A Reachability Method for Verifying Dynamical Systems with Deep Neural Network Controllers

Kyle D. Julian and Mykel J. Kochenderfer
Department of Aeronautics and Astronautics, Stanford University, Stanford, CA 94305
{kjulian3, mykel}@stanford.edu

Abstract
Deep neural networks can be trained to be efficient and effective controllers for dynamical systems; however, the mechanics of deep neural networks are complex and difficult to guarantee. This work presents a general approach for providing guarantees for deep neural network controllers over multiple time steps using a combination of reachability methods and open source neural network verification tools. By bounding the system dynamics and neural network outputs, the set of reachable states can be over-approximated to provide a guarantee that the system will never reach states outside the set. The method is demonstrated on the mountain car problem as well as an aircraft collision avoidance problem. Results show that this approach can provide neural network guarantees given a bounded dynamic model.

1 Introduction
Neural networks are global function approximators that can efficiently and accurately represent any target domain. In recent years, neural networks have become widely used for decision making and control in dynamical systems such as pendulums [Riedmiller, 2005], video games [Mnih et al., 2015], autonomous vehicles [Bojarski et al., 2016], and aircraft collision avoidance systems [Julian et al., 2016]. Neural network controllers can be generated in many ways, including supervised learning to compress existing controllers [Julian and Kochenderfer, 2017], Neural Fitted Q Iteration [Riedmiller, 2005], and deep Q-learning [Mnih et al., 2015].

In some domains, neural network controllers can make decisions better than human experts, but guarantees about their performance are difficult to prove, inhibiting their use in safety-critical systems. Whereas traditional model checking techniques may be sufficient to verify logic-based controllers [McMillan, 1993], new verification techniques are required to ensure that complex and non-linear neural network controllers will always generate safe trajectories.

Recent advancements in neural network verification tools have yielded many options for proving bounds on neural network outputs [Liu et al., 2019]. Given some region of the input space, these verification tools define what values could be given by the neural network and provide either a guarantee of a correct output or a counterexample. Some tools work layer by layer through the network to compute a reachable set of each layer, ending with the reachable output of the network given the input region [Gehr et al., 2018; Xiang et al., 2017]. Other tools formulate neural network verification as a mixed-integer linear programming and use optimization methods to find counterexamples [Lomuscio and Maganti, 2017; Tjeng et al., 2017]. Tools such as Reluplex [Katz et al., 2017] and Sherlock [Dutta et al., 2018] convert neural networks into linear programs and systematically search for a satisfying counterexample. While these tools ensure that input regions satisfy some network output property for a single evaluation, ensuring safety for multiple network evaluations over time remains difficult due to system dynamics and uncertainties.

Recent work involved developing methods for the verification of neural network controllers for agents acting in an environment. These methods use a reachability approach that computes the set of states the system can reach given an initial set of states and a controller. Akintunde et al. [2018] formulate neural network controllers as a hybrid system to compute reachability sets over time. Xiang et al. [2019] train neural networks to output the next state in a nonlinear system instead of a control input. They compute over-approximations of the reachable set at each time step to bound the trajectories of state variables.

This work builds upon this prior work and formulates a reachability algorithm for an agent controlled by a neural network. Whereas previous approaches combine the neural network controller output with a dynamics model, we separate the two components and analyze each individually. Our method is applicable to any network architecture or activation function, and we can use any existing neural network verification tool. Furthermore, by analyzing the dynamics separately, our method can consider complex actions that are not well represented by linear models. Our work also considers the degree of uncertainty that can be present in the system dynamics before safety can no longer be guaranteed, allowing us to...
make stronger claims about how a real system with errors will be able to maintain safety. A mountain car example is provided to illustrate the approach on a classic problem, and an additional aircraft collision avoidance problem demonstrates how the approach can handle more complex systems.

2 Approach
This section describes the reachability method used to provide guarantees about neural network controlled systems. First, the neural network is over-approximated in order to provide guarantees about neural network controlled systems. This section describes the reachability method used to provide guarantees considering some level of error in the dynamics. This work considers these errors to obtain safety guarantees considering some level of error in \( g(s, a) \). One way to incorporate these errors is to expand \( R_c \) to include all states that could be reached with error in the dynamics.

\[ R_c = \{ g(s, a) : s \in S, a \in A_N \} \]

In general, there may be uncertainty surrounding \( g(s, a) \). Factors such as modeling errors, disturbances, or human error may cause the dynamics of real systems to differ from ideal equations. This work considers these errors to obtain safety guarantees considering some level of error in \( g(s, a) \). One way to incorporate these errors is to expand \( R_c \) to include all states that could be reached with error in the dynamics.

