 W_1.$$

The gradient $\nabla V(x)$ can be computed as

$$\nabla V(x) = W_{k+1} \text{diag}[\sigma'_k(\hat{z}_k)] \cdot W_k \text{diag}[\sigma'_{k-1}(\hat{z}_{k-1})] \dots \dots \text{diag}[\sigma'_2(\hat{z}_2)] W_2 \text{diag}[\sigma'_1(W_1 x)] W_1, \quad (6)$$

where only matrices W_i and the value x are needed. This implementation allows a fast computation of $\dot{V}(x)$ especially in the presence of polynomial activations σ_i .

IV. FINDING COUNTEREXAMPLES USING SMT-SOLVING

The aim of the verifier is to certify that the LNN received during the j -th CEGIS loop is a valid Lyapunov function. At the j -th loop, the learner offers to the verifier the weights matrices W_i , for $i = 1, \dots, k$, to compute the symbolic expression of $V(x)$ and $\dot{V}(x)$.

A valid Lyapunov function must fulfil the requirements in Eq. (2) for every x within the domain \mathcal{D} . An equivalent, yet easier, check is the search for a point $c \in \mathcal{D}$ that falsifies the Lyapunov conditions. Formally speaking, the verifier searches for a witness of the negation of Eq. (2). Let us denote by formula F the conjunction of the Lyapunov conditions, and further denote by formula d a constraint on the variables (the domain), i.e.

$$F := V(x) > 0 \wedge \dot{V}(x) < 0, \quad d := x \in \mathcal{D}. \quad (7)$$

The condition we aim to verify results in $G := d \wedge \neg F$, i.e. the verifier searches for points that invalidate F within the domain \mathcal{D} . If there exists a witness c that satisfies $\neg F$ and belongs to \mathcal{D} , the verification finds a counterexample. The point c is passed to the learner and added to the dataset S , as illustrated in Fig. 2. If such a point does not exist, the formula G is invalid and thus the candidate Lyapunov function is provably an actual Lyapunov function within the given domain, so the CEGIS procedure terminates.

Particularly for high-dimensional models, the generation of meaningful counterexamples is crucial to find a Lyapunov function quickly. Further, we expect a whole neighbourhood of c either to invalidate or to barely satisfy formula F . Thus, in order to better train the neural network and provide more training data, we randomly sample n_C data points in a neighbourhood of c . These points are then added to S and passed to the learner. In this way the point c and the corresponding region is significantly represented within S . This expedient helps to explore the whole state space and enhances the information received by the learner.

V. CASE STUDIES AND EXPERIMENTS

In this Section we provide a detailed presentation of our approach and offer a portfolio of benchmarks to validate it. Our technique is coded in Python 3.6 using the Pytorch package, whereas the implementation of the verifier is based on the SMT solver Z3 [4], which allows us to test polynomial systems and LNN with polynomial activation functions.

LNN uses sums of polynomials, which are typical as Lyapunov templates. In LNN this is achieved using polynomial activation functions that render the Lyapunov candidate a mixture of polynomials. In the experiments we use a square activation function, i.e. $\sigma(p) = p^2$, but our framework supports any polynomial. Further, the presence of hidden layers increases the order of the Lyapunov function: from Eq. (3), a single hidden layer network offers a quadratic Lyapunov function, whereas a two-layer network models a fourth-order function, formally resulting in

$$V(x) = W_2(W_1x)^2, \text{ and } V(x) = W_3(W_2(W_1x)^2)^2.$$

Recall that we consider system dynamics with (at least) one equilibrium point in the origin and impose zero bias, thus always offering a Lyapunov function satisfying $V(x_e) = 0$. Further, we set the last-layer weights to be equal to one, in order to guarantee an always positive $V(x)$, as seen in the previous equation. Note that these settings limit the LNN generality, but significantly simplify the verifier task: indeed, the verification can focus exclusively on the $\dot{V}(x) < 0$ check.

