Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)

Trajectory Tracking Control for Robotic Vehicles Using Counterexample Guided Training of Neural Networks

Arthur Clavière,[1] **Souradeep Dutta,**[2] **Sriram Sankaranarayanan**[2]
[1]École Polytechnique and ISAE-SUPAERO, France
[2]University of Colorado Boulder, USA

Abstract

We investigate approaches to train neural networks for controlling vehicles to follow a fixed reference trajectory *robustly*, while respecting limits on their velocities and accelerations. Here robustness means that if a vehicle starts inside a fixed region around the reference trajectory, it remains within this region while moving along the reference from an initial set to a target set. We consider the combination of two ideas in this paper: (a) demonstrations of the correct control obtained from a model-predictive controller (MPC) and (b) falsification approaches that actively search for violations of the property, given a current candidate. Thus, our approach creates an initial training set using the MPC loop and builds a first candidate neural network controller. This controller is repeatedly analyzed using falsification that searches for counterexample trajectories, and the resulting counterexamples are used to create new training examples. This process proceeds iteratively until the falsifier no longer succeeds within a given computational budget. We propose falsification approaches using a combination of random sampling and gradient descent to systematically search for violations. We evaluate our combined approach on a variety of benchmarks that involve controlling dynamical models of cars and quadrotor aircraft.

Introduction

We study how counterexamples generated from systematic search can be used to retrain neural networks to control a vehicle to follow a desired trajectory while maintaining limits on the lateral deviation from the trajectory as well as keeping the velocities and accelerations within fixed limits. The target networks to be trained input the current estimates of vehicle state, i.e, its position, pose, and velocities. The network outputs a feedback value that determines the vehicle's acceleration. Although our approach focuses on a given single robust trajectory, it can be readily integrated into a higher level planner that is equipped with a library of trajectories and their corresponding feedback laws discovered by our approach as *motion primitives* (Majumdar and Tedrake 2017; Frazzoli, Dahleh, and Feron 2005).

One approach to infer such networks is through *reinforcement learning*, wherein a Markov decision process (MDP) is defined with rewards for achieving a trajectory that satisfies the specifications, and punishments (or negative reward) for violations (Sutton and Barto 1998). A variety of deep reinforcement learning approaches can then be used to train these networks (Goodfellow, Bengio, and Courville 2016). However, a key disadvantage of this approach is the need to convert properties into numerical rewards. For instance, adding a reward of ∞ for property satisfaction and $-\infty$ for violations often results in very poor performance. In this paper, we study an alternative approach based on counterexample guided synthesis of control systems (Alur et al. 2013). At a high level, counterexample guided synthesis proposes candidate solutions for the synthesis problems that are then subject to formal verification using approaches such as model checking (Baier and Katoen 2008). If the verification succeeds, we provide a verified controller. Alternatively, if it fails, the verifier describes a counterexample in terms of initial states and inputs that cause the system to fail. Based on this counterexample, we eliminate the current candidate control law, as well as all other candidates that would be invalidated by the same counterexample.

However, applying this approach to learning neural network controllers is quite challenging. For one, counterexample-guided synthesis approaches are limited to domains where verification tools can effectively prove correctness or discover violations. Furthermore, the ability to eliminate candidates based on counterexamples is important. Both problems are challenging for systems that combine physical dynamics with feedback through neural network controllers. In this paper, we propose partial solutions that employ two ideas to enable counterexample-guided training: (a) we propose falsification solvers that analyze a candidate neural network controller in the closed loop to search for concrete trajectories that violate the trajectory; (b) we use constrained model-predictive control (MPC) to compute a control strategy for the counterexample states discovered by our falsifier to provide appropriate controls to be employed.

We demonstrate the application of our overall approach to benchmark systems including steering ground vehicles and quadrotor aircrafts. For each instance, we consider a variety of trajectories and show that our approach can infer networks that can robustly control for these trajectories while

satisfying bounds on the velocities and the control inputs. We also compare our falsification search to random search to demonstrate that a systematic falsifier can discover violations even when numerous random samples do not uncover property violations. Our approach succeeds in 44 out of 50 instances tested, using tiny network topologies with fewer than 3 layers and 10 neurons per layer to implement the feedback law.

Related Work

Neural networks have been applied to control dynamical systems using approaches such as deep reinforcement learning (Mnih et al. 2013; Lillicrap et al. 2015; Levine et al. 2015). However, deep reinforcement learning differs from the approach presented here in many fundamental ways. For one, our approach relies on a dynamical model of the system whereas deep reinforcement learning does not necessarily require such a model. Another key difference is that our approach works with property specifications of what a system must and must not do, whereas deep reinforcement learning operates using positive and negative rewards corresponding to states and actions. Using negative rewards for violations and large positive rewards for property satisfaction is problematic, whenever temporal properties are involved. In such situations, large negative/positive rewards may be obtained for a tiny fraction of the overall state action pairs, whereas zero rewards are obtained elsewhere wherein the status of the property is as yet unknown. Attempts to circumvent this often require hand-designed and problem specific rewards. Alternatively, preference learning approaches can be used (Wirth et al. 2017; Sadigh et al. 2017). Model based reinforcement does not directly tackle the problem of "robust policies", wherein the behavior of a policy on a neighborhood of a given state is also of concern. Also, in this work, we incorporate a counterexample generator (falsification) explicitly in the learning loop. This iteration is akin to a supervised learning setup, wherein the teacher is obtained by a combining the counterexample generator and the demonstrator in the form of a model-predictive controller (MPC).

The use of MPC as an expert teacher to train a neural network iteratively has been explored by approaches such as DAGGER and PLATO. Both approaches share a broad commonality with our approach, namely the use of an expert teacher to help train a neural network. Furthermore, our approach follows the broad outline of the DAGGER strategy, wherein we iteratively add to a training dataset by exploring the current policy (Ross, Gordon, and Bagnell 2011). Whereas, DAGGER mixes the expert demonstrator with the current policy (i.e, flips a coin to choose between running the expert or the current policy), our approach explores counterexamples to properties obtained from the current policy. Pereira et al present a recent adaptation of this idea to using MPCs in particular (Pereira et al. 2018). PLATO, on the other hand focuses on learning how to control directly from sensor/camera inputs to the system using an MPC as a supervisor. A key difference is that PLATO adapts the MPC to balance making an optimal/safe control action against matching the current policy (Kahn et al. 2017).