Algorithm 1 Reachability for networks with discrete actions

**Input:** \( R_0, S_N, T \)

**Output:** \( \{R_1, R_2, \ldots, R_T\} \)

1: for \( t = 1 \) to \( T \) 
2: \( R_t \leftarrow \emptyset \)
3: for \( S \in R_{t-1} \) 
4: Compute \( A_S \) using neural network verification tool
5: for \( a \in A_S \) 
6: Compute \( R_s \) using state dynamics
7: for \( S' \in S_N \) 
8: if \( S' \cap R_s \neq \emptyset \)
9: \( R_t \leftarrow R_t \cup S' \)
10: return \( \{R_1, R_2, \ldots, R_T\} \)

Next, the set of cells that could be reached from cell \( S_i \in R_t \) is calculated as

\[ R_{t+1} = \{ S : S \in S_N, S \cap R_{S_i} \neq \emptyset \} \]  \hspace{1cm} (2)

Equation (2) over-approximates \( R_S \) by including all cells that have some intersection with \( R_S \). Lastly, the set of cells reachable at the next time step from any cell in \( R_t \) is

\[ R_{t+1} = \bigcup_{s_i \in R_t} R_{S_i} \]  \hspace{1cm} (3)

Algorithm 1 outlines the steps for this method.

Because the neural network and system dynamics are over-approximated, this method will over-approximate the reachable set at each time step. As a result, all states outside the reachable set at time \( t \) are impossible to reach from the initial set of states. If the reachable set is contained within a goal set or excludes a failure set, then this method verifies that the neural network controller will be safe at time \( t \).

2.3 Implementation Considerations
Although the proposed method can verify neural network controllers, its implementation may raise a few concerns. First, neural network verification tools can be slow, so checking each discrete cell may seem slow especially as the number of cells grows. However, many neural network verification tools run faster as input bounds become tighter, so checking many small regions may not be slower than checking fewer large regions. Furthermore, computing \( R_S \) becomes more complicated with higher-dimensional state spaces. In two dimensions with linear dynamics and rectangular cells, computing \( R_S \) is straightforward, but more complicated dynamic models for higher-dimensional spaces can prove challenging. Fortunately many tools exist to perform reachability over-approximations for complex problems, such as SpaceEx, which uses lazy set computation with Zonotopes for efficient computation with hundreds of state variables [Frehse et al., 2011]. Another tool, JuliaReach, uses the high performance Julia language to compute reachable sets assuming convex sets and currently supports linear dynamics [Bogomolov et al., 2019]. These tools provide efficient methods for reachable set estimation that can be used to enable this approach for complex neural network systems.
The goal of the car is to reach a crete Markov decision process using POMDPs.jl [Egorov et al., 2017]. To generate a neural network controller, though there are many other methods that could be used with this reachability approach. This section presents a method for generating neural network controllers, as described in Section 2.1. The state space was divided into 341523 input cells with smaller cells around the cells. However, making cells smaller can quickly lead to an explosion in the number of cells to consider, especially at higher dimensions. It is important to reduce cell size in critical areas of the state space without generating so many cells that reachability computation becomes too slow.

3 Mountain Car Example

To demonstrate the approach, we consider the classic mountain car problem in which a one-dimensional car must climb to the top of a hill [Moore, 1991]. However, the car’s acceleration cannot overcome gravity, so the car must rock between two hills to gain enough momentum to reach the top of the hill. The system has two state variables, position $p$ and velocity $v$, and one control input, $u$. We consider the same dynamic model as used by Ivanov et al. [2018] but with disturbance $\delta_u \sim \mathcal{U}(-w, w)$ added to the control input, where $w$ is the maximum disturbance. The state variables are updated as

$$
[p, v] \leftarrow \left[ v + 0.0015(u + \delta_u) - 0.0025 \cos(3p) \right] \tag{4}
$$

This formulation uses discrete actions, $u \in \{-1, 0, 1\}$, instead of continuous values. The range of positions and velocities is $p \in [-1.2, 0.6]$ and $v \in [-0.07, 0.07]$ respectively, and the goal of the car is to reach $p = 0.6$ as quickly as possible.

