Validation of Image-Based Neural Network Controllers through Adaptive Stress Testing

Kyle D. Julian[1], Ritchie Lee[2], and Mykel J. Kochenderfer[1]

***Abstract*— Neural networks have become state-of-the-art for computer vision problems because of their ability to efficiently model complex functions from large amounts of data. While neural networks can be shown to perform well empirically for a variety of tasks, their performance is difficult to guarantee. Neural network verification tools have been developed that can certify robustness with respect to a given input image; however, for neural network systems used in closed-loop controllers, robustness with respect to individual images does not address multi-step properties of the neural network controller and its environment. Furthermore, neural network systems interacting in the physical world and using natural images are operating in a black-box environment, making formal verification intractable. This work combines the adaptive stress testing (AST) framework with neural network verification tools to search for the most likely sequence of image disturbances that cause the neural network controlled system to reach a failure. An autonomous aircraft taxi application is presented, and results show that the AST method finds failures with more likely image disturbances than baseline methods. Further analysis of AST results revealed an explainable cause of the failure, giving insight into the problematic scenarios that should be addressed.**

I. Introduction

Many autonomous systems interact in complex environments and operate with high-dimensional data, such as images. Recent work has shown that neural networks can be trained efficiently to make decisions for image-based problems. Mnih *et al.* use deep reinforcement learning to train neural network controllers that map Atari screen images to controller commands to create game-playing agents that outperform humans [1]. Additional work has shown that neural networks can play chess [2], classify objects [3], and recognize digits [4]. In addition, neural networks can be used to control vehicles, such as steering cars [5], [6], guiding aircraft to waypoints [7], and controlling quadrotors [8].

Although misclassifying a cat image is undesirable, steering vehicles off roads or into other vehicles can be catastrophic. Recently, tools have been developed that can verify input-output properties of neural networks, such as those representing aircraft collision avoidance policies [9]. These tools use the simplex method [9], [10], mixed integer linear programming [11], symbolic interval analysis with linear relaxation [12], and other approaches [13].

However, verifying input-output properties of systems acting as closed-loop controllers is insufficient to verify safety. Additional work has focused on verification of multi-step properties of neural network controllers acting within their environment [14]–[17]. Existing closed-loop verification work is applicable when the network input is low-dimensional and adequate environmental models exist; however, when the network input is high-dimensional and the environment is complex, verification approaches are intractable. As a result, many image-based neural network verification approaches focus on local robustness around validation images [18], [19]. However, local robustness is an input-output property and does not address closed-loop safety.

This work focuses on validation of image-based neural network controllers. Existing work on validation of complex systems has led to the development of adaptive stress testing (AST), which uses reinforcement learning to find the most likely ways systems fail [20], [21]. However, existing work with AST has only considered low-dimensional problems.

Our method combines ideas from local robustness verification with AST black box validation to efficiently search for sequences of image disturbances that lead to failure. Using neural network verification tools allows the algorithm to search for multi-step sequences in a lower-dimensional space than the size of the image. As a result, this method scales well to neural networks with hundreds of input variables without making any assumptions about the environment, allowing the tool to be easily integrated with any existing simulator. An example aircraft taxiway application is presented that uses the X-Plane 11 photo-realistic flight simulator. The method is able to find sequences of image disturbances that cause the neural network to guide the aircraft off the taxiway, and further analysis reveals explainable weaknesses of the neural network that were exploited to cause failures.

II. Background

This work combines ideas from reinforcement learning and neural network verification, which are described in this section.

A. Markov Decision Process

A Markov decision process (MDP) is a general framework for modeling sequential decision-making problems and is described by the tuple $(\mathcal{S}, \mathcal{A}, R, T)$ [22]. An agent in state

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1656518, AFRL and DARPA under contract FA8750-18-C-0099, and a NASA Ames Research Center summer internship. Any opinions, findings, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Government.

[1]Kyle D. Julian and Mykel J. Kochenderfer are with the Department of Aeronautics and Astronautics, Stanford University, Stanford, CA 94305, USA {kjulian3, mykel}@stanford.edu

[2]Ritchie Lee is with the Robust Software Engineering group at NASA Ames Research Center, Moffett Field, CA 94035, USA ritchie.lee@nasa.gov

$s \in \mathcal{S}$ takes action $a \in A$, transitions to s' with probability $T(s' \mid s, a)$, and receives reward $r = R(s, a, s')$. The action-value function $Q^\pi(s, a)$ gives the expected value of taking action a from state s and following the policy $a = \pi(s)$ for all future states, computed as

$$Q^\pi(s,a) = \mathbb{E}\left[\sum_{t=0} \gamma^t r_t \mid s_0 = s, a_0 = a, a' = \pi(s)\right], \quad (1)$$

where discount factor γ is set to $\gamma = 1$ for finite horizon problems and $0 < \gamma < 1$ for infinite horizon problems. The goal with an MDP to compute $\pi(s)$ that maximizes $Q(s, a)$, denoted as $\pi^*(s)$ and $Q^*(s, a)$ respectively.