Deep neural networks used for classification tasks in computer vision, has been shown to be fairly brittle when it comes to adversarial inputs (Goodfellow, Bengio, and Courville 2016). Similarly, adversarial inputs have been shown for neural networks representing policies learned through reinforcement learning approaches (Huang et al. 2017). The use of counterexamples can be seen as an adversarial attack that continually challenges the neural network. However, in combination with the MPC the adversary now serves as a *teacher* who alternates between testing and informing the learner.

The use of MPCs has been explored recently to train control Lyapunov (potential) functions by Ravanbakhsh et al (Ravanbakhsh and Sankaranarayanan 2017). Therein, the authors use a counterexample guided learning loop driven by a verifier that checks a candidate potential function. The counterexamples found by this verifier are then updated through demonstrations. Finally, the approach uses cuts to refine the space of remaining candidates. Our work has similarities in that it is counterexample driven as well. However, our candidate space is given by the set of possible weights of the network we wish to train. Also, we directly learn a policy rather than a potential function.

The idea of using neural networks to directly mimic MPC controllers has been studied (Piche et al. 2000; Psichogios and Ungar 1991). In this work, we use counterexamples to be sample efficient in the learning process. This is important since MPC is well known to be expensive, in practice.

Finally, the area of verification of neural networks and learning with guarantees has received considerable interest, recently. Leofonte et al provide a survey of recent advances (Leofante et al. 2018). Approaches using ideas such as barrier certificates (Tuncali et al. 2018), and reachability analysis (Xiang and Johnson 2018; Dutta et al. 2018) have been explored in the context of learning. One of the recent approaches, in testing against system-level specifications for control systems with machine learning components was proposed in (Yaghoubi and Fainekos 2018). However, these approaches either focus on learning a network and a proof of correctness at the same time, or decouple the learning process from the verification process without feeding back into the learning.

Problem Statement and Approach

In this section, we will discuss the problem statement and a high level overview of the approach proposed for its solution. The inputs to our problem include a mathematical *vehicle model* and a desired reference *trajectory*. The vehicle model is given as an ordinary differential equation (ODE)

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}, \mathbf{w}) \,,$$

wherein $\mathbf{x}$ represents the state of the vehicle that includes position, pose (orientation) and velocities. We will consider different benchmark vehicle models that will include a simple car model as well as a model of a quadcopter $\mathbf{u}$ represents the control inputs that include throttle/brakes and steering inputs to the vehicle; and $\mathbf{w}$ models external uncontrolled disturbances that affect the vehicle dynamics. We will denote by $X \subseteq \mathbb{R}^n$ the set of possible vehicle states,

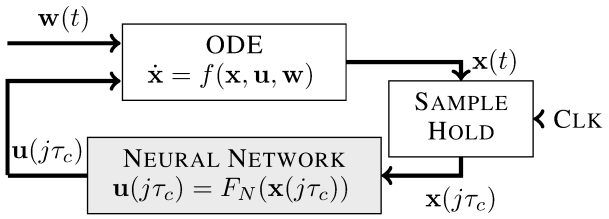

Figure 1: Block diagram of a neural feedback control system.

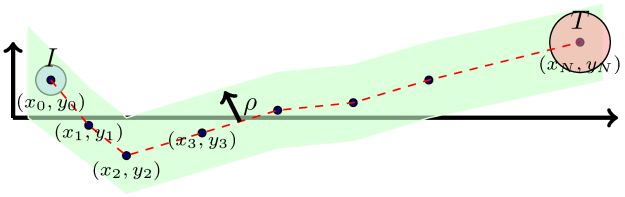

Figure 2: Illustration of the trajectory, initial set I and target set T, the desired reference trajectory C specified by the line joining the waypoints and the safe set given by a cylinder S of radius ρ around the curve C.

$U \subseteq \mathbb{R}^m$ the set of possible control inputs and $W \subseteq \mathbb{R}^k$ the set of possible disturbances.

Next, we describe the trajectory tracking specification. Let (x_0, y_0, z_0) describe the position of the vehicle in space. Our goal is to steer the vehicle to follow a curve specified by *way points* $(x_0, y_0, z_0), (x_1, y_1, z_1), \ldots, (x_N, y_N, z_N)$, in a robust manner. Note that in a planar version of the problem, the z coordinate will be omitted. Let C be the curve formed by connecting each waypoint to the next using a straight line (alternatively, a low degree spline that passes through the way points may be chosen). Our goal is to control the vehicle so that starting from an initial set I containing the initial point, the vehicle stays within a cylinder of radius ρ around the curve C, denoted by the *safe set* S until it reaches a target set T that contains the final way point (x_N, y_N, z_N). Mathematically, a cylinder of radius ρ is defined as $S : \bigcup_{(x,y,z)\in C} \mathbf{Ball}_\rho(x, y, z)$, wherein $\mathbf{Ball}_\rho(x_c, y_c, z_c) : \{(x, y, z) \mid \| ((x_c - x), (y_c - y), (z_c - z)) \|_2 \leq \rho\}$.

The overall goal given the model and trajectory is to design a feedback function $F_N : X \mapsto U$ that maps the current state of the vehicle to an associated control input. Figure 1 shows the overall diagram of the control setup for the problem. The feedback function F_N is periodically executed using a clock with time period of τ_c.