3.1 Neural Network Controller

This section presents a method for generating neural network controllers, though there are many other methods that could be used with this reachability approach. To generate a neural network controller, the problem was first solved as a discrete Markov decision process using POMDPs.jl [Egorov et al., 2017]. In a Markov decision process, an agent in state $s \in S$ takes action $a \in A$, receives reward $r(s, a)$, and transitions to a new state $s'$ with probability $T(s, a, s')$, where $T$ is the transition function [Kochenderfer, 2015b]. Discrete value iteration was used to compute policy $\pi$ that maximizes the accumulation of reward overtime. In discrete value iteration, $Q$-values are associated with each state-action pair, with initial $Q_0(s, a) = 0$. The $Q$-values are updated according to the finite-horizon Bellman equation as

$$
Q_{t+1}(s, a) = r(s, a) + \max_{a'} \sum_s T(s, a, s')Q_t(s', a') \tag{5}
$$

After the $Q$-values converge, the policy is computed as $\pi(s) = \arg \max_a Q(s, a)$. For the mountain car problem, $s = (p, v)$, $S$ is the Cartesian product of 100 uniformly distributed positions and velocities, $a = u$, $r(s, a) = -1(p < 0.6)$, and $T(s', a, s)$ has $1/3$ probability for each of the next states calculated with $\delta_u$ as -0.5, 0.0, and 0.5, which encourages the computed policy to be robust to noisy accelerations. Multilinear interpolation is used to compute the $Q(s', a')$ when $s'$ does not fall on one of the points in the grid.

After computing the $Q$-values for the optimal policy, a neural network is trained through supervised learning to approximate the $Q$-values. The neural network has an input variable for each state dimension and an output variable associated with each action. An asymmetric loss function that penalizes under-valuing the optimal action or over-valuing the suboptimal actions was used, which encourages the network to approximate both the $Q$-values and policy well [Julian et al., 2016]. For the mountain car problem, the neural network uses ReLU activations and has five hidden layers of 30 units each. The neural network was trained for 1000 epochs using Adamax optimization [Kingma and Ba, 2015], and the trained network predicts actions with 97.80% accuracy and an average absolute error of 5.69. The policy of the trained network is shown in Figure 1, which shows that the neural network does not recommend $u = 0$ actions often if at all.

3.2 Neural Network Approximation

Three methods were used to approximate the neural network policy as described in Section 2.1. The state space was divided into 341523 input cells with smaller cells around $v = 0$ to reduce over-approximation errors. The actions the network could give within each cell were determined using three methods: 100 random samples, ReluVal [Wang et al., 2018], and Reluplex [Katz et al., 2017]. Table 1 reports the results where the number of actions is the sum of the number of actions found in all cells, and the run time reported uses a single CPU, though all methods are easily parallelized to decrease runtime. The sampling method misses an action in one of the cells, and Reluplex finds three spurious actions that occur very close to the cell but outside the cell boundaries. Decreasing the boundary error tolerance removes these spurious actions. ReluVal identifies all actions given within each cell and performs faster than both Reluplex and sampling approaches.

| Method   | Num. Actions | Runtime (s) | Complete |
|----------|--------------|-------------|----------|
| Sampling | 344126       | 172.9       | No       |
| ReluVal  | 344127       | 72.99       | Yes      |
| Reluplex | 344130       | 57360       | Yes      |

Figure 1: Mountain car neural network controller policy

Table 1: Mountain car neural network approximation results
3.3 Reachability Results

Figure 2 shows an example of the reachability process for a cell \( S \). With \( w = 0.5 \) and \( u = 1.0 \), \( R_S \) is computed by considering the extreme accelerations at the four corner points of \( S \) and forming a polygon from the resulting next states. Then, \( R^S \) is derived as the set of cells that overlap with \( R_S \).

Initially, the car could be in any state, so \( \mathcal{R}_0 = S_X \). As seen in Figure 3, over time the reachable set diminishes to include only states where \( p = 0.6 \), which proves that the car will always reach the goal when using the neural network controller regardless of initial condition.