If the transition function is known, then optimization algorithms can compute $\pi^*(s)$; however, when the transition function is unknown or difficult to model, such as with image-based navigation, alternative methods are needed. Reinforcement learning is a model-free method that uses simulations to estimate $Q^*(s, a)$. The following two subsections describe reinforcement learning algorithms used in this work.

B. Monte Carlo Tree Search

Monte Carlo tree search (MCTS) begins with a root node with initial state, s_0. New states s' are added to the tree as leaf nodes from an existing node s via the action a used to arrive at s' from s. At each node, the algorithm keeps track of $N(s)$, the number of times the node with state s has been visited, $N(s, a)$, the number of times action $a \in A(s)$ has been taken from s, and $Q(s, a)$, the value estimate of taking action a from state s. MCTS follows three basic steps:

1) *Search*. Beginning at the root node with state s and actions $A(s)$ already added to the tree, new actions are added to the tree if $\|A(s)\| < kN(s)^\alpha$, where k and α are hyperparameters that balance exploration with exploitation. If a new action is not added, then an existing action is taken to maximize

$$Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \quad (2)$$

where c is also a hyperparameter. The search process repeats from s' until a new action is selected.
2) *Expansion*. The tree is expanded with the new action by simulating the action to compute s', which is added as a new leaf node to the tree with an empty $A(s')$. This work assumes that transitions are deterministic, although other versions of MCTS can incorporate stochastic actions [20].
3) *Rollout*. Once a new state s' has been added to the tree, a random rollout simulation is used to initialize $Q(s, a)$. The rollout uses a default policy $\pi_0(s)$ to determine actions taken, and the rollout continues until a pre-determined depth or terminal state is reached. $Q(s, a)$ is computed from the rollout simulation using Eq. (1) with $\gamma = 1$ since the rollout has a finite depth. The estimated $Q(s, a)$ is then propagated up through all parent nodes according to

$$N(s,a) \leftarrow N(s,a) + 1 \quad (3)$$

$$q \leftarrow R(s,a,s') + \gamma Q(s',a') \quad (4)$$

$$Q(s,a) \leftarrow Q(s,a) + \frac{q - Q(s,a)}{N(s,a)} \quad (5)$$

where $Q(s', a')$ is the updated value of the child node. Once the rollout finishes, MCTS begins another iteration by searching from the root node until a new state is added. For further details on the MCTS algorithm, see [20].

C. Deep Q-Learning

While MCTS can simulate many trajectories to optimize $\pi(s)$, the algorithm does not generalize from the values of states already seen to predict values of new states. Another popular algorithm, deep Q-learning, or deep Q-networks (DQN), maintains a global functional representation of $Q(s, a)$ [1]. As a result, updating $Q(s, a)$ for one state updates the values for nearby states.

DQN uses a deep neural network to approximate $Q(s, a)$, and the network parameters are updated through gradient descent methods using a loss value based on the temporal difference between $r + \gamma Q(s', a')$ and $Q(s, a)$. This work used the OpenAI Baselines implementation of DQN [23] with prioritized experience replay [23], [24].

D. Neural Network Verification

Recent advancements have produced tools that verify neural network input-output properties [9], [10], [12]. For neural networks of the form $y = f(x)$, where f is composed of computational layers with piecewise-linear activations, these properties are defined as $x \in \mathcal{X} \implies f(x) \notin \mathcal{Y}$, where $\mathcal{X}$ and $\mathcal{Y}$ are convex polytopes. The verification tools provide either a guarantee that the property holds (UNSAT) or a satisfying x' such that $x' \in \mathcal{X} \wedge f(x) \in \mathcal{Y}$ (SAT).

When x is an image, neural network verification tools can compute the robustness of the neural network to noise added to a given image. Local robustness around an image x for a neural network with a scalar output can be defined as

$$\|\tilde{x}\|_\infty < \delta \implies |f(x + \tilde{x}) - f(x)| < \epsilon \quad (6)$$

where δ limits changes to pixels, ϵ limits change to the network output, and $\tilde{x}$ is image noise. Verifying robustness around validation images can check if the network is overly sensitive to small perturbations, but robustness alone cannot determine what level of perturbations can be safely tolerated.

