ReachNN*: A Tool for Reachability Analysis of Neural-Network Controlled Systems

Jiameng Fan[1*], Chao Huang[2*], Xin Chen[3], Wenchao Li[1], and Qi Zhu[2]

[1] Boston University, Massachusetts, USA. {jmfan,wenchao}@bu.edu
[2] Northwestern University, Illinois, USA. {chao.huang,qzhu}@northwestern.edu
[3] University of Dayton, Ohio, USA. xchen4@udayton.edu

Abstract. We introduce ReachNN*, a tool for reachability analysis of neural-network controlled systems (NNCSs). The theoretical foundation of ReachNN* is the use of Bernstein polynomials to approximate any Lipschitz-continuous neural-network controller with different types of activation functions, with provable approximation error bounds. In addition, the sampling-based error bound estimation in ReachNN* is amenable to GPU-based parallel computing. For further improvement in runtime and error bound estimation, ReachNN* also features optional controller re-synthesis via a technique called *verification-aware knowledge distillation* (KD) to reduce the Lipschitz constant of the neural-network controller. Experiment results across a set of benchmarks show 7× to 422× efficiency improvement over the previous prototype. Moreover, KD enables proof of reachability of NNCSs whose verification results were previously unknown due to large overapproximation errors. An open-source implementation of ReachNN* is available at https://github.com/JmfanBU/ReachNNStar.git.

Keywords: Neural-network controlled systems · Reachability · Bernstein polynomials · GPU acceleration · Knowledge distillation.

1 Introduction

There has been a growing interest in using neural networks as controllers in areas of control and robotics, e.g., deep reinforcement learning [13], imitation learning [14,7], and model predictive control (MPC) approximating [3,9]. We consider neural-network controlled systems (NNCSs) that are closed-loop sampled-data systems where a neural-network controller controls a continuous physical plant in a periodic manner. Given a sampling period $\delta > 0$, the neural-network (NN) controller reads the state x of the plant at the time $t = i\delta$ for $i = 0, 1, 2, \ldots$, feeds it to a neural network to obtain the output u, and updates the control input in the plant's dynamics to u. Our tool ReachNN* aims to solve the following reachability problem of NNCSs.

[*] Jiameng Fan and Chao Huang contributed equally.

We acknowledge the support from NSF grants 1646497, 1834701, 1834324, 1839511, 1724341, ONR grant N00014-19-1-2496, and the US Air Force Research Laboratory (AFRL) under contract number FA8650-16-C-2642.

Problem 1. The *reach-avoid problem* of a NNCS is to decide whether from any state in an initial set X_0, the system can reach a target set X_f, while avoiding an unsafe set X_u within the time interval $[0, T]$.

A major challenge facing reachability analysis for NNCSs is the presence of non-linearity in the NN controllers. Existing reachability analysis tools for NNCSs typically target specific classes of NN controllers [5,12,15,2]. Sherlock [5] and NNV [15] for instance only consider neural networks with RELU activation functions, while Verisig [12] requires the neural networks to have differentiable activation functions such as tanh/Sigmoid.

In this paper, we present our tool ReachNN*, which is a significantly extended implementation of our previous prototype ReachNN [11]. ReachNN* provides two main features. First, it can verify an NNCS with any activation functions by Bernstein polynomial approximation [11]. Second, based on the proportionality relationship between approximation error estimation Lipschitz constant of the NN controller, ReachNN* can use knowledge distillation (KD) [10] to retrain a verification-friendly NN controller that preserves the performance of the original network but has a smaller Lipschitz constant, as proposed in [6].

Another significant improvement in ReachNN* is the acceleration of the sampling-based error analysis in ReachNN by using GPU-based parallel computing. The sampling-based approach uniformly samples the input space for a given sample density and evaluates the neural network controller and the polynomial approximation at those sample points. We use the Lipschitz constant of the neural network and the samples to establish an upper bound on the true error (details in [11]). For networks with many inputs, this approach may require many sample points to avoid a blowup in the overapproximation. Here, we make the observation that the sampling-evaluation step is a single instruction, multiple data (SIMD) computation which is amenable to GPU-based acceleration. Experimental results across a set of benchmarks show 7× to 422× efficiency improvement over the previous prototype.