Reachable sets were also computed with different levels of dynamic disturbance to study how errors in the dynamics effect verification. The reachability method was compared to two Monte Carlo simulations that use worst-case and random disturbances. The worst-case disturbance is \( da = -\text{sign}(u)w \) so that the disturbance always opposes the car's acceleration, while random disturbances are sampled from \( \mathcal{U}(-w, w) \) at each time step. For both Monte Carlo methods, 10000 simulations from random starting locations were evaluated, and the maximum number of time steps to reach the goal state was recorded. Figure 4 summarizes the results and shows that the reachability method guarantees that the car will reach the goal up to disturbance level \( w = 0.25 \); however, worst-case Monte Carlo simulations show that the car will reach the goal for \( w \leq 0.5 \). Over-approximation errors contribute to the size of the gap between the two approaches, and decreasing the cell sizes to reduce over-approximation error could decrease this gap. Random Monte Carlo simulations show no difference for different levels of disturbances. Since the disturbance distribution has zero mean, the disturbance has no effect in expectation. Verifying safety in critical systems means ensuring failures do not occur even with low probability, which requires firm bounds on dynamic uncertainties to prevent large disturbances from having non-zero probability.

4 Aircraft Collision Avoidance System

Example

The second example considers commercial aircraft, which are required to operate with a collision avoidance system that gives vertical climbrate advisories to pilots to prevent near midair collisions (NMACs). An NMAC occurs when the aircraft are separated by less than 100 ft vertically and 500 ft horizontally. An aircraft collision avoidance system called ACAS X uses \( Q \)-values to represent much of the decision making logic [Kochenderfer, 2015a]. It uses a large table of state-action \( Q \)-values to make decisions, but Julian et al. [2016] explored compressing the table using a neural network. This section presents a notional example based loosely on an early prototype of ACAS X [Kochenderfer, 2015a].

4.1 System Description

The example collision avoidance system, referred to as VerticalCAS, considers two aircraft: an ownship equipped with VerticalCAS, and an intruder aircraft. In this formulation, the intruder is assumed to maintain level flight, but future work can relax this assumption. The system uses four variables to describe the encounter with the intruder aircraft:

1. \( h \) (ft): Intruder altitude relative to own [−3000, 3000]
2. \( \dot{h}_0 \) (ft/min): Ownship vertical climbrate [−2500, 2500]
3. \( \tau \) (s): Time to loss of horizontal separation [0, 40]
4. \( s_{\text{adv}} \): Previous advisory from VerticalCAS

The first two state variables describe the encounter geometry vertically. The \( \tau \) variable condenses the horizontal geometry into a single variable by providing a countdown until the intruder will no longer be separated horizontally, at which point the ownship must be vertically separated to avoid an NMAC. The \( s_{\text{adv}} \) variable is categorical and can be any one of the nine possible advisories given by the system, and conditioning the next advisory on the current advisory allows the system to maintain consistency when alerting pilots. The nine possible advisories are

1. COC: Clear of Conflict
2. DNC: Do Not Climb
3. DND: Do Not Descent
4. DES1500: Descend at least 1500 ft/min
5. CL1500: Climb at least 1500 ft/min
6. SDES1500: Strengthen Descent to at least 1500 ft/min
7. SCL1500: Strengthen Climb to at least 1500 ft/min
8. SDES2500: Strengthen Descent to at least 2500 ft/min
9. SCL2500: Strengthen Climb to at least 2500 ft/min

Each advisory instructs the pilot to accelerate until complying with the specified climb or descent rate, except for COC, which allows the pilot freedom to choose any acceleration \( \dot{h}_0 \in [-g/8, g/8] \), where \( g \) is the sea-level gravitational acceleration constant, 32.2 ft/s\(^2\). For advisories DNC, DND, DES1500, and CL1500, the pilot is assumed to accelerate in the range \( |\dot{a}| \in [g/4, g/3] \) with the sign of \( \dot{h}_0 \) determined by the specific advisory. If the pilot is already compliant with the given advisory, then the pilot is assumed to continue at the current climbrate. For advisories SDES1500, SCL1500, SDES2500, and SCL2500, the pilot is assumed to accelerate at \( \pm g/3 \) until compliance. For example, a pilot receiving the CL1500 advisory while descending at −500 ft/min is
that the system tends to strengthen the previous advisory in order to avoid collisions. However, if the intruder is well below the aircraft, then descending could result in an NMAC, so the system reverses the advisory and strengthens to SCL1500.