A CEGIS procedure is not guaranteed to terminate, hence in practice we set a timeout. The timeout can be set both in terms of the CEGIS iterations, namely the maximum number of candidate Lyapunov functions generated per test, and of computational time. The maximum number of CEGIS iterations is set to 100, whereas the verification time limit is set to 30 seconds. Focussing on learning, we opt for a Leaky ReLU loss function as illustrated in Section III, and we select parameter $\varepsilon = 0.01$, whereas parameter a is set to be proportional to the domain and to the system dynamics, as follows. Let us denote $x_M := \arg \max_{x \in \mathcal{D}} \|x\|^2$. We compute the value of the system dynamics $M := f(x_M)$; we then approximate this value to M_{10} , the closest power of 10, and set $a = M_{10}^{-1}$. Finally, for every counterexample C the algorithm generates $n_c = 20$ data points, which are randomly sampled in a neighbourhood around it.

In the following, we outline two test cases to highlight the flexibility of a neural network framework and to prove its effectiveness. First, we test the performance of our method varying h , the number of hidden neurons, and \mathcal{D} , the input domain. We consider the system in Eq. (4) and six spherical domains, whose radii are $\gamma_1 = 10$, $\gamma_2 = 20$, $\gamma_3 = 50$, $\gamma_4 = 100$, $\gamma_5 = 200$ and $\gamma_6 = 500$. The LNN is either composed by a single hidden layer with the number of hidden neurons h varying within set $\{2, 5, 10, 50, 100, 200\}$, or by two layers with a number of hidden neurons (h_1, h_2) within set $\{(5, 2), (5, 5), (10, 5), (50, 10), (100, 50)\}$. The outcomes of the computational times are reported in Table I.

As expected, enlarging the domain makes the search of a valid Lyapunov function harder: the verifier understandably suffers from large domains \mathcal{D} . Further, the variation of the number of hidden neurons provides interesting insights. Firstly, a single-hidden-layer fits best the synthesis of Lyapunov functions for the system under consideration: this means that a quadratic activation function is sufficiently expressive, and surely has the least computational overhead.

Secondly, our results clearly highlight a dependency between the size of the LNN and the domain diameter. As intuition suggests, a small number of hidden neurons might not provide the necessary flexibility to compute a Lyapunov function over a large domain. For this reason, utilising a multi-layer network is promising, although it must be still optimised towards learning generalisation and scalability in verification. We expect the best network configuration to be a compromise between complexity and flexibility, although finding the optimal tradeoff is still far from trivial.

We now compare our approach against a similar approach presented in [3], denoted NLC (Neural Lyapunov Control), against the approach presented in [2], denoted CBS (Constraint-Based Synthesis), and against SOSTOOLS [20]. We challenge our procedure by considering systems that do not admit a global, polynomial Lyapunov function, and as in [7] we focus on the positive octant of the state space. Therefore, we look for a Lyapunov function over domain $\mathcal{D}(\gamma) = \{x_i \geq 0, \forall i, \|x\|_2 < \gamma\}$, where γ is a predefined radius of interest. Data points close to x_e represent a numerical and analytical challenge using the NLC algorithm. Thus, as per [3], we remove from the domain a sphere around the origin, hence considering solely a disk between two concentric spheres, denoted $\mathcal{D}(\rho, \gamma)$, where ρ and γ represent the radii of inner and outer spheres, respectively. In a two-dimensional setting, this domain is a disc (annulus). A Lyapunov function valid on such a domain proves the so-called *practical* (or Lagrange) stability, which is weaker than Lyapunov asymptotic stability (as in our work). We report the best results in terms of computational time and maximum γ in Table II. We consider the system in Eq. (4), together with the following models [7]:

$$\begin{cases} \dot{x} = -x + 2x^2y \\ \dot{y} = -y; \end{cases} \quad (8)$$

$$\begin{cases} \dot{x} = -x \\ \dot{y} = -2y + 0.1xy^2 + z \\ \dot{z} = -z - 1.5y; \end{cases} \quad (9)$$

$$\begin{cases} \dot{x} = -3x - 0.1xy^3 \\ \dot{y} = -y + z \\ \dot{z} = -z. \end{cases} \quad (10)$$

