Counter-Example Guided Imitation Learning of Feedback Controllers from Temporal Logic Specifications

Thao Dang[1], Alexandre Donzé[2], Inzemamul Haque[3], Nikolaos Kekatos[4], Indranil Saha[3]

***Abstract*— We present a novel method for imitation learning for control requirements expressed using Signal Temporal Logic (STL). More concretely we focus on the problem of training a neural network to imitate a complex controller. The learning process is guided by efficient data aggregation based on counter-examples and a coverage measure. Moreover, we introduce a method to evaluate the performance of the learned controller via parameterization and parameter estimation of the STL requirements. We demonstrate our approach with a flying robot case study.**

I. Introduction

The aim of this work is to integrate formal specification and validation techniques in the Imitation Learning (IL) methodology for synthesizing feedback controllers for complex dynamical systems. While formal methods have the advantage of rigorous formalization and reasoning, they are very limited in the complexity and scalability of the problems that can be practically solved. Imitation Learning, also known as learning from demonstrations, involves the process of learning how to mimic the behavior of an expert by observing their actions in a given task [1]. It has many successful applications in various fields such as robotics, natural language processing, image and speech recognition.

In this work, we focus on the problem of training a neural network (NN) (playing the role of a learner) to imitate a complex controller (playing the role of an expert). The ultimate goal is to replace this complex controller with a trained NN. NNs have long been used to control dynamical systems from inverted pendulums to quadcopters, learning from scratch to control the plant by maximizing an expected reward, e.g. [2], [3]. NNs can also be trained to replace an existing controller that is unsatisfactory for non-functional reasons, e.g., computationally expensive (consider model-predictive control [4]), slow, or energy intensive. A well-trained NN controller can provide similar control performance much faster and is readily implemented on cheap and energy-efficient embedded platforms [5].

To make such an imitation learning framework more formal and more efficient, we add the following novel features: (i) a formalization of performance evaluation for both the learner's and expert's policies using their abilities to satisfy requirements specified by temporal logic, (ii) a leverage of the power of existing temporal logic property falsification tools to create training data that matter, (iii) a new method of data aggregation in order to guarantee a good performance of NN in terms of imitation and generalization.

To explain these features, let us first point out some major difficulties in this problem. Training a NN to imitate a feedback controller is more complex than the problem of approximating a function using pairs of input and output values, since feedback controllers can themselves be stateful dynamical systems. We identify the following difficulties in data generation by executing the nominal controller in closed loop:

- *Infinite behavior space.* The behavior space is not only large but can also be infinite. It is thus important to define a coverage measure to quantify how representative the generated training data is.
- *Non-uniform accuracy.* Depending on the control requirement, the NN may need to be very precise around some region of the state space while in other regions a rough approximation is acceptable.

The problem of non-uniform accuracy is particularly pronounced when the requirement depends on time or sequences of events. This is frequently the case in control applications, where properties such as *rise time*, *settling time*, and *overshoot* are typical. We consider complex properties that can include not only time but also causal relationships. They can be described in *Signal Temporal Logic* (STL), a formal language that finds widespread use in formal methods and increasing adoption in industry [6].

While observing closed-loop behaviors may reduce the number of behaviors to be sampled, we still need to find good training samples that are relevant to an STL property. To do this, we find counter-examples that are closed-loop behaviors violating this property by leveraging the existing falsification tools [7]. A falsification process can also be useful in providing correctness guarantees for the resulting NN. Indeed, if no counter-example is found after a sufficiently large number of scenarios, we consider the NN controller satisfactory and stop. If a counter-example is found, we replay the nominal controller from the counter-example situation in order to obtain new training data, and retrain the neural network. This new data creation is crucial for the efficiency of the process of correcting counter-examples as well as assuring good generalization of the NN.

Related Work. Several comprehensive surveys exist [8]

* This work is partially supported by the Indo-French Collaborative research project FOVERAS funded by IFCPAR/CEFIPRA, joint French-Japanese ANR-JST project CyPhAI, and the Auvergne-Rhône-Alpes Region Project DetAI.