4.2 Reachability Results

To define $S_N$, we first note that $s_{adv}$ is discrete with nine values, and $\tau$ acts independently and can be discretized to $\{0, 1, \ldots, 40\}$. The remaining variables, $h$ and $\dot{h}_0$, are continuous and need to be discretized to define cells. Because we are interested in the NMAC region where $|h| \leq 100$ ft, low magnitude $h$ values are more finely discretized to limit over-approximation errors in that region of the state space. The final discretization used 231 points for $h$ and 44 for $\dot{h}_0$, resulting in 3.65 million discrete cells. Reluplex was used to approximate the neural networks, generating 32.8 million queries in order to check for each of the nine advisories in all discrete cells. Reluplex took 175 CPU-hours to complete, an order of magnitude longer than for the mountain car example.

After approximating the neural networks, the reachable sets can be computed. The reachable region $R_S$ was over-approximated by computing the minimum and maximum $h$ and $\dot{h}_0$ values achievable from cell $s$ given the acceleration constraints. The initial reachable set is all cells at $\tau = 0$ s with $s_{adv} = COC$. Each new reachable set counts down $\tau$ until eventually $\tau = 0$ s and the aircraft must be separated.
by 100 ft vertically to avoid an NMAC. As seen in Figure 6, the neural network controller causes the ownship to climb or descend in order to avoid the unsafe NMAC set. Therefore, this method guarantees that no collision will occur when using the neural network collision avoidance system, given that assumptions made to constrain the acceleration hold. If the acceleration limits are relaxed such that the pilot can accelerate up to \( g/2 \), for example, then the reachable set will overlap with the NMAC region, and safety cannot be guaranteed.

### 4.3 Pilot Delay

Assuming that a human pilot will react immediately to collision avoidance advisories from the neural network system is unreasonable. In reality, there will be some amount of pilot delay when following the advisories. For maximum pilot delay \( \tau \), the pilot could choose to follow any of the advisories of the previous \( \epsilon \) seconds. Pilot delay can be incorporated into the reachable set formulation by tracking the \( \epsilon \) most recent advisories that result in reaching each cell. A union of cells reachable using all possible delayed advisories is used to compute \( R_{\tau+1} \) instead of using the current advisory.

For a pilot delay of three seconds, safety is no longer guaranteed, as seen in the left plot of Figure 7. For example, consider an ownship slightly below the intruder. VerticalCAS would alert the ownship to descend to avoid an NMAC, but with pilot delay, the ownship could climb for \( \epsilon \) seconds instead. At this point, the ownship could be above the intruder aircraft in a region where VerticalCAS would reverse the advisory to make the ownship continue climbing and gain further separation. However, with pilot delay, the pilot could begin following the stale descend advisories and descend instead. The cycle repeats indefinitely, resulting in an NMAC.

One solution to fix such issues uses online costs to penalize undesirable behavior [Kochenderfer, 2015a]. For example, an online cost could be applied to prevent the system from reversing the direction of given advisories more than once. A count of reversals can be tracked along with the previous advisories given to reach each cell. As shown in Figure 7, preventing double reversals is sufficient to prevent an NMAC. This reachability method allows complex systems involving neural network controllers alongside other variables and penalties to be analyzed to ensure safety.

### 5 Conclusions

Guaranteeing the performance of neural network controllers in the presence of uncertainty is paramount to incorporating such controllers in safety critical systems. We presented a general method that uses existing neural network verification tools to constrain the neural network output before computing over-approximated reachable sets of system states. If the over-approximated set does not contain any unsafe states, then the system is guaranteed to be safe. A mountain car example demonstrated the approach and provided guarantees on time to reach the goal for different levels of disturbances. A collision avoidance example demonstrated how complex systems involving human delay and online costs could be modeled and shown to be safe. The proposed method is flexible and applicable to any neural network controller of a dynamical system. Future work will explore higher dimensional problems along with open source reachable set libraries in order to decrease computation time.