Approach

Figure 3 illustrates our overall approach to the problem. Our approach begins with an initial training data $\mathcal{D}_0$ that involves pairs $(\mathbf{x}_i, \mathbf{u}_i)_{i=1}^{N_0}$ of N_0 states and corresponding control inputs generated by querying a model-predictive controller, which is a computationally expensive but reliable approach to solving the trajectory tracking problem. The overall approach iterates starting from data $\mathcal{D}_i$ to produce data $\mathcal{D}_{i+1}$

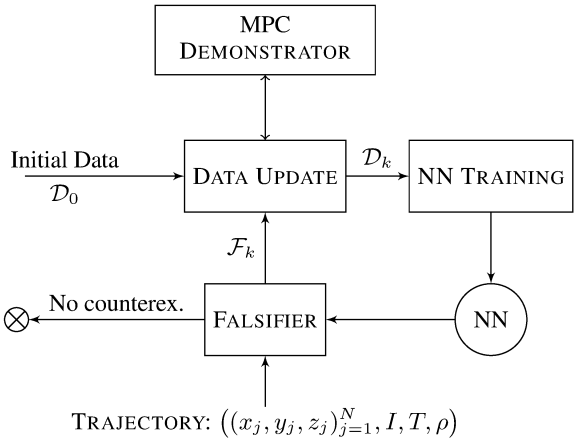

Figure 3: A schematic illustration of the counterexample-driven training approach.

as follows:

1. First we use standard backpropagation-based TRAINING to build a feedforward neural network (NN) using the current data set $\mathcal{D}_i$. This network $\mathcal{N}_i$ is the current candidate controller that maps from a state $\mathbf{x}$ to a control $\mathbf{u}$.
2. A FALSIFIER is used to systematically "attack" this controller in order to find a set of counterexamples. Each counterexample is a sequence of states starting from an initial state wherein the feedback produced by the network $\mathcal{N}_i$ leads to a violation.
3. The falsification result is used to perform a DATA UPDATE, wherein the MPC is queried to obtain new data. This is *blended* in with existing samples to yield the data $\mathcal{D}_{i+1}$ for the next round.
4. The procedure terminates successfully if the FALSIFIER is unable to discover counterexamples after some fixed number of trials, or in failure if some upper bound K on the number of iterations is exceeded.

The rest of the paper describes each of the components in turn starting from the model predictive controller (MPC) that is used as a "teacher" or "demonstrator" to obtain training data for our learning process.

Model Predictive Control

A receding horizon Model Predictive Controller (MPC) is used to compute a feedback law given the current state of the vehicle $\mathbf{x}_0$, and a time horizon $k\tau_c$ that "looks ahead" k steps into the future, each step taking τ_c time. The output is a sequence of control inputs $\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{k-1}$ wherein $\mathbf{u}_i$ is intended control for time $i\tau_c$. Typically, the very first control input in the sequence is applied to the vehicle. After time τ_c, a new state measurement $\mathbf{x}_1$ is obtained and the computations are carried out afresh.

A MPC requires knowledge of the dynamics of the vehicle f, which is time discretized using a Euler scheme as:

$$\mathbf{x}(t + \tau_c) = \mathbf{x}(t) + \tau_c f(\mathbf{x}(t), \mathbf{u}(t)) .$$

Disturbance is not considered in this set up for simplicity.

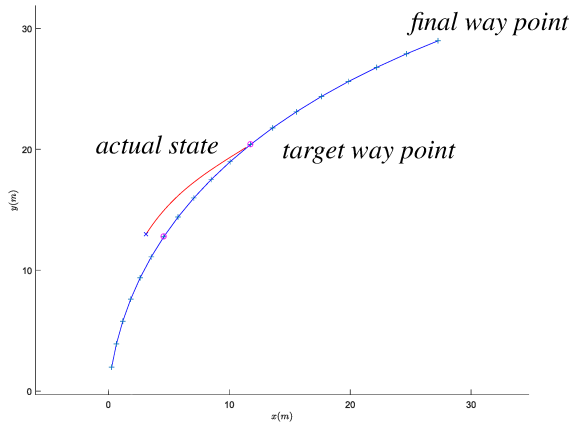

Figure 4: Choice of a local target for the MPC.

MPC poses an optimization problem wherein the future control inputs $\mathbf{u}_0, \ldots, \mathbf{u}_{k-1}$ are unknowns along with the unknown future vehicle states: $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k)$. Rather than impose constraints, we will design a cost function $\mathcal{C}(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k, \mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_{k-1})$ is used to evaluate the quality of the control and the amount of control effort. Thus, the overall MPC scheme solves the following optimization problem :

$$\begin{array}{ll} \min & \mathcal{C}(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k, \mathbf{u}_0, \ldots, \mathbf{u}_{k-1}) \\ \text{s.t.} & \vec{x}_1 = \mathbf{x}_0 + \tau_c f(\mathbf{x}_0, \mathbf{u}_0) \\ & \vdots \\ & \vec{x}_k = \mathbf{x}_{k-1} + \tau_c f(\vec{x}_{k-1}, \mathbf{u}_{k-1}) \end{array} \tag{1}$$

MPC for Trajectory Tracking

We will now describe the use of MPC for trajectory tracking given waypoints $(x_i, y_i, z_i)_{i=0}^N$. The MPC formulation ignores the initial set I, target set T and safety set S. Instead, it simply focuses on steering the vehicle to stay as close as possible to the curve C connecting the waypoints. Although such an approach is not guaranteed to yield a controller that satisfies the properties, we find that, in practice, a cost function can be designed using trial and error to achieve the desired robust trajectory tracking for chosen initial sets I, target sets T and safe set S. Also note that designing an MPC is not the final goal of our approach.

First, given the current vehicle position, it is important to choose a corresponding waypoint as a *local target* for the MPC. The choice is illustrated in Figure 4. Algorithm 1 shows the formal computations needed by the MPC to choose a target way point by first computing the waypoint whose position is closest to the current position of the vehicle and using one that is p positions ahead in the sequence of way points. We also compute a rough estimate of the number of lookahead steps N based on the current velocity and distance from the chosen target waypoint.

The overall cost function is simply given by the distance from the position achieved by the vehicle at time $N\tau$ (N steps into the future) and the desired way point for that time

Algorithm 1 Algorithm to systematically compute the target way point and the time horizon, given the current state of the system and the way points in the trajectory.