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, Grenoble, France, thao.dang@univ-grenoble-alpes.fr

[2] Decyphir SAS, Moirans, France, alex@decyphir.com

[3] Department of Computer Science and Engineering, IIT Kanpur, India, {inzemam, isaha}@cse.iitk.ac.in

[4] Aristotle University of Thessaloniki, Thessaloniki, Greece, nkekatos@csd.auth.gr

on imitation learning. A common approach is behavioral cloning [9] for which the main problem is *compounding error*, when the assumption of independent and identically distributed data between training and testing data is not valid for sequential predictions. A common solution to this problem is Dataset Aggregation (DAgger) which was proposed in [10]. It is an iterative algorithm which improves on behavioral cloning by training on a dataset that better resembles the observations the trained policy is likely to encounter. In this paper, we propose a similar approach in the sense that the expert is queried for good actions and new data are created and aggregated in the dataset. However, our goal is to design an approach that learns the dynamics of *desirable* behavior and not the dynamics of *possible* behavior. Applications of imitation learning for Model Predictive Control have been proposed in [11]. A recent approach to "compress" an MPC into a NN using robust tube MPC is proposed in [12]. In [13], the need to replace an MPC controller with a NN is exemplified for planning purposes. Our approach is different from the previous works as we use counter-example guided synthesis and a combination of coverage and PSTL formal specifications. There are counter-example-based approaches similar to ours though - for instance, some use counter-example exploration (adversarial sample) to train NNs that seek to satisfy a given property expressed in temporal logic (see [14]) or through a reference trajectory (see [15]). The closest to our work is [15], where the behavior of an MPC is approximated with a NN to enable a robot to follow a reference trajectory. The NN is refined by generating additional training data from counter-examples. However, unlike our framework, their approach is limited to memoryless controllers, and they use tracking closeness as the sole criterion to identify counter-examples. In our experience, a NN need not always closely track the nominal behavior to satisfy the specifications. It may have some characteristics that are better than the nominal controller, such as a smaller overshoot, or quicker stabilization in some areas of the state space. Such behaviors would be eliminated using their approach.

The rest of the paper is organized as follows. In Section II, we formalize the imitation learning problem. This requires definitions of control requirements specified using Parametric Signal Temporal Logic. We also propose a notion of policy performance to quantify the difference between the policy of the learner and the expert which is necessary to assess the imitation quality. Subsequently, in Section III and Section IV, we describe our dataset aggregation-based learning methodology. Finally, we demonstrate our approach on a robotic case study in Section V.

II. Controller Imitation Learning Problem

We consider a continuous-time plant S with state $x \in \mathbb{R}^n$ that is controlled by an input signal u, observed through an output signal y with dynamics[1]: $\dot{x}(t) = f(x(t), u(t))$ and $y(t) = \zeta(x(t))$. The control input u is computed by a discrete-time controller $\mathscr{C}$ with state $z \in \mathbb{R}^{n_z}$ and $z_{k+1} = f_c(z_k, y_k)$, $u_k = \upsilon(z_k, y_k)$, where y_k is a discrete-time signal resulting from sampling the output $y(\cdot)$ with time step h. The continuous-time control $u(\cdot)$ is a piece-wise constant function defined as $\forall t \in [kh, (k+1)h]$ $u(t) = u_k$ with $k = 0, 1, \ldots$. We denote the closed-loop system by $\mathscr{C} || S$.

[1]External disturbance can be included in the system dynamics, and the data generation process for learning can then be straightforwardly extended to cover the disturbance space.

A. Control Requirements

We are given a nominal controller satisfying some requirement that captures essential qualitative properties while allowing some quantitative behavioral flexibility. This requirement is expressed using *Parametric Signal Temporal Logic (PSTL)*, a formalism suitable for various control performance properties [6]. We want to train a neural network-based controller achieving performance comparable to that of the nominal controller, as measured by *valid* parameters of the PSTL requirement.