### Acknowledgments

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE—1656518. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.
References

[Akintunde et al., 2018] Michael Akintunde, Alessio Lomuscio, Lalit Maganti, and Edoardo Provenzo. Reachability analysis for neural agent-environment systems. In International Conference on Principles of Knowledge Representation and Reasoning, 2018.

[Bogomolov et al., 2019] Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Kostiantyn Potomkin, and Christian Schilling. JuliaReach: a toolbox for set-based reachability. arXiv preprint arXiv:1901.10736, 2019.

[Bojarski et al., 2016] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. arXiv preprint arXiv:1604.07316, 2016.

[Dutta et al., 2018] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. Output range analysis for deep feedforward neural networks. In NASA Formal Methods Symposium, pages 121–138. Springer, 2018.

[Egorov et al., 2017] Maxim Egorov, Zachary N. Sunberg, Edward Balaban, Tim A. Wheeler, Jayesh K. Gupta, and Mykel J. Kochenderfer. POMDPs.jl: A framework for sequential decision making under uncertainty. Journal of Machine Learning, 18(26):1–5, 2017.

[Frehse et al., 2011] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In International Conference on Computer Aided Verification, pages 379–395. Springer, 2011.

[Gehr et al., 2018] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Rajarshi Ray, and Martin Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In Symposium on Security and Privacy (SP), pages 3–18. IEEE, 2018.

[Ivanov et al., 2018] Radoslav Ivanov, James Weimer, Rajeev Alur, George J Pappas, and Insup Lee. Verisig: verifying safety properties of hybrid systems with neural network controllers. arXiv preprint arXiv:1811.01828, 2018.

[Julian and Kochenderfer, 2017] Kyle D Julian and Mykel J Kochenderfer. Neural network guidance for UAVs. In AIAA Guidance, Navigation, and Control Conference (GNC), page 1743, 2017.

[Julian et al., 2016] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. Policy compression for aircraft collision avoidance systems. In Digital Avionics Systems Conference (DASC), 2016.

[Katz et al., 2017] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In International Conference on Computer Aided Verification, pages 97–117. Springer, 2017.

[Kingma and Ba, 2015] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In International Conference on Learning Representations (ICLR), 2015.

[Kochenderfer, 2015a] Mykel J. Kochenderfer. Optimized airborne collision avoidance. In Decision Making under Uncertainty: Theory and Application, chapter 10, pages 249–273. MIT Press, 2015.

[Kochenderfer, 2015b] Mykel J. Kochenderfer. Sequential problems. In Decision Making under Uncertainty: Theory and Application, chapter 4, pages 77–112. MIT Press, 2015.

[Liu et al., 2019] Changliu Liu, Tomer Arnon, Christopher Lazarus, Clark Barrett, and Mykel J Kochenderfer. Algorithms for verifying deep neural networks. arXiv preprint arXiv:1903.06758, 2019.

[Lomuscio and Maganti, 2017] Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward ReLU neural networks. arXiv preprint arXiv:1706.07351, 2017.

[McMillan, 1993] Kenneth L McMillan. Symbolic model checking. In Symbolic Model Checking, pages 25–60. Springer, 1993.

[Mnih et al., 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. Nature, 518(7540):529–533, 2015.

[Moore, 1991] Andrew Moore. Efficient Memory-based Learning for Robot Control. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, March 1991.

[Riedmiller, 2005] Martin Riedmiller. Neural fitted Q iteration – first experiences with a data efficient neural reinforcement learning method. In European Conference on Machine Learning (ECML), pages 317–328. Springer, 2005.

[Tjeng et al., 2017] Vincent Tjeng, Kai Xiao, and Russ Tedrake. Evaluating robustness of neural networks with mixed integer programming. arXiv preprint arXiv:1711.07356, 2017.

[Wang et al., 2018] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In Security Symposium (USENIX Security 18), pages 1599–1614, 2018.

[Xiang et al., 2017] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. Reachable set computation and safety verification for neural networks with ReLU activations. arXiv preprint arXiv:1712.08163, 2017.

[Xiang et al., 2019] Weiming Xiang, Diego Manzanas Lopez, Patrick Musau, and Taylor T Johnson. Reachable set estimation and verification for neural network models of nonlinear dynamic systems. In Safe, Autonomous and Intelligent Vehicles, pages 123–144. Springer, 2019.