1: **procedure** PREPROCESSING(Current State $\mathbf{x}$, Trajectory $\mathcal{T}$, Velocity v, time step δt)
2: $ind_C \leftarrow$ findIndexClosestWayPoint ($\mathbf{x}$)
3: $ind_T \leftarrow ind_C$ + p
4: WayPoint $\leftarrow \mathcal{T}[ind_T]$
5: dist $\leftarrow$ calculateDistancePosition($\mathbf{x}$, WayPoint)
6: N $\leftarrow \lfloor dist/(v \times \delta t) \rfloor$
7: **return** WayPoint, N

chosen using Algorithm 1, plus a penalty term that penalizes the magnitude of the control effort needed. The advantage of choosing a waypoint p positions ahead lies in enhanced stability since the closest waypoint itself will often be unachievable given the current vehicle state. As mentioned earlier, MPC involves a nonlinear optimization problem that is computationally expensive to solve in practice. However, it serves in our framework as a demonstrator that yields a control input that can be applied at a given state to move the vehicle towards a given trajectory.

Training Neural Network Controllers

We use a neural network as a function approximator to mimic the control actions generated by an MPC controller. We assume that we have the control action as a traning set given by, Train $= \{(\mathbf{x}_0, \mathbf{u}_0), (\mathbf{x}_1, \mathbf{u}_1), \ldots, (\mathbf{x}_n, \mathbf{u}_n)\}$. The difference between the function approximated by the neural network, $F_{\mathcal{N}}(\mathbf{x})$ and the set of point Train, can be expressed as a simple squared loss function as,

$$\mathcal{L} = \sum_{i=1}^{n} (F_{\mathcal{N}}(\mathbf{x}_i) - \mathbf{u}_i)^2$$

Thus, given the above loss function the aim is to tune the weights W, and biases b, by repeated calculation of the gradients $\nabla_W \mathcal{L}$, and $\nabla_b \mathcal{L}$, and updating the values to follow the negative of the gradient. The process of taking gradients for deep neural networks, can be accomplished by a technique known as the back-propagation. We used an off the shelf gradient descent optimizer to train the neural networks in this paper.

Counter Example Generation

In this section, we describe the systematic search for counterexamples through falsification. We will first describe what constitutes a counterexample and present a gradient descent approach that searches for counterexamples starting from randomly sampled initial points. Recall that we are given an initial set I, target set T, waypoints $(x_j, y_j, z_j)_{j=1}^N$ and a corresponding safe set S. Let $\mathcal{N}$ be a neural network that provides a feedback from state $\mathbf{x}$ of the vehicle to a control input $F_N(\mathbf{x})$, wherein F_N is the function described by the network $\mathcal{N}$.

Our goal is to find an initial state $\mathbf{x}_0$ such that the vehicle position belongs to the set I, and the trajectory of the

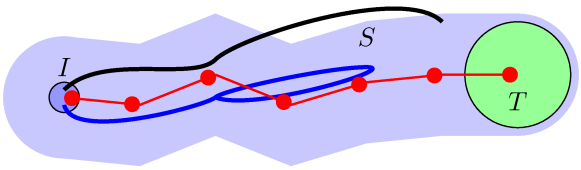

Figure 5: Falsifying trajectory shown in black that exits the safe set S and a falsifying trajectory in blue that remains inside S but does not make progress towards the goal T.

closed loop system with the feedback provided by the network $\mathcal{N}$ (A) reaches a state outside the safe set S before the target T is reached, or alternatively, (B) the target T is not reached after a sufficiently long timeout. Figure 5 depicts the two scenarios. Note that (A) and (B) are qualitatively different classes of properties: whereas (A) concerns a *safety* property asserting that the trajectory must remain inside S, (B) concerns a *liveness* property stating that the target must eventually be reached. Each property requires a qualitatively different approach towards verification and counterexample search. However, since we are working with vehicle models, we can unify both (A) and (B) into safety properties by mandating that at any point $(x, y, z) \in S$ the set of possible velocity vectors must not point "away" from the target way point chosen for (x, y, z) according to Algorithm 1. This allows us to strengthen the set S which constrains the possible positions of the vehicle to a set $\hat{S}$ that also imposes the corresponding constraints on the velocities. As a result, we will focus on finding a trajectory that exits the set $\hat{S}$ which includes constraints on the positions as well as velocities of the vehicle. In summary, given sets I, T and $\hat{S}$, we see a (time discretized) counterexample trajectory $\mathbf{x}(t)$ for $t \in \{0, \tau_c, \cdots, K\tau_c\}$ such that $\mathbf{x}(0) \in I$ and $\mathbf{x}(K\tau_c) \notin \hat{S}$.

The key question is to systematically search for a counterexample trajectory, given a candidate neural network $\mathcal{N}$. A standard "go to" approach involves using randomized search such as uniform random sampling, a "guided" Monte-Carlo search or a metaheuristic search such as TABU or Genetic algorithms. The advantage of such approaches include their ability to treat the system under verification as a black-box model. The complexity of neural networks makes this quite desirable. However, these approaches do not utilize properties such as the continuous sensitivity of the trajectory to initial conditions even in the presence of neural network feedback. In this paper, we will consider an approach that attempts to find a counterexample through gradient descent.

The gradient descent approach to finding a potential falsification relies on the following key steps: (a) identifying a "critical point" and a direction for falsification; (b) "backpropagating" from a chosen point and gradient direction; (c) choosing a step size and (d) terminating the falsification search.

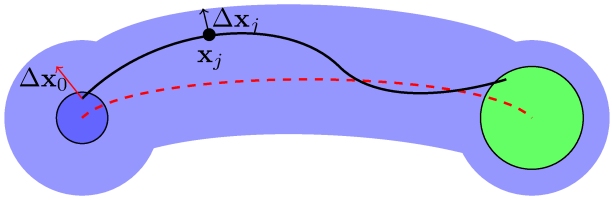

Figure 6: The critical point $\mathbf{x}_j$ and the desired direction $\Delta\mathbf{x}_j$ are chosen based on the closest approach to the boundary of $\hat{S}$. We backpropagate the desired change direction $\Delta\mathbf{x}_j$ at time $t = j\tau_c$ to a direction for the initial state $\Delta\mathbf{x}_0$ at time $t = 0$.