III. Methodology

This section describes the Adaptive Stress Testing method used to validate image-based neural network controllers.

A. Adaptive Stress Testing

Adaptive Stress Testing (AST), is a particular configuration of model-free reinforcement learning (RL). Rather than learning a policy that optimizes performance of an agent in an environment, AST optimizes the environment to cause a learned agent to fail. AST uses states s that define the state of

the simulator and actions a that define environmental factors controlled by the simulator. This work considers image-based neural networks, so actions are disturbances to the input images of the controller.

AST treats the simulator as a black box and interacts with the simulator through the following functions:

1) Initialize(s). Load a state of the simulator.
2) Step(s, a). Advance the simulator one step from state s given action a.
3) IsTerminal(s). Return true if state is a terminal state.
4) IsFailure(s). Return true if the state is a failure state.

The AST reward function $R(s, a)$ encourages RL algorithms to find the most likely sequence of actions that causes the system to fail, as discussed in the following subsection.

B. Reward Function

The goal of AST is to find the most likely sequence of actions $a_{0:t-1}$ from s_0 with $s_i = \text{Step}(s_{i-1}, a_{i-1})$ such that IsFailure(s_t) is true. Assuming that each action is independent, the optimization problem becomes

$$\begin{aligned} \underset{a_{0:t-1}}{\text{maximize}} & \prod_{i=0}^{t-1} p(a_i) \\ \text{subject to} \quad & \text{IsFailure}(s_t) \end{aligned} \tag{7}$$

where $p(a_i)$ is the likelihood of the environment producing action a_i.

To incentivize RL algorithms to find such a sequence of actions, AST uses reward function $R(s, a)$ defined as

$$R(s,a) = \begin{cases} 0, & \text{if IsFailure(s)} \\ \log p(a), & \text{else if not IsTerminal(s)} \\ -\alpha - \beta \times \text{Dist}(s), & \text{otherwise} \end{cases} \tag{8}$$

where Dist(s) is some measure of the simulator's closeness to a failure, and α and β scale the penalty term given when a terminal state is reached that is not a failure [21]. In practice α and β are very large to encourage the simulator to find a failure before optimizing the action sequence to reach failure. The following subsection further describes the action space when testing image-based neural networks.

C. Action Space

In previous work with AST, the action space has been low-dimensional [20], [21]. However, when the action space is a high-dimensional image, previous AST algorithms will not perform well. The size of the action space grows exponentially with the dimensionality, so AST would need to sample exponentially more actions to achieve good performance.

This work presents a different approach to computing actions that scales better to high-dimensional spaces. Given a neural network controller f that maps images x to a scalar value y, neural network verification tools can compute an image disturbance $\tilde{x}$ that changes the network output as much as possible assuming that $x + \tilde{x}$ is in the neighborhood of a given image x. This work defines the neighborhood around x as images where each pixel changes by at most δ, though other definitions could be used. For a given input image x, the problem becomes

$$\begin{aligned} \underset{\tilde{x}}{\text{maximize}} \; & f(x+\tilde{x}) - f(x) \\ \text{subject to} \quad & \|\tilde{x}\|_\infty \leq \delta. \end{aligned} \tag{9}$$

Equation (9) can also be written with minimize to compute the perturbation that minimizes the neural network output.

As described in Section II-D, neural network verification tools can compute local robustness properties and verify that the output does not change by more than ϵ (UNSAT) or provide a counterexample where $\tilde{x}$ satisfies $\|\tilde{x}\|_\infty \leq \delta$ and $f(x + \tilde{x}) - f(x) \geq \epsilon$ (SAT). To use these tools in an optimization problem as described in Eq. (9) rather than a satisfiability problem, multiple queries need to be evaluated to search for the largest ϵ value that returns SAT. This search can be done in parallel using the algorithm described in Algorithm 1, which requires the number of parallel queries to run, N, ϵ tolerance, tol, and an upper bound on ϵ, $\bar{\epsilon}$. If no upper bound is known ahead of time, then the algorithm can be modified to increase $\bar{\epsilon}$ until the solver returns UNSAT. Algorithm 1 describes the parallel search for the largest positive change to the network output; the largest negative change to the network output can also be computed in a similar manner. For simplicity, negative values of δ and ϵ used in this work imply a search for the disturbance $\tilde{x}$ that minimizes network output.