Table II shows the synthesis results as a function of γ and of computation time. The NLC approach successfully synthesises Lyapunov functions for domains of radius $\gamma = 1$ but times out whenever we set a larger γ . Also the CBS struggles with the models under consideration, as the algorithm performs a linearisation that is valid only within a small domain of radius $\gamma = 1$. The LNN methodology shows faster results and synthesises over much wider domains: note that we successfully synthesise Lyapunov function with domains of radius $\gamma \geq 100$ for all the considered models. In three of these four test cases we are faster than NLC, whilst coping with a much wider domain. SOSTOOLS synthesises numerical Lyapunov functions but does not provide a sound

$\gamma \backslash h$	2	5	10	50	100	200	[5, 2]	[5, 5]	[10, 5]	[50, 10]	[100, 50]
10	0.06	0.14	0.23	1.63	1.87	11.41	0.56	1.62	2.28	3.68	9.74
20	0.14	0.67	0.21	2.99	11.85	63.03	8.86	1.34	5.64	14.32	59.28
50	0.11	2.27	1.96	7.02	21.65	110.25	121.30	21.78	3.26	82.44	158.09
100	3.68	1.90	3.03	11.46	51.63	119.40	—	—	222.12	—	—
200	48.17	23.10	53.17	30.89	165.99	301.71	—	—	—	—	—
500	—	70.65	72.09	12.01	33.91	371.65	—	—	—	—	—

TABLE I: Performance results in terms of computational time [sec] varying the number of hidden neurons h and the radius γ of the domain \mathcal{D} . Fastest outcomes for one- and two-hidden-layer are shaded in green; sign — indicates timeout.

Test Eq. #	LNN Total Time [sec]	LNN Ver. Time [sec]	LNN γ	NLC Total Time [sec]	NLC Ver. Time [sec]	NLC Domain	CBS Time [sec]	CBS Ver. Time [sec]	CBS γ	SOS Time [sec]	SOS γ
(4)	12.01	1.28	500	6.28	0.29	$\mathcal{D}(0.1, 1)$	0.22	0.08	1	6.67	800
(8)	0.29	0.08	100	5.45	0.22	$\mathcal{D}(0.1, 1)$	0.30	0.09	1	7.76	25
(9)	0.32	0.29	1000	54.12	23.70	$\mathcal{D}(0.1, 1)$	2.22	0.58	1	11.80	T/O
(10)	33.27	33.11	1000	37.80	13.45	$\mathcal{D}(0.1, 1)$	0.42	0.09	1	9.65	T/O

TABLE II: Comparison between proposed approach (LNN), CBS and NLC approaches, and SOSTOOLS: total computation time, verification time, and domain width. T/O represents when the verifier times out.

verification test. We then pass the offered $V(x)$ and $\dot{V}(x)$ to Z3 and ask to compute the validity domain. The synthesis is usually fast but SOSTOOLS generally returns Lyapunov functions with ill-conditioned coefficients: it is not rare to find terms with coefficients ranging from 10^3 to 10^{-12} . This affects the final verification step, which times out in the last two case studies. Concluding, our algorithm offers a simple, black-box approach to synthesise Lyapunov functions for tuneable domains. It outperforms existing methods as NLC by reasonably employing polynomial activation functions and using multiple hidden layers. It also shows a comparable computational time with respect to SOSTOOLS, which however does not offer any soundness guarantee.

VI. CONCLUSIONS AND FUTURE WORK

In this work we have proposed a neural network approach to automatically synthesise sound Lyapunov functions for polynomial dynamical systems. We have exploited the CEGIS framework, equipped with a sound verifier (the Z3 SMT solver) and a template-free synthesiser as a neural network. We have provided a simple, plug-and-play methodology to synthesise Lyapunov functions for polynomial systems and shown evidence of scalability and reliability of our method against benchmarks from the Lyapunov synthesis literature. Our approach promises flexibility and expressive power by increasing the number of hidden neurons, while easily providing quadratic or higher-order functions by increasing the number of hidden layers - we shall target this tradeoff in future work. Beyond improving scalability, future work also includes the implementation of non polynomial activation functions with biases, together with a procedure for an automatic selection of activation functions that are tailored to specific models.