Identifying a Critical Point

Let $\mathbf{x}_j : \mathbf{x}(j\tau_c)$ for $j \in \{0, \ldots, K\}$ be a current sampled trajectory starting from $\mathbf{x}_0 \in I$. Furthermore, we will assume that the current trajectory is not a counterexample: i.e, $\mathbf{x}_j \in \hat{S}$ for all $j \in [0, K]$ with $\mathbf{x}_K \in T$. The goal is to identify a time $t^* = j\tau_c$ such that the corresponding state $\mathbf{x}_j$ is "closest" to violating the safe set $\hat{S}$. We will call $\mathbf{x}_j$ a critical point. Furthermore, we will identify a critical direction $\Delta\mathbf{x}_j$ such that $\mathbf{x}_j + \lambda\Delta\mathbf{x}_j \notin \hat{S}$ for some $\lambda \geq 0$. In other words, moving from the critical point along this direction yields a state that falls outside $\hat{S}$. The process is illustrated in Figure 6.

Identifying a critical point requires us to compute the distance of a trajectory state $\mathbf{x}_j$ to the boundary of the set $\hat{S}$. This is performed in our approach as the minimum of two distances: (a) the distance between the position denoted by $\mathbf{x}_j$ to the boundary of the set S and (b) distance between the velocity denoted by $\mathbf{x}_j$ to the boundary of the set of admissible velocities in $\hat{S}$. Since S is chosen as a cylinder of distance ρ from a curve joining the way points, we simply calculate the distance of $\mathbf{x}_j$ to the curve C, which is in turn computed by calculating the distance to each line segment joining successive way points. Subtracting this distance from the overall cylinder radius ρ yields the distance to the boundary of the safe region S. The direction to the boundary is also given by inverting the perpendicular joining the position $\mathbf{x}_j$ to the curve C.

Backpropagating the Gradient

Having chosen a critical point $\mathbf{x}_j$ at time $t = j\tau_c$ and a direction $\Delta\mathbf{x}_j$, our goal is to "backpropagate" this direction back to the initial set to obtain a direction $\Delta\mathbf{x}_0$. We obtain $\Delta\mathbf{x}_0$ by solving the equation involving the Jacobian $J : \frac{d\mathbf{x}_j}{d\mathbf{x}_0}$:

$$J\Delta\mathbf{x}_0 = \Delta\mathbf{x}_j \,.$$

However, the equation need not always have a solution. One alternative is to solve it in a least square sense to find a direction $\Delta\mathbf{x}_0$ that results in a new trajectory which at time $j\tau_c$ approaches closest to $\mathbf{x}_j + \Delta\mathbf{x}_j$. Another alternative is to choose $\Delta\mathbf{x}_0$ as a gradient of a cost function defined over states $\mathbf{x}$ encountered at time $t = j\tau_c$:

$$c(\mathbf{x}) : \; -(\Delta\mathbf{x}_j)^T \, (\mathbf{x} - \mathbf{x}_j)$$

The function is simply the dot product between the change direction $\mathbf{x} - \mathbf{x}_j$ caused by moving the initial state to $\mathbf{x}_0 + \Delta\mathbf{x}_0$ and the desired direction $\Delta\mathbf{x}_j$. We compute $\frac{dc(\mathbf{x})}{d\mathbf{x}_0}$ as

$$\frac{dc(\mathbf{x})}{d\mathbf{x}_0} = -(\Delta\mathbf{x}_j)^T \frac{d\mathbf{x}}{d\mathbf{x}_0}|_{\mathbf{x}=\mathbf{x}_j},$$

which in turn yields a gradient descent direction :

$$\Delta\mathbf{x}_0 = -\frac{dc(\mathbf{x})}{d\mathbf{x}_0} = (\Delta\mathbf{x}_j)^T \frac{d\mathbf{x}_j}{d\mathbf{x}_0}.$$

Given such a direction, we update

$$\mathbf{x}_0 := \mathbf{x}_0 + \delta \frac{\Delta\mathbf{x}_0}{||\Delta\mathbf{x}_0||} \; \delta > 0$$

wherein δ is a chosen step size. There are many standard approaches to choosing δ such as the Armijo step sizing rule (Luenberger 1977). Note that if the new initial state $\mathbf{x}_0 \notin I$, we will need to reduce δ to ensure that $\mathbf{x}_0 \in I$ or use a projection operator that projects a step outside the initial set I back into the initial set. Assuming I is given as a box, our implementation projects a point $\mathbf{x}_0 \notin I$ back to the set I by thresholding each dimension $\mathbf{x}_{0,i}$ that goes out of bounds back to the upper or lower limit.

We will now go over the computation of $\frac{d\mathbf{x}_j}{d\mathbf{x}_0}$. Let us use the notation K_j to denote the matrix $\frac{d\mathbf{x}_j}{d\mathbf{x}_0}$. Using the chain rule, we have the following recursive formulation of K_{j+1} in terms of K_j:

$$\begin{aligned} K_{j+1} &= \frac{d\mathbf{x}_{j+1}}{d\mathbf{x}_j}\frac{d\mathbf{x}_j}{d\mathbf{x}_0} \\ &= \frac{d(\mathbf{x}_j + \tau_c f(\mathbf{x}_j, \mathbf{u}_j))}{d\mathbf{x}_j} K_j \\ &= \left(I + \tau_c \left(\frac{\partial f(\mathbf{x}_j, \mathbf{u}_j)}{\partial \mathbf{x}_j} + \frac{\partial f(\mathbf{x}_j, \mathbf{u}_j)}{\partial \mathbf{u}_j}\frac{d\mathbf{u}_j}{d\mathbf{x}_j}\right)\right) K_j \end{aligned}$$

Note that $\frac{\partial f(\mathbf{x}_j, \mathbf{u}_j)}{\partial \mathbf{x}_j}$ and $\frac{\partial f(\mathbf{x}_j, \mathbf{u}_j)}{\partial \mathbf{u}_j}$ are calculated knowing the functional form of the dynamics of the vehicle. However, the derivative $\frac{d\mathbf{u}_j}{d\mathbf{x}_j}$ is of interest since the function relationship $u_j = F_N(\mathbf{x}_j)$ is governed by the neural network $\mathcal{N}$. Thus, we now need to compute the derivative of the output of the neural network $\mathcal{N}$ in terms of its input. However, doing so simply requires an application of chain rule layer by layer starting from the output layer all the way back to the input layer. Assuming that the activation functions used in the network $\mathcal{N}$ are differentiable, this calculation is straightforward. Note that $K_0 = \frac{d\mathbf{x}_0}{d\mathbf{x}_0} = I$.