Algorithm 1 Parallel image disturbance optimization

Input: f, x, δ, N, tol, $\bar{\epsilon}$
Output: $\underline{\epsilon}$, $\tilde{x}$

$\underline{\epsilon}$, $\tilde{x} = 0$
while $\bar{\epsilon} - \underline{\epsilon} >$ tol
  for $i = 1 : N$
    $\epsilon_i = \underline{\epsilon} + (\bar{\epsilon} - \underline{\epsilon}) \times i/(N+1)$
    result$_i$, $\tilde{x}_i = \text{solve}(f, x, \delta, \epsilon_i)$
  Run all solve calls in parallel
  result$_0$, $\tilde{x}_0$, $\epsilon_0 =$ SAT, $\tilde{x}$, $\underline{\epsilon}$
  result$_{N+1}$, $\epsilon_{N+1} =$ UNSAT, $\bar{\epsilon}$
  $\underline{\epsilon} \leftarrow \max \epsilon_i$ s.t. result$_i =$ SAT
  $\bar{\epsilon} \leftarrow \min \epsilon_i$ s.t. result$_i =$ UNSAT
  $\tilde{x} \leftarrow \tilde{x}_i$ s.t. $\epsilon_i = \underline{\epsilon}$
return $\underline{\epsilon}$, $\tilde{x}$

Algorithm 1 returns the largest ϵ for which there exists a satisfying $\tilde{x}$. Using this approach, the AST action a is equivalent to the image disturbance $\tilde{x}$, effectively reducing the action space for AST from the dimensionality of $\tilde{x}$ to one dimension, δ. Furthermore, since a is constrained by $\|a\|_\infty \leq \delta$, $p(a)$ can be defined as a function of δ using

$$\begin{aligned} p(a) &= \mathcal{N}(\|a\|_\infty \mid 0, \sigma^2) \\ &= \mathcal{N}(\delta \mid 0, \sigma^2) \end{aligned} \tag{10}$$

where σ defines a zero-mean univariate normal distribution. This approach implies that small image disturbances are more likely than large disturbances. Using this approach, AST can be applied to image-based control problems.

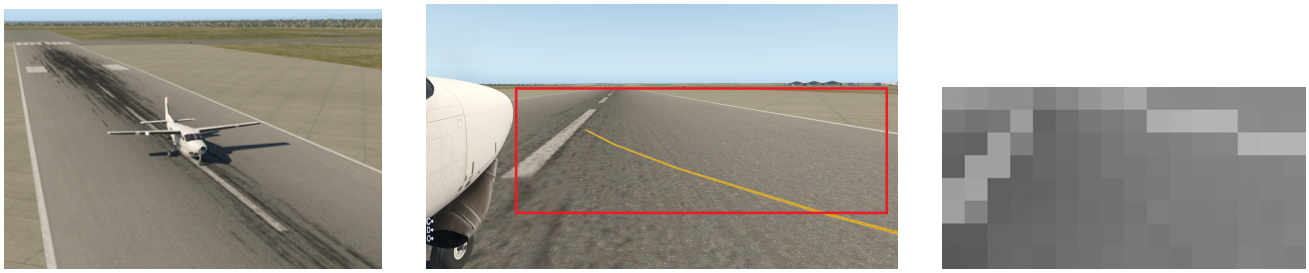

Fig. 1. X-Plane 11 aircraft on taxiway (left), view from camera with cropped region shown in red (middle), and downsampled taxiway image (right)

IV. Aircraft Taxi Application

An aircraft taxi problem is presented here to demonstrate the neural network adaptive stress testing method. A Cessna 208B Grand Caravan simulated in X-Plane 11 is taxiing at 5 m/s along runway 04 of Grant County International Airport and must stay on the taxiway using only images taken once per second from a camera on the right wing of the aircraft. A neural network is trained through supervised learning to map runway images to crosstrack position d and heading angle θ, which are used to control the aircraft. The taxiway center is defined as $d = 0\,\text{m}$, and the taxiway heading angle is defined as $\theta = 0°$. The following subsections discuss the design of the neural network controller, preliminary validation, and AST experimental setup.

A. Neural Network Design

Existing works show that image-based neural networks are susceptible to adversarial attacks, where small changes to pixel values cause large changes to the network output [25], [26]. Because this application uses runway images as inputs, the images can be downsampled significantly without losing important information, which allows the trained network to be smaller as well. The reduced input representation and network size may help the network be more robust to pixel perturbations, and a smaller neural network representation will be more quickly verified by Marabou.