Acknowledgments: This work is in part supported by the HiClass project (113213), a partnership between the Aerospace Technology Institute (ATI), Department for Business, Energy & Industrial Strategy (BEIS) and Innovate UK.

REFERENCES

- [1] Amir Ali Ahmadi, Miroslav Krstic, and Pablo A. Parrilo. A Globally Asymptotically Stable Polynomial Vector Field with No Polynomial Lyapunov Function. In *CDC-ECE*, pages 7579–7580. IEEE, 2011.
- [2] Daniele Ahmed, Andreas Peruffo, and Alessandro Abate. Automated and Sound Synthesis of Lyapunov Functions with SMT Solvers. In *TACAS*, 2020.
- [3] Ya-Chien Chang, Nima Roohi, and Sicun Gao. Neural Lyapunov Control. In *NeurIPS*, pages 3240–3249, 2019.
- [4] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
- [5] Peter Giesl and Sigurdur Hafstein. Review on Computational Methods for Lyapunov Functions. *Discrete and Continuous Dynamical Systems-Series B*, 20(8):2291–2331, 2015.
- [6] Peter A Giesl and Sigurdur F Hafstein. Revised CPA Method to Compute Lyapunov Functions for Nonlinear systems. *Journal of Mathematical Analysis and Applications*, 410(1):292–306, 2014.
- [7] Eric Goubault, Jacques-Henri Jourdan, Sylvie Putot, and Sriram Sankaranarayanan. Finding Non-polynomial Positive Invariants and Lyapunov Functions for Polynomial Systems through Darboux Polynomials. In *ACC*, pages 3571–3578. IEEE, 2014.
- [8] Tor A Johansen. Computation of Lyapunov Functions for Smooth Nonlinear Systems using Convex Optimization. *Automatica*, 36(11):1617–1626, 2000.
- [9] Pedro Julian, Jose Guivant, and Alfredo Desages. A Parametrization of Piecewise Linear Lyapunov Functions via Linear Programming. *International Journal of Control*, 72(7-8):702–715, 1999.
- [10] R. Kalman and J. Bertram. Control System Analysis and Design via the Second Method of Lyapunov: Part I Continuous-time Systems. *Trans. AMSE Series D J. Basic Eng.*, 82(2):371–393, 1960.
- [11] James Kapinski, Jyotirmoy V. Deshmukh, Sriram Sankaranarayanan, and Nikos Aréchiga. Simulation-guided Lyapunov Analysis for Hybrid Dynamical Systems. In *HSCC*, pages 133–142. ACM, 2014.
- [12] Edwin Kinnen and Chiou-Shiun Chen. Liapunov Functions Derived from Auxiliary Exact Differential Equations. *Automatica*, 4(4):195–204, 1968.
- [13] N. N. Krasovskii. *Stability of Motion: Applications of Lyapunov's Second Method to Differential Systems and Equations With Delay*. Stanford Univ. Press, 1963.
- [14] J. LaSalle and S. Lefschetz. *Stability by Liapunov's Direct Method With Applications*. Academic Press, 1961.
- [15] Y Long and MM Bayoumi. Feedback Stabilization: Control Lyapunov Functions Modelled by Neural Networks. In *CDC*, pages 2812–2814. IEEE, 1993.
- [16] Ian R Manchester and Jean-Jacques E Slotine. Transverse Contraction Criteria for Existence, Stability, and Robustness of a Limit Cycle. *Systems & Control Letters*, 63:32–38, 2014.