Overall Algorithm

The key pitfall of gradient descent lies in the possibility of getting stuck in a local optimum wherein further gradient steps will make virtually no progress towards finding a counter example. Therefore, we mix the gradient descent steps with random search as follows: (a) choose a random starting state and perform up to $n_{\max}$ steps of gradient descent; (b) if a violation is obtained then append it to the list; (c) reinitialize to a new starting state $\mathbf{x}_0$ after $n_{\max}$ gradient descent steps. Algorithm 2 summarizes the overall approach. Note that N, in Algorithm 2 is some upper bound on the number of random restarts.

Algorithm 2 Hybrid Gradient and Stochastic Descent

```
function FalsificationSearch
    x_0 ← randomStateInInitialRegion()
    n ← 0
    for k ∈ [1, N] do
        (trajectory, violation) ← simulateSystem(x_0)
        if (not violation) and n < n_max then
            x_0 ← gradientDescent(trajectory, δ)
            n ← n + 1
        else
            if violation then
                append x_0 to counterexample list.
            x_0 ← randomStateInInitialRegion()
            n ← 0
```

Data Update

We briefly discuss the process of using counterexample traces to generate training data. Each counterexample trace is generated as a series of states $\mathbf{x}_0^{(i)}, \ldots, \mathbf{x}_j^{(i)}$. Clearly, the output of the current candidate neural network needs to be altered so as to avoid obtaining these counterexamples in the next iteration. To do so, we query the MPC on the states visited in the counterexample, choosing states $\mathbf{x}_j^{(i)}$ and the control $\mathbf{u}_{j,i}$ computed by the MPC whenever $\mathbf{u}_{j,i}$ is sufficiently far away from the output computed by the network $\mathcal{N}$. Furthermore, since the dynamics vary continuously with respect to the state and control, and the neural network feedback is continuous as well, sampling states close to those of the falsifying trajectory also yield falsifications.

As a result, at the k^{th} iteration the falsifying trajectories yield new data $\mathcal{F}_k$ that must be combined with the training data $\mathcal{D}_{k-1}$ used in the previous iteration to yield the training data $\mathcal{D}_k$. Since $\mathcal{F}_k$ focuses on counterexamples, training the network on just $\mathcal{F}_k$ introduces a bias wherein the network is likely to overfit to the current falsifications without retaining correct behaviors. Therefore, we accumulate the datasets at each iteration. First, we control the size of the new data $\mathcal{F}_k$ by subsampling so that we have precisely N_f samples. We simply take the union of the newly obtained data $\mathcal{F}_k$ to the previously available data $\mathcal{D}_{k-1}$ to obtain $\mathcal{D}_k$.

Experimental Results

We evaluate our approach on two benchmark models: a nonlinear dynamical model of a car that includes lateral dynamics and a quadrotor model.

Benchmark 1: Car Model

We use a standard "bicycle" model taken from (Manceur and Menhour 2013), as the model of our vehicle model, the equations of which are given below. Here, x_1 and x_2 are the coordinates (position), x_3 and x_4, are the heading angle and velocity respectively.

Table 1: Comparison of various counterexample generation schemes over neural network candidates sampled during the training process. **Legend** : #C: number counterexample traces found, T: time taken for search in seconds. G-Search:gradient descent search with uniform samples (Algorithm 2, S-Search: uniform samples without gradient descent, and D-Search: exhaustive search over discrete grid.

Autonomous Car						
	G-Search		S-Search		D-Search	
ID	#C	T	#C	T	#C	T
1	61	42	0	36	0	34
2	22	45	0	34	0	34
3	8	44	0	34	0	33
4	42	39	6	34	0	34
5	38	40	0	33	0	33
6	36	42	0	33	5	33
7	20	56	0	44	0	44
8	34	50	6	42	0	43
9	19	53	0	43	1	43
10	5	55	24	44	1	44
11	0	56	2	44	1	44
12	54	49	44	40	5	43
13	39	49	0	44	1	44

Quadrotor						
	G-Search		S-Search		D-Search	
ID	#C	T	#C	T	#C	T
1	22	126	0	129	-	-
2	39	115	0	131	-	-
3	26	123	0	131	-	-
4	45	117	0	131	-	-
5	44	113	0	130	-	-
6	47	111	0	129	-	-
7	15	128	0	130	-	-
8	35	116	0	128	-	-
9	24	124	0	128	-	-
10	7	131	0	129	-	-

$$\begin{aligned} \beta &= tan^{-1}(\tfrac{l_r}{l_r+l_f} \times tan(u_2)) \\ \dot{x_1} &= x_4 \times cos(x_3+\beta) \\ \dot{x_2} &= x_4 \times sin(x_3+\beta) \\ \dot{x_3} &= \tfrac{x_4}{l_r} \times sin(\beta), \dot{x_4} = u_1 \end{aligned} \tag{2}$$

The parameter values are taken to be $m = 2278kg$ and $F_d = 228N$. The control inputs u_1 and u_2 model the *acceleration*, and *steering angle*, respectively.