1) Signal Temporal Logic [6]: An STL formula φ consists of atomic predicates along with logical and temporal operators. Atomic predicates are defined over signal values and have the form $g(y(t)) \sim 0$, where g is a scalar-valued function over the signal y evaluated at time t and $\sim \in \{<, \leq, >, \geq, =, \neq\}$. Temporal operators "always" ($\Box$), "eventually" ($\Diamond$), and "until" ($\mathscr{U}$) have the usual meaning and are scoped using intervals of the form (a,b), $(a,b]$, $[a,b)$, $[a,b]$, or (a,∞), where $a, b \in \mathbb{R}_0^+$ and $a < b$. If I is a time interval, the following grammar defines the STL language.

$$\varphi \ := \ \top \mid g(y(t)) \sim 0 \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathscr{U}_I \varphi_2 \qquad (1)$$

The $\Diamond$ operator is formally defined as $\Diamond_I \varphi \triangleq \top \mathscr{U}_I \varphi$, and the $\Box$ operator is defined as $\Box_I \varphi \triangleq \neg(\Diamond_I \neg\varphi)$. When omitted, the interval I is taken to be $[0, \infty)$. The "always" operator in $\Box\varphi$ conveys that from the current time point onwards, φ always holds. The "eventually" operator in $\Diamond\varphi$ means that there exists a time point in the future where φ holds. The "until" operator in $\varphi_1 \mathscr{U} \varphi_2$ means that, starting from the current time point φ_1 should hold continuously until a future time point where φ_2 holds. Additionally, an interval I can be combined with the operators to bound their scope to a segment of time in the future rather than the whole. For example, $\varphi_1 \mathscr{U}_{[a,b]} \varphi_2$ holds true at time t if φ_2 holds true at some point $t' \in [t+a, t+b]$ and φ_1 is always true in $[t, t']$.

2) Parametric Signal Temporal Logic [16]: Parametric STL (PSTL) is a variant of STL which makes it possible to replace numeric constants in an STL formula with symbolic variables or parameters. For instance, the formula $\varphi = \Box_{[0,\tau]}(\|y(t)\| < s)$ with two parameters τ and s expresses the requirement that during τ seconds, the norm of signal y should be less than s. Formally, a PSTL formula is concretized into a STL formula by composing it with a valuation, defined as a mapping from symbolic parameters to reals. As an example, consider valuation $p : \{\tau \to 2, s \to 10\}$, then $\varphi(p)$ is the STL formula $\varphi(p) = \Box_{[0,2]}(\|y(t)\| < 10)$.

Given a PSTL formula φ and a valuation p, a behavior γ satisfies $\varphi(p)$ is denoted by $\gamma \models \varphi(p)$. By extension, we say that a valuation p is satisfied by a controller $\mathscr{C}$ if for all behaviors γ of the closed-loop system $\mathscr{C}||S$, $\gamma \models \varphi(p)$.

The problem of finding a valuation p such that $\mathscr{C}$ satisfies $\varphi(p)$ is sometimes called mining and can be seen (and solved) as a learning problem. In [17], an approach is presented that works for a *monotonic* PSTL formula. Intuitively a formula is monotonic if its satisfaction is monotonic *w.r.t.* the valuation of each individual parameter. For example, $\varphi = \Box_{[0,\tau]}(\|y(t)\| < s)$ is monotonic because if $\|y(t)\|$ is always smaller than s between time 0 and τ, then clearly $\|y(t)\|$ is smaller than s' for any $s' > s$ and it is also smaller than s between time 0 and τ' for $\tau' \leq \tau$. Formally, the set $\mathscr{P}$ of all valuations is a subset of $\mathbb{R}^{n_p}$, where n_p is the number of parameters in the formula. Let $\preceq$ be the standard partial order on $\mathbb{R}^{n_p}$, *i.e.*, $p = (p_1, \dots, p_{n_p}) \preceq p' = (p'_1, \dots, p'_{n_p})$ iff $\forall i\ p_i \leq p'_i$, then if $\forall \gamma$ the Boolean function $p \rightarrow \gamma \models \varphi(p)$ is monotonic in p, then the PSTL formula φ is monotonic. For illustration purposes, to characterize our controller performance, the PSTL formula Φ that we will use is as follows:

$$\mu_{\text{ov}} := \|y(t)\| > s_{\text{ov}},\ \mu_{\text{st}} := \|y(t)\| < s_{\text{st}} \tag{2}$$

$$\varphi_{\text{st}} := \neg\mu_{\text{st}} \Rightarrow \Diamond_{[0,\tau_{\text{tr}}]}\Box_{[0,\tau_{\text{st}}]}\mu_{\text{st}} \tag{3}$$

$$\Phi := \Box\neg\mu_{\text{ov}} \wedge \Box\varphi_{\text{st}} \tag{4}$$

Equation (2) defines the atomic predicates which check at a given time whether the signal norm is above s_{ov} (overshoot), or below s_{st} (defining a stabilization region around the equilibrium). Equation (3) defines a formula requiring that if the system is not stabilizing (μ_{st} not satisfied), then it should eventually stabilize, *i.e.*, after at most τ_{tr} seconds, μ_{st} should remain true for at least τ_{st} seconds.