The following procedure was used to shrink input images from 200×360 RGB images to 8×16 grayscale images:

1) Crop out the sky and airplane nose.
2) Resize image to 128×256 and convert to grayscale.
3) Downsample image by splitting image into 128 16×16 boxes and averaging the 16 brightest pixels within each box, resulting in an 8×16 image.
4) Bias all pixel values so that the average value is 0.5 (when pixel values range from 0 to 1).

Forcing the mean pixel value to be 0.5 helps the network generalize to different lighting conditions and increases robustness by adding a constraint to adversarial images.

A neural network with 3 hidden layers and 32 total ReLU activations was trained. The network is composed of an 8×8 convolutional layer with 8 filters and stride of 8 followed by two fully connected layers of size 8. The architecture was designed to contain as few ReLUs as possible while still providing accurate estimates of d and θ. Network outputs d and θ are combined into a rudder command r with proportional control law

$$r = 0.015d + 0.008\theta. \tag{11}$$

Because the final layer of the neural network and proportional control law are linear, they can be combined by modifying the last layer of the neural network. The final neural network controller maps 8×16 downsampled images of the runway to rudder commands.

B. Preliminary Validation

To test the controller, 100 random simulations were run for four cloud conditions with initial crosstrack position $d_0 \sim \mathcal{U}(-5\,\text{m}, 5\,\text{m})$ and heading angle $\theta_0 \sim \mathcal{U}(-20°, 20°)$ at a random simulated time between 9am and 3pm. After 20 seconds elapsed, crosstrack position had an average absolute value of 0.287 m with standard deviation 0.534 m and maximum value 1.015 m, and the heading angle had an absolute value under 2° for all 400 simulations.

These results suggest that the controller performs well, but success in nominal conditions does not mean the controller will always perform correctly. Taxiways in the real world have skid marks, reflective puddles, paint spots, and more, so the controller also needs to perform well when the images are perturbed. The method proposed here addresses these questions to better understand the network shortcomings and failure modes, as discussed in Section V.

C. Experimental Setup

The Marabou tool was used to generate image perturbations because the tool scales well to high-dimensional inputs and the python interface was easy to integrate to other python components [10]. Three types of AST experiments were run: MCTS with Marabou, DQN with Marabou, and MCTS with random samples. All experiments used a discrete number of actions, N_{actions}, with $\delta \in \text{linspace}(-\delta_{\max}, \delta_{\max}, N_{\text{actions}})$. MCTS begins with a root node near the center of the taxiway, while the initial state for DQN trajectories is initialized with $d_0 \sim \mathcal{U}(-5\,\text{m}, 5\,\text{m})$ and $\theta_0 \sim \mathcal{U}(-20°, 20°)$. When random samples are used instead of Marabou, 5000 random image perturbations are sampled, and the perturbation that changes the network output the most is used. The time required to sample and evaluate 5000 images is approximately the time used by Marabou. Failure states are defined as $|d| > 10\,\text{m}$, and terminal states are defined as more than $200\,\text{m}$ downtrack.

TABLE I

AST RESULTS USING MCTS

N_{actions}	$\delta_{\max}$	Failure?	Steps	δ_{mean}	Log-likelihood
2	0.02	No	N/A	N/A	N/A
2	0.027	No	N/A	N/A	N/A
2	0.028	Yes	26	0.028	-125.812
2	0.029	Yes	24	0.029	-122.975
2	0.035	Yes	21	0.035	-147.923
2	0.040	Yes	11	0.040	-98.108
3	0.035	Yes	35	0.030	-215.913
4	0.035	Yes	35	0.031	-213.871
6	0.035	Yes	46	0.028	-249.541

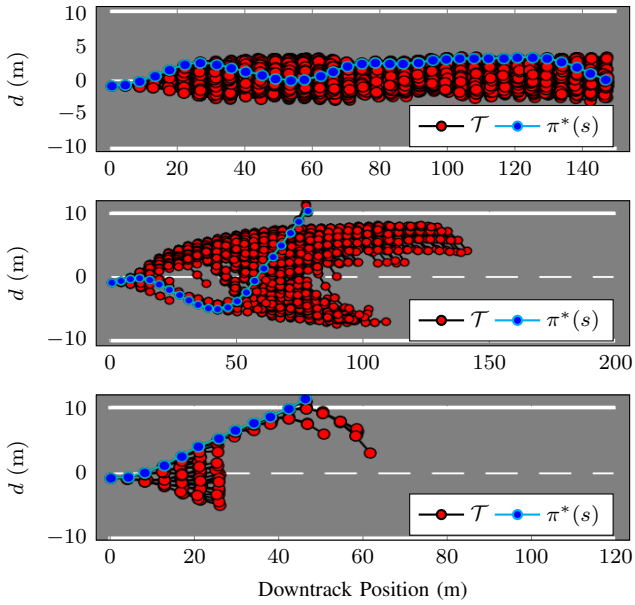

Fig. 2. MCTS search trees when using two actions with $\delta_{\max} = 0.02$ (top), $\delta_{\max} = 0.035$ (middle), and $\delta_{\max} = 0.04$ (bottom)