Benchmark 2 : Quadrotor Model

The quadrotor model is taken from (Luis and Ny 2016). Its state variables include the position (x, y, z) in an inertial frame, velocities $(\dot{x}, \dot{y}, \dot{z})$, attitude (ϕ, θ, ψ), and attitude rates $(\dot{\phi}, \dot{\theta}, \dot{\psi})$. The system has four control inputs: the thrust u_1 and the roll, pitch and yaw torques: u_2, u_3 and u_4. The

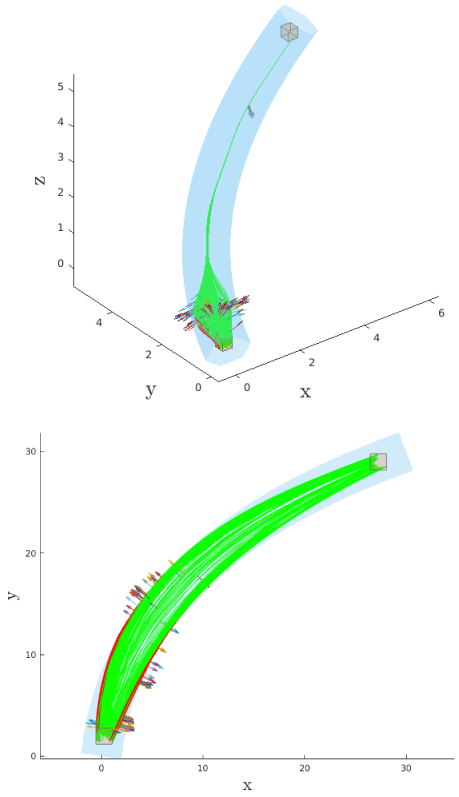

Figure 7: Sample trajectories shown during the training phase for the Quadrotor (top) and Car model (bottom). The region shaded in blue refers to the safety envelope of the trajectory, and the green trajectories satisfy the specification. Counterexamples are shown in red. The arrows pointing out show the desired directions for falsifications.

dynamics are described by an ODE:

$$\begin{aligned} \ddot{x} &= \tfrac{\theta u_1}{m} \\ \ddot{y} &= -\tfrac{\phi u_1}{m} \\ \ddot{z} &= \tfrac{\theta u_1}{m} - g \\ \ddot{\phi} &= \tfrac{u_2}{I_x} \\ \ddot{\theta} &= \tfrac{u_3}{I_y} \\ \ddot{\psi} &= \tfrac{u_4}{I_z} \end{aligned} \tag{3}$$

The parameters corresponding the above equation are $m = 3.3 \times 10^{-2} kg$, $I_x = 1.395 \times 10^{-5} kgm^2$, $I_y = 1.436 \times 10^{-5}$, $I_z = 2.173 \times 10^{-5} kgm^2$.

First, we study the effectiveness of our counterexample generation against two other strategies. The gradient-based search described in Algo 2 is run for 250 iterations and compared against 250 initial states sampled uniformly at random and an exhaustive search over a grid. Table 1, presents the results of this comparison over a series of neural network candidates obtained during runs of the overall iterative learning scheme. Note that for the car model (Benchmark 1), the number of subdivisions is $\mathsf{N}_{subd} = 4 \times 4 \times 4 \times 4$. However, discretizing the quadrotor model in a similar manner

Table 2: Trajectory Tracking, **Legend :** R refers to the maximum curvature of the trajectory,$\mathcal{N}_a$ lists number of neurons for each layer of the network with the list size providing the number of hidden layers, $\mathcal{R}_t$ refers to the number of iterations of the learning loop in Fig. 3. † denotes that the learning process did not terminate after 25iterations.

Autonomous Car				Quadrotor			
ID	R	$\mathcal{N}_a$	$\mathcal{R}_t$	ID	R	$\mathcal{N}_a$	$\mathcal{R}_t$
S_1	0.037	[6, 8]	3	S_1	0.13	[18]	2
	0.044	[7, 3, 3]	7		0.21	[16, 11, 10, 20]	2
	0.013	[6]	1		0.46	[13, 16]	5
	0.037	[9, 8, 6]	3		0.45	[12, 19]	4
	0.016	[8, 3, 3]	2		0.15	[10, 15, 20]	2
S_2	0.042	[5, 4]	3	S_2	0.75	[18, 18]	1
	0.043	[3, 6, 3]	†		0.39	[15, 17, 11]	2
	0.008	[3]	4		0.28	[15, 10, 13]	2
	0.06	[3, 8, 10]	†		0.23	[10, 18, 13]	3
	0.024	[9, 3]	1		0.03	[12, 20, 19]	2
S_3	0.06	[8]	17	S_3	0.20	[13, 18, 19]	2
	0.017	[3, 5]	†		0.25	[12, 13]	4
	0.057	[3, 5, 10]	21		0.38	[13, 11, 15, 13]	5
	0.021	[3, 3, 7]	†		0.13	[14, 18, 20, 20]	2
	0.043	[9, 7]	2		0.17	[19]	2
S_4	0.052	[4]	†	S_4	0.44	[17, 18]	3
	0.025	[9, 6]	2		0.33	[15, 15, 12, 17]	2
	0.043	[6, 3]	1		0.28	[10, 16, 13]	4
	0.009	[10]	1		0.07	[20]	2
	0.036	[7, 10]	2		1.52	[20, 16, 17, 13]	†
S_5	0.050	[3, 9]	12	S_5	0.17	[13, 12, 12]	2
	0.011	[10]	3		0.25	[15]	2
	0.043	[8, 5, 9]	2		0.34	[16, 10, 12]	2
	0.022	[3, 7, 5]	3		0.11	[20]	4
	0.033	[5, 10]	1		0.10	[12, 15, 19]	2

would yield upwards of 2^{24} cells and is thus not feasible. We note that our approach finds the the largest number of counterexamples for a majority of the evaluation instances. Furthermore, for all the ten instances of the quadrotor benchmark, uniform random search fails to find any falsifications, whereas gradient descent search discovers a considerable number of counterexamples. This demonstrates that gradient descent search combined with uniform samples over initial states can provide an effective approach for discovering counterexamples.