2 Tool Design

The architecture of ReachNN* is shown in Fig. 1. The input consists of three parts: (1) a file containing the plant dynamics and the (bounded) reach-avoid specification, (2) a file describing the NN controller, and (3) an optional target Lipschitz constant for controller retraining. The tool then works as follows. For every sampling step $[i\delta, (i+1)\delta]$ for $i = 0, 1, 2 \ldots$, a polynomial approximation along with a guaranteed error bound for the NN controller output is computed and then used to update the plant's continuous dynamics. The evolution of the plant is approximated by flowpipes using Flow*. During the flowpipe construction, it checks every computed flowpipe whether it lies entirely inside the target set X_f and outside the avoid set X_u. The tool terminates when (1) the reachable set at some time $t \leq T$ lies inside the target set and all computed flowpipes do not intersect with the avoid set, i.e. the reach-avoid specification is satisfied; or

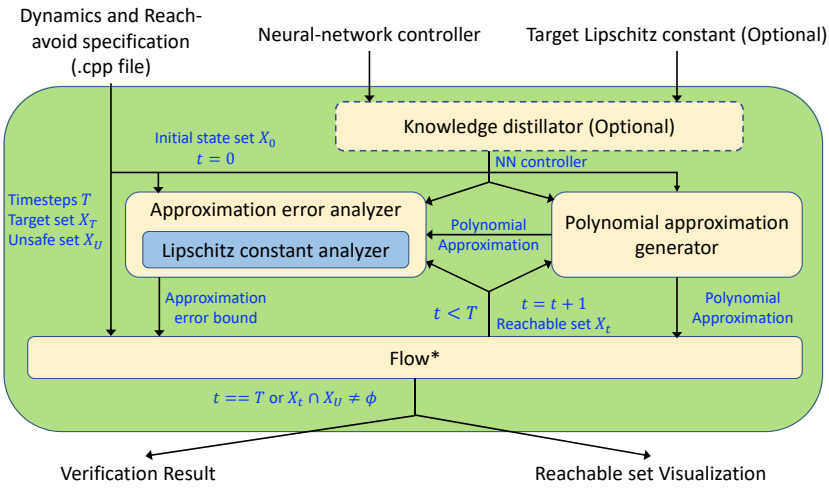


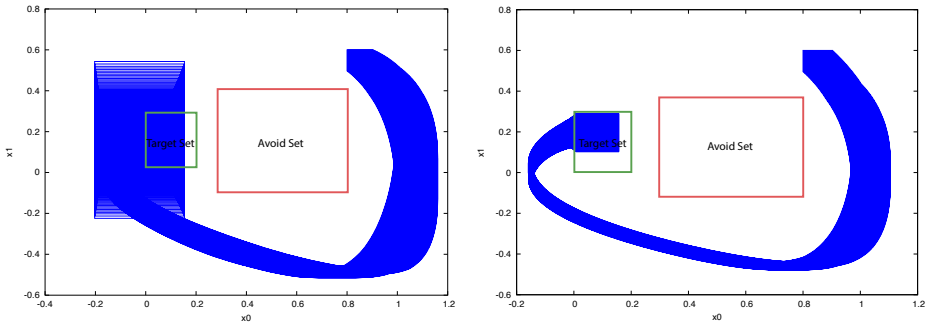
Fig. 1: Structure of ReachNN*.

(2) an unsafe flowpipe is detected, i.e. it enters the avoid set X_u; or (3) the reachable set at some time t intersects with but is not entirely contained in X_f, in which case the verification result is unknown. The tool also terminates if Flow* fails due to a blowup in the size of the flowpipes. Along with the verification result (Yes, No or Unknown), the tool generates a Gnuplot script for producing the visualization of the computed flowpipes relative to X_0, X_f and X_u.

When the tool returns Unknown, it is often caused by a large overapproximation of the reachable set. As noted before, the overapproximation error is directly tied to the Lipschitz constant of the network in our tool. In such cases, the user can enable the knowledge distillation option to retrain a new neural network. The retrained network has similar performance compared to the original network but a smaller Lipschitz constant. The tool will then perform reachability analysis on the retrained network. We describe the function of each model in ReachNN* in more detail below.

[**Polynomial approximation generator**] We implement this module in Python. It generates the approximation function of a given neural network over a general hyper-rectangle, with respect to a given order bound for the Bernstein polynomials. The generated polynomial is represented as a list of monomials' orders and the associated coefficients.

[**Approximation error analyzer**] This module is implemented in Python. It first invokes a sub-module – Lipschitz constant analyzer, to compute a Lipschitz constant of the neural network using a layer-by-layer analysis (see Section 3.2 of [11] for details). Then, given the Lipschitz constant, this module estimates the approximation error between a given polynomial and a given neural network by uniformly sampling over the input space. To achieve a given precision, this



(a) Before KD (12718 seconds) (b) After KD (103 seconds)

Fig. 2: Reachability analysis results: Red lines represent boundaries of the obstacles and form the avoid set. Green rectangle represents the target region. Blue rectangle represents the computed flowpipes.

sampling-based error estimation may result in a large number of samples. In ReachNN*, we leverage Tensorflow [1] to parallelize this step using GPUs.