We can then see that Φ is monotonic. This follows from the monotonicity of atomic predicates and temporal operators $\Box$ and $\Diamond$ and the fact that each parameter appears only once in each sub-formula [16]. Monotonicity here is advantageous not only for the mining problem but also for computing a performance measure defined by Pareto fronts as discussed in the next section.

B. *Control Policy Performance Measure*

In imitation learning, it is essential to have an appropriate measure of performance of control policies, especially when it is unclear what reward function is being optimized [18]. In our framework, we use the relation between the parameters in the PSTL requirements to compare the performance of different controllers. E.g., for the stabilization requirement (4), for a given size s_{st} of the neighborhood around the equilibrium, the smaller the stabilization time τ_{st} is, the faster the controller is. Assume that $\mathscr{P}(\Phi)$ (set of parameter valuations for Φ) is compact. Any controller $\mathscr{C}$ defines a partition of this set into falsified and valid formulas:

$$\mathscr{P}(\Phi) = \text{False}(\mathscr{C}||S, \Phi) \cup \text{Valid}(\mathscr{C}||S, \Phi)$$

The False and Valid sets are separated by the set of *Pareto-efficient* parameter values, also called the *Pareto front* [19]; that means no parameter can be improved without compromising the others. We argue that the relative volumes of the False (and Valid) sets can be used to measure and compare performance of controllers. More specifically, we define the following measure of similarity:

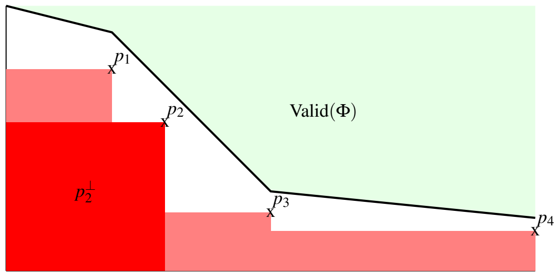

Fig. 1. Volume estimation of False parameter set. Each p_i outside the Valid set is close to the Pareto front and defines a closed hyper-box $p_i^{\perp}$ strictly included in False(Φ). Computing the volume of $\cup_i p_i^{\perp}$ yields an under-approximation of vol(False(Φ).

Definition 1 (Control Policy Similarity): Given a plant S and two controllers $\mathscr{C}$ and $\mathscr{C}'$ designed to satisfy a PSTL requirement Φ, the performance similarity between $\mathscr{C}$ and $\mathscr{C}'$ is defined as $\sigma_{S,\Phi}(\mathscr{C},\mathscr{C}') = \frac{vol(\text{False}(\mathscr{C}'||S,\Phi)}{vol(\text{False}(\mathscr{C}||S,\Phi)}$ where vol is the volume of a set, assumed to be non-zero for False($\mathscr{C}||S,\Phi$). Exact volume computation is difficult in general but monotonicity makes it easy to compute under-approximations. Indeed, consider a finite set of parameters $p_1, \dots, p_k$ in False($\mathscr{C}||S,\Phi$) and for each p_i, define $p_i^{\perp}$ the set of parameters dominated by p_i according to $\preceq$. Intuitively, $p_i^{\perp}$ is a hyper-box with largest corner p_i. Then $\bigcup_i p_i^{\perp} \subset$ False($\mathscr{C}||S,\Phi$) and $\sum_i vol(p_i^{\perp}) \leq vol(\text{False}(\mathscr{C}||S,\Phi))$. Since $p_i^{\perp}$ is an hyperbox, its volume computation is trivial. This approximation is illustrated in Figure 1.

C. *Imitation Learning Problem Formulation*

Problem 1 (Feedback Controller Imitation Learning): Given a plant S, a nominal controller $\mathscr{C}$ such that the closed-loop system $\mathscr{C}||S$ satisfies a PSTL specification Φ, our problem is to learn a neural network controller $