The experiments use NASA's open source XPlaneConnect to interface with X-Plane 11. Experiments were conducted on a desktop with 16GB of RAM, a 6 core Intel i7 processor, and an NVidia GeForce GTX 1070 Ti GPU. Ten Marabou queries were run in parallel with a rudder disturbance tolerance of 0.003. To speed up MCTS rollouts, a set of rollouts were pre-computed and used to approximate new rollout values through linear interpolation.

V. RESULTS

A summary of results using MCTS with Marabou is shown in Table I. When only two actions are used, δ is always equal to $\delta_{\max}$, while larger numbers of discrete actions also use actions with δ values less than the maximum. When $\delta_{\max}$ is too small, AST cannot find a sequence of image disturbances that leads to failure, while failure sequences are easily found for large $\delta_{\max}$. At some critical $\delta_{\max}$, image disturbances are just strong enough to control the aircraft off the taxiway. Increasing the number of actions makes finding a failure sequence more difficult because MCTS also considers many weak image disturbances, which are less likely to result in failure. As a result, MCTS finds longer sequences to failure, which has a lower log-likelihood but also a lower δ_{mean}.

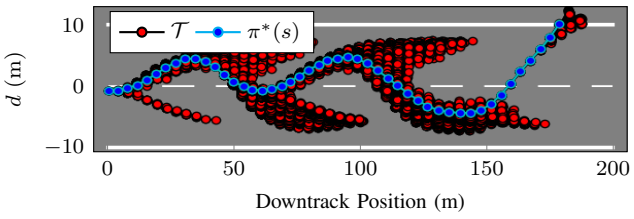

Fig. 3. MCTS search tree when using six actions with $\delta_{\max} = 0.035$

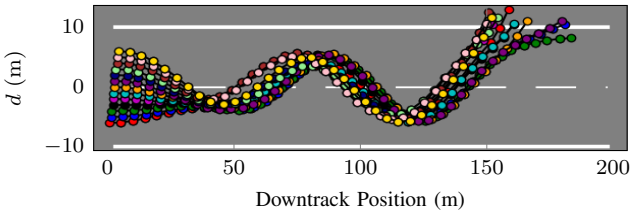

Fig. 4. Simulated trajectories using learned DQN policy

Figure 2 shows the MCTS search tree $\mathcal{T}$ along with optimized policy $\pi^*(s)$. For $\delta_{\max} = 0.02$ the aircraft never exceeds 5 m from the centerline. For $\delta_{\max} = 0.04$, disturbances that always maximize the neural network output can push the aircraft off the left side of the taxiway. For $\delta_{\max} = 0.035$, always maximizing the neural network output will not be enough to leave the taxiway. However, AST is able to find a failure by first decreasing the network output to turn the aircraft right before increasing the network output and turning the aircraft sharply left and off the taxiway. This failure sequence represents a novel failure mode that is non-obvious.

Figure 3 shows the search tree for MCTS using 6 discrete actions. The sequence found is longer than when using only two actions, but the behavior is similar. The image disturbances cause the aircraft to oscillate across the centerline like a pendulum, eventually gaining enough momentum to leave the taxiway.

Experimental results using DQN in AST instead of MCTS are shown in Table II. DQN does not perform as well as MCTS for low numbers of actions but scales better to larger number of actions. Whereas MCTS uses only one initial state, DQN creates a policy that generalizes to any initial state, as shown in Figure 4. DQN takes more samples to train the Q-network than to build a search tree, so MCTS required only 1–6 hours while DQN required 24–36 hours to run. However, both methods are model-free and return image

TABLE II

AST RESULTS USING DQN

N_{actions}	$\delta_{\max}$	Failure?	Steps	δ_{mean}	Log-likelihood
2	0.035	Yes	28	0.035	-197.23
4	0.035	Yes	39	0.033	-252.94
6	0.035	Yes	27	0.0318	-166.67