[Flow*] We use the C++ APIs in Flow* [4] to carry out the following tasks: (a) flowpipe construction under continuous dynamics using symbolic remainders, (b) checking whether a flowpipe intersects the given avoid set, (c) checking whether a flowpipe lies entirely in the given target set, and (d) generating a visualization file for the flowpipes.

[Knowledge distillator] This module is implemented in Python with GPU support for retraining. The inputs for this module are the original NN, a target Lipschitz constant number, and a user-specified tolerance of the training error between the new network and the original network. The output is a retrained network. Details of the distillation procedure can be found in [6]. We note that this module also supports distilling the original network into a new network with a different architecture, which can be specified as an additional input.

Example 1. Consider the following nonlinear control system [8]: $\dot{x}_0 = x_1, \dot{x}_1 = ux_1^2 - x_0$, where u is computed from a NN controller κ that has two hidden layers, twenty neurons in each layer, and ReLU and tanh as activation functions. Given a control stepsize $\delta_c = 0.2$, we want to check if the system will reach $[0, 0.2] \times [0.05, 0.3]$ from the initial set $[0.8, 0.9] \times [0.5, 0.6]$ while avoiding $[0.3, 0.8] \times [-0.1, 0.4]$ over the time interval $[0, 7]$.

The verification finished in 12718 seconds and the result is Unknown, which indicates the computed flowpipes intersect with (and are not contained entirely in) the avoid set or the target set. The flowpipes are shown in Fig. 2a. With KD enabled, we retrain a new NN controller with the same architecture, a target Lipschitz constant as 0 (0 means the knowledge distillator will try to minimize the Lipschitz constant) and a regression error tolerance of 0.4. The resulting

Table 1: Comparison with ReachNN. We use l to represent the number of layers in the neural network controller, n to represent the number of neurons in the hidden layers, and $\bar{\varepsilon}$ for the error bound in sampling-based analysis. We use the same benchmarks from [11]. The dimensions of states are from 2 to 4 for these benchmarks. Time shows the runtime of the reachability analysis module. The *After KD* results do not include the runtime for knowledge distillation. The average runtime for knowledge distillation is 245 seconds [4]. *Acc* (short for acceleration) denotes the ratio between the runtime of ReachNN and that of ReachNN* on the same NNCS without applying knowledge distillation.

#	NN Controller			Setting	Verification Result		Time (Seconds)			*Acc*
	Act	l	n	$\bar{\varepsilon}$	Before KD	After KD	ReachNN [11]	ReachNN*	After KD	
1	ReLU	3	20	0.001	Yes(35)	–	3184	26	–	112×
	sigmoid	3	20	0.005	Yes(35)	–	779	76	–	10×
	tanh	3	20	0.005	Unknown(35)	Yes(35)	543	76	70	7×
	ReLU+tanh	3	20	0.005	Yes(35)	–	589	76	–	7×
2	ReLU	3	20	0.01	Yes(9)	–	128	5	–	25×
	sigmoid	3	20	0.001	Yes(9)	–	280	13	–	21×
	tanh	3	20	0.01	Unknown(7)	Yes(7)	642	71	69	9×
	ReLU+tanh	3	20	0.01	Yes(7)	–	543	25	–	21×
6	ReLU	4	20	0.01	Yes(10)	Yes(10)	7842	1126	12	7×
	sigmoid	4	20	0.01	No(7)	–	32499	77	–	422×
	tanh	4	20	0.01	No(7)	–	3683	11	–	334×
	ReLU+tanh	4	20	0.01	Yes(10)	Yes(10)	10032	1410	674	7×

flowpipes are shown in Fig. 2b. We can see that the new NN controller can be verified to satisfy the reach-avoid specification. In addition, the verification for the new NN controller is 123× faster compared to verifying the original NNCS.

3 Experiments

We provide a full comparison between ReachNN* and the prototype ReachNN on all the examples in [11]. If the verification result is Unknown, we apply our verification-aware knowledge distillation framework to synthesize a new NN controller and check the resulting system with ReachNN*. All experiments are performed on a desktop with 12-core 3.60 GHz Intel Core i7 and NVIDIA GeForce RTX 2060 (ReachNN does not make use of GPU).