Next, we evaluate whether our approach can successfully learn a neural network model that meets our termination criterion: the falsification search does not find any counterexamples within a fixed number of iterations. For each of the benchmarks, we picked 5 sets of initial and final sets within the state space of the system. Next, for each initial/final set, we generated 5 reference trajectories at random recording the maximum curvature. This process yields 25 instances, in all, for each benchmark. We applied our approach on each instance and report the number of iterations if our approach was ultimately successful in learning a network, and the time taken. The structure of the network used was chosen randomly within a given range, lacking any insights into the ideal network topology, currently. The results are shown in Table 2. We initialized the training data with 100 and 200 samples for the quadrotor and car models, respectively. Each iteration added 50 and 100 counterexample samples, respectively for the quadrotor and car models. We note that learning concludes successfully in 20 out of 25 instances for the car model and 24 out of 25 instances, for the quadrotor model. For the majority of instances, the process converged rapidly within 5 iterations with just 4 out of 44 successful instances requiring more than 5 iterations. Figure 7 shows some of the sample trajectories obtained during the training along with counterexamples. Our experimental results did not further adjust the number of iterations or the choice of a neural network topology for the failing cases. This investigation will be completed and reported in an extended version of this paper.

Conclusion

In conclusion, we present a counterexample driven learning scheme that uses counterexample generation in conjunction with a MPC to successfully train neural networks for tracking reference trajectories robustly. We tested our approach against car and quadrotor models. Our evaluation focused on evaluating the impact of the gradident-based counterexample search strategy and the overall applicability with promising results. However, further work is needed to empirically study the impact of neural network topology on the suc-

cess of the learning. Using formal verification approaches to prove guarantees on the final result is yet another avenue of future work.

Acknowledgments: This work was supported in part by the Air Force Research Laboratory (AFRL) and by the US NSF under Award # 1646556.

References

Alur, R.; Bodik, R.; Juniwal, G.; Martin, M. M. K.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, 1–8.

Baier, C., and Katoen, J.-P. 2008. *Principles of Model Checking*. MIT Press.

Dutta, S.; Jha, S.; Sankaranarayanan, S.; and Tiwari, A. 2018. Learning and verification of feedback control systems using feedforward neural networks. *IFAC-PapersOnLine* 51(16):151 – 156. 6th IFAC Conference on Analysis and Design of Hybrid Systems ADHS 2018.

Frazzoli, E.; Dahleh, M. A.; and Feron, E. 2005. Maneuver-based motion planning for nonlinear systems with symmetries. *IEEE Transactions on Robotics* 21(6):1077–1091.

Goodfellow, I.; Bengio, Y.; and Courville, A. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Huang, S. H.; Papernot, N.; Goodfellow, I. J.; Duan, Y.; and Abbeel, P. 2017. Adversarial attacks on neural network policies. *CoRR* abs/1702.02284.

Kahn, G.; Zhang, T.; Levine, S.; and Abbeel, P. 2017. Plato: Policy learning using adaptive trajectory optimization. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 3342–3349.

Leofante, F.; Narodytska, N.; Pulina, L.; and Tacchella, A. 2018. Automated verification of neural networks: Advances, challenges and perspectives. *CoRR* abs/1805.09938.

Levine, S.; Finn, C.; Darrell, T.; and Abbeel, P. 2015. End-to-end training of deep visuomotor policies. *CoRR* abs/1504.00702.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2015. Continuous control with deep reinforcement learning. *CoRR* abs/1509.02971.

Luenberger, D. G. 1977. *Linear and Non-Linear Programming*. Addison-Wesley.

Luis, C., and Ny, J. L. 2016. Design of a trajectory tracking controller for a nanoquadcopter. *CoRR* abs/1608.05786.

Majumdar, A., and Tedrake, R. 2017. Funnel libraries for real-time robust feedback motion planning. *The International Journal of Robotics Research* 36(8):947–982.

Manceur, M., and Menhour, L. 2013. Higher order sliding mode controller for driving steering vehicle wheels: Tracking trajectory problem. In *52nd IEEE Conference on Decision and Control*, 3073–3078.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. A. 2013. Playing atari with deep reinforcement learning. *CoRR* abs/1312.5602.

Pereira, M.; Fan, D. D.; An, G. N.; and Theodorou, E. 2018. Mpc-inspired neural network policies for sequential decision making. *CoRR* abs/1802.05803.

Piche, S.; Sayyar-Rodsari, B.; Johnson, D.; and Gerules, M. 2000. Nonlinear model predictive control using neural networks. *IEEE Control Systems Magazine* 20(3):53–62.

Psichogios, D. C., and Ungar, L. H. 1991. Direct and indirect model based control using artificial neural networks. *Industrial & Engineering Chemistry Research* 30(12):2564–2573.

Ravanbakhsh, H., and Sankaranarayanan, S. 2017. Learning lyapunov (potential) functions from counterexamples and demonstrations. *CoRR* abs/1705.09619.

Ross, S.; Gordon, G. J.; and Bagnell, D. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *AISTATS*, volume 15 of *JMLR Proceedings*, 627–635. JMLR.org.

Sadigh, D.; Dragan, A. D.; Sastry, S.; and Seshia, S. A. 2017. Active preference-based learning of reward functions. In *Robotics: Science and Systems*.

Sutton, R. S., and Barto, A. G. 1998. *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1st edition.

Tuncali, C. E.; Kapinski, J.; Ito, H.; and Deshmukh, J. V. 2018. Reasoning about safety of learning-enabled components in autonomous cyber-physical systems. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, 30:1–30:6. New York, NY, USA: ACM.

Wirth, C.; Akrour, R.; Neumann, G.; and Fürnkranz, J. 2017. A survey of preference-based reinforcement learning methods. *Journal of Machine Learning Research* 18(136):1–46.

Xiang, W., and Johnson, T. T. 2018. Reachability analysis and safety verification for neural network control systems. *CoRR* abs/1805.09944.

Yaghoubi, S., and Fainekos, G. 2018. Gray-box adversarial testing for control systems with machine learning component. *CoRR* abs/1812.11958.