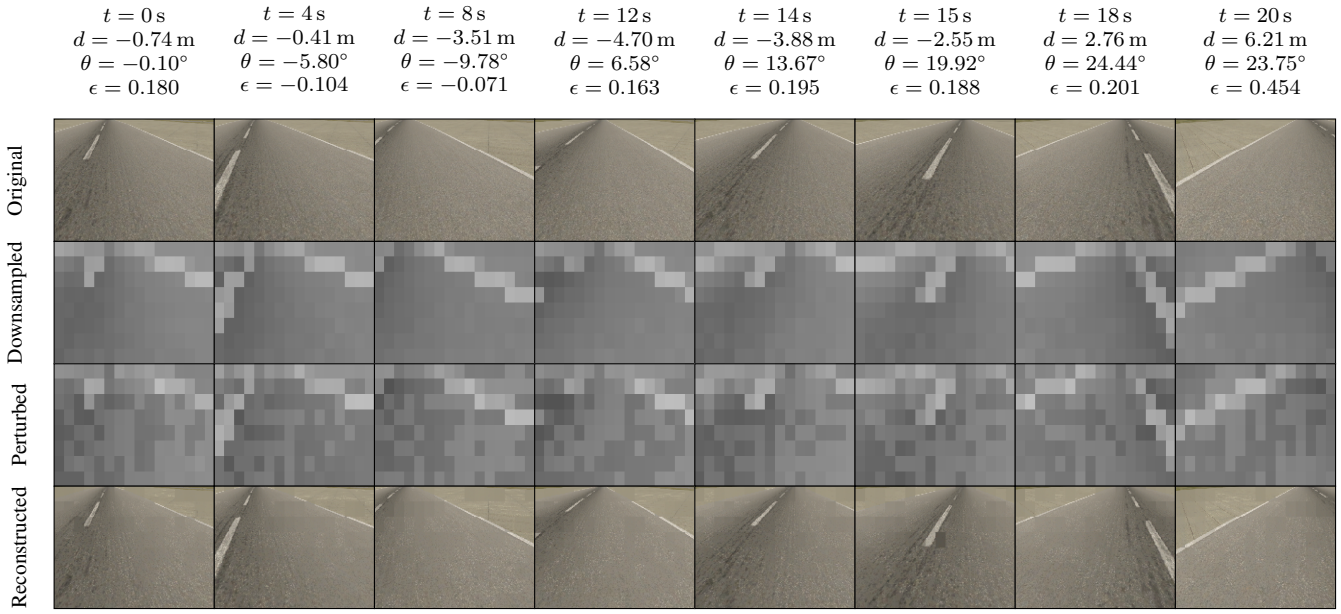

Fig. 5. Images at different times during simulation of AST policy when using MCTS with two actions and $\delta_{\text{max}} = 0.035$

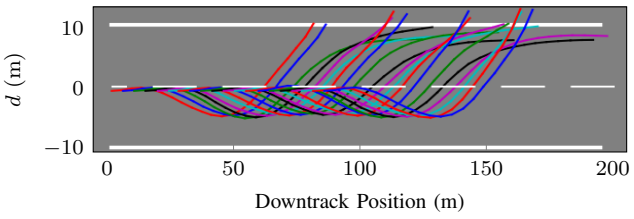

Fig. 6. Simulations of AST policy from different initial downtrack positions

disturbance sequences that lead the aircraft off the taxiway. When random samples are used instead of Marabou with $\delta_{\text{max}} = 0.035$, the crosstrack position never exceeds $5\,\text{m}$. This result demonstrates that using tools like Marabou are important for high dimensional problems.

Figure 5 shows four types of images at different times along the two action MCTS policy with $\delta_{\text{max}} = 0.035$: the original taxiway image, downsampled image, downsampled image with perturbation added, and a reconstructed image of the taxiway image that would produce the perturbed image if downsampled. Reconstructing the taxiway image from the perturbed image is an under-defined problem, so this approach simply biases the brightness of pixels in the original taxiway image. The images reveal key insights into how the neural network can be tricked into failure. The reconstruction shows that AST darkens the taxiway edge lines until the thin boundary lines are difficult to discern, indicating that the boundary lines are important for making accurate predictions.

Furthermore, times $t = 14, 15$ show that AST guides the aircraft across the centerline in the gap between centerline dashes. These images also have much greater ϵ values than previous times, which suggests that the neural network is more easily fooled when the aircraft travels between the centerline dashes. To investigate this hypothesis, the AST policy was simulated from different initial points along the taxiway. As shown in Figure 6, trajectories that cross the centerline between dashes reach the edge of the taxiway while other trajectories remain on the taxiway.