We highlight part of the results for Benchmark #1, #2 and #6 in Table 1 due to space constraint (results on all benchmarks can be found in `https://github.com/JmfanBU/ReachNNStar.git`). ReachNN* achieves from 7× to 422×

[4] The runtime of the knowledge distillation module does not vary much across different benchmarks.

efficiency improvement on the same NNCSs (across all benchmarks also). In Benchmark #1 and #2 with Unknown results, we applied our knowledge distillation procedure to obtain new NN controllers and performed reachability analysis again on the resulting systems. Observe that ReachNN* produces a Yes answer for these systems. In addition, it took a shorter amount of time to compute the verification results compared to ReachNN. In Benchmark #6, ReachNN* took more than 1000 seconds to obtain a Yes result in two cases. We run knowledge distillation for these two cases to evaluate whether KD can be beneficial solely from an efficiency perspective. In both cases, ReachNN* with KD significantly improves runtime compared to ReachNN* without KD.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: OSDI. pp. 265–283 (2016)
2. Bogomolov, S., Forets, M., Frehse, G., Potomkin, K., Schilling, C.: Juliareach: a toolbox for set-based reachability. In: HSCC. pp. 39–44 (2019)
3. Chen, S., Saulnier, K., Atanasov, N., Lee, D.D., Kumar, V., Pappas, G.J., Morari, M.: Approximating explicit model predictive control using constrained neural networks. In: ACC. pp. 1520–1527. IEEE (2018)
4. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Flow*: An analyzer for non-linear hybrid systems. In: CAV. pp. 258–263. Springer (2013)
5. Dutta, S., Chen, X., Sankaranarayanan, S.: Reachability analysis for neural feedback systems using regressive polynomial rule inference. In: HSCC. pp. 157–168 (2019)
6. Fan, J., Huang, C., Li, W., Chen, X., Zhu, Q.: Towards verification-aware knowledge distillation for neural-network controlled systems. In: ICCAD. IEEE (2019)
7. Finn, C., Yu, T., Zhang, T., Abbeel, P., Levine, S.: One-shot visual imitation learning via meta-learning. In: Conference on Robot Learning. pp. 357–368 (2017)
8. Gallestey, E., Hokayem, P.: Lecture notes in nonlinear systems and control (2019)
9. Hertneck, M., Köhler, J., Trimpe, S., Allgöwer, F.: Learning an approximate model predictive controller with guarantees. IEEE Control Systems Letters **2**(3), 543–548 (2018)
10. Hinton, G.E., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. CoRR **abs/1503.02531** (2015)
11. Huang, C., Fan, J., Li, W., Chen, X., Zhu, Q.: Reachnn: Reachability analysis of neural-network controlled systems. TECS **18**(5s), 1–22 (2019)
12. Ivanov, R., Weimer, J., Alur, R., Pappas, G.J., Lee, I.: Verisig: verifying safety properties of hybrid systems with neural network controllers. In: HSCC. pp. 169–178 (2019)
13. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. International Conference on Learning Representation (2016)
14. Pan, Y., Cheng, C.A., Saigol, K., Lee, K., Yan, X., Theodorou, E., Boots, B.: Agile autonomous driving using end-to-end deep imitation learning. RSS (2018)
15. Tran, H.D., Cai, F., Diego, M.L., Musau, P., Johnson, T.T., Koutsoukos, X.: Safety verification of cyber-physical systems with reinforcement learning control. TECS **18**(5s), 1–22 (2019)

A Appendix

We test our tool on the benchmarks that have the same dynamics as the benchmarks proposed in Sherlock, but with different settings of neural-network (NN) controllers. The main difference lies on the NN controller, where the NN controller we use may have different activation functions simultaneously, e.g. we have ReLU+sigmoid NN controller for Ex. #1. The input dimension of the NN controller, which is the dimension of the system state, ranges from 2 to 4. Each NN controller has either 3 or 4 hidden layers, and width (the number of neurons) of each hidden layer ranges from 20 to 100. The detailed setting can be found in [11] and `https://github.com/JmfanBU/ReachNNStar`.

In addition, a navigation control benchmark with Dubins car model and various neural-network controller can be found in [6] and `https://github.com/JmfanBU/NNCS-Dubins-Car`. The goal is to navigate the vehicle through a corridor. The controller drives the vehicle to turn at the first corner and avoids the obstacle in the middle of the corridor. The state input is 3 and the controller output is the steering as a scalar. NN controller has 2 hidden layers and each hidden layer has 20 neurons. Different NN controllers have different activation functions. We did the simulation in Matlab for all the benchmarks.

The main difference of our benchmarks are the heterogeneous architecture of our NN controllers that have different activation functions, which is common in practice. Such a setting can effectively test the generality of verification techniques.