- [17] S Margolis and W Vogt. Control Engineering Applications of VI Zubov's Construction Procedure for Lyapunov Functions. *IEEE Transactions on Automatic Control*, 8(2):104–113, 1963.
- [18] Sigur F Marinósson. Lyapunov Function Construction for Ordinary Differential Equations with Linear Programming. *Dynamical Systems: An International Journal*, 17(2):137–150, 2002.
- [19] Navid Noroozi, Paknoosh Karimaghaee, Fatemeh Safaei, and Hamed Javadi. Generation of Lyapunov Functions by Neural Networks. In *World Congress on Engineering*, 2008.
- [20] Antonis Papachristodoulou, James Anderson, Giorgio Valmorbida, Stephen Prajna, Pete Seiler, and Pablo Parrilo. SOSTOOLS Version 3.03. Sum of Squares Optimization Toolbox for MATLAB, 2018.
- [21] Antonis Papachristodoulou and Stephen Prajna. On the Construction of Lyapunov Functions using the Sum of Squares Decomposition. In *CDC*, volume 3, pages 3482–3487. IEEE, 2002.
- [22] Pablo A Parrilo. *Structured Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimization*. PhD thesis, California Institute of Technology, 2000.
- [23] Vassilios Petridis and Stavros Petridis. Construction of Neural Network Based Lyapunov Functions. In *IJCNN*, pages 5059–5065. IEEE, 2006.
- [24] Danil V Prokhorov. A lyapunov machine for stability analysis of nonlinear systems. In *International Conference on Neural Networks (ICNN'94)*, volume 2, pages 1028–1031. IEEE, 1994.
- [25] Stefan Ratschan and Zhikun She. Providing a Basin of Attraction to a Target Region of Polynomial Systems by Computation of Lyapunov-Like Functions. *SIAM J. Control and Optimization*, 2010.
- [26] Hadi Ravanbakhsh and Sriram Sankaranarayanan. Counter-example guided synthesis of control lyapunov functions for switched systems. In *CDC*, pages 4232–4239. IEEE, 2015.
- [27] Hadi Ravanbakhsh and Sriram Sankaranarayanan. Robust controller synthesis of switched systems using counterexample guided framework. In *EMSOFT*, pages 8:1–8:10. ACM, 2016.
- [28] Hadi Ravanbakhsh and Sriram Sankaranarayanan. Learning Control Lyapunov Functions from Counterexamples and Demonstrations. *Autonomous Robots*, pages 1–33, 2018.
- [29] Spencer M. Richards, Felix Berkenkamp, and Andreas Krause. The Lyapunov Neural Network: Adaptive Stability Certification for Safe Learning of Dynamical Systems. In *CoRL*, volume 87 of *Proceedings of Machine Learning Research*, pages 466–476. PMLR, 2018.
- [30] Sriram Sankaranarayanan, Xin Chen, and Erika Abraham. Lyapunov Function Synthesis using Handelman Representations. *IFAC Proceedings Volumes*, 46(23):576–581, 2013.
- [31] Mohamed Amin Ben Sassi, Sriram Sankaranarayanan, Xin Chen, and Erika Abrahám. Linear Relaxations of Polynomial Positivity for Polynomial Lyapunov Function Synthesis. *IMA J. Math. Control & Information*, 33(3):723–756, 2016.
- [32] Gursel Serpen. Empirical Approximation for Lyapunov Functions with Artificial Neural Nets. In *Proceedings. International Joint Conference on Neural Networks, 2005.*, volume 2, pages 735–740. IEEE, 2005.
- [33] Zhikun She, Haoyang Li, Bai Xue, Zhiming Zheng, and Bican Xia. Discovering Polynomial Lyapunov Functions for Continuous Dynamical Systems. *Journal of Symbolic Computation*, 58:41–63, 2013.
- [34] Zhikun She, Bican Xia, Rong Xiao, and Zhiming Zheng. A Semi-algebraic Approach for Asymptotic Stability Analysis. *Nonlinear Analysis: Hybrid Systems*, 3(4):588–596, 2009.
- [35] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial Sketching for Finite Programs. *ACM Sigplan Notices*, 41(11):404–415, 2006.
- [36] Cees F. Verdier and Manuel Mazo Jr. Formal Synthesis of Analytic Controllers for Sampled-Data Systems via Genetic Programming. In *CDC*, pages 4896–4901. IEEE, 2018.
- [37] V. I. Zubov. *Methods of A. M. Lyapunov and Their Application*. Noordhoff, 1964.