VI. Conclusions

Although neural networks perform well empirically, they can be susceptible to adversarial attacks. For safety critical image-based applications acting in the real world, verifying that failures will never occur is intractable or impossible. This work presented a method for validation of image-based neural network controllers that uses reinforcement learning to find the most likely failure modes. The analysis is tractable for high-dimensional inputs, and a taxiway navigation application demonstrated how the algorithm can be integrated with a black box simulator. The results showed that adaptive stress testing finds image disturbances that cause the neural network to guide the aircraft off the runway and revealed that the gaps between centerline dashes are more susceptible to adversarial perturbations.

Future work will study methods for making the neural network more robust to adversarial attacks and incorporate continuous actions. In addition, other image perturbations besides pixel disturbances could be considered, such as large skid marks or reflective puddles. Future work could also study how adding an additional camera to the left wing of the aircraft improves system robustness. Finally, a real aircraft on a taxiway could be used to validate the accuracy and applicability of the X-Plane 11 simulator.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[4] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[5] D. A. Pomerleau, "Alvinn: An autonomous land vehicle in a neural network," in *Advances in Neural Information Processing Systems (NeurIPS)*, 1989.

[6] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[7] K. D. Julian and M. J. Kochenderfer, "Neural network guidance for UAVs," in *AIAA Guidance, Navigation, and Control Conference (GNC)*, 2017.

[8] S. Bansal, A. K. Akametalu, F. J. Jiang, F. Laine, and C. J. Tomlin, "Learning quadrotor dynamics using neural network for flight control," in *IEEE Conference on Decision and Control (CDC)*, 2016.

[9] G. Katz, C. Barrett, D. L. Dill, K. D. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *International Conference on Computer Aided Verification*, 2017.

[10] G. Katz, D. A. Huang, D. Ibeling, K. Julian, C. Lazarus, R. Lim, P. Shah, S. Thakoor, H. Wu, A. Zeljić, *et al.*, "The marabou framework for verification and analysis of deep neural networks," in *International Conference on Computer Aided Verification*, 2019.

[11] A. Lomuscio and L. Maganti, "An approach to reachability analysis for feed-forward ReLU neural networks," *arXiv preprint arXiv:1706.07351*, 2017.

[12] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Efficient formal safety analysis of neural networks," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[13] C. Liu, T. Arnon, C. Lazarus, C. Barrett, and M. J. Kochenderfer, "Algorithms for verifying deep neural networks," *arXiv preprint arXiv:1903.06758*, 2019.

[14] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Verisig: Verifying safety properties of hybrid systems with neural network controllers," in *Hybrid Systems: Computation and Control (HSCC)*, 2019.

[15] M. Akintunde, A. Lomuscio, L. Maganti, and E. Pirovano, "Reachability analysis for neural agent-environment systems," in *International Conference on Principles of Knowledge Representation and Reasoning*, 2018.

[16] W. Xiang and T. T. Johnson, "Reachability analysis and safety verification for neural network control systems," *arXiv preprint arXiv:1805.09944*, 2018.

[17] K. D. Julian and M. J. Kochenderfer, "Guaranteeing safety for neural network-based aircraft collision avoidance systems," in *Digital Avionics Systems Conference (DASC)*, 2019.

[18] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Towards proving the adversarial robustness of deep neural networks," in *Workshop on Formal Verification of Autonomous Vehicles (FVAV)*, 2017.

[19] D. Gopinath, G. Katz, C. S. Păsăreanu, and C. Barrett, "Deepsafe: A data-driven approach for assessing robustness of neural networks," in *International Symposium on Automated Technology for Verification and Analysis*, 2018.

[20] R. Lee, M. J. Kochenderfer, O. J. Mengshoel, G. P. Brat, and M. P. Owen, "Adaptive stress testing of airborne collision avoidance systems," in *Digital Avionics Systems Conference (DASC)*, 2015.

[21] M. Koren and M. J. Kochenderfer, "Efficient autonomy validation in simulation with adaptive stress testing," in *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2019.

[22] M. J. Kochenderfer, "Sequential problems," in *Decision Making under Uncertainty: Theory and Application*, MIT Press, 2015, ch. 4, pp. 77–112.

[23] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, *OpenAI baselines*, `https://github.com/openai/baselines`, 2017.

[24] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *International Conference on Learning Representations (ICLR)*, 2016.

[25] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations (ICLR)*, 2015.

[26] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *IEEE Symposium on Security and Privacy (SP)*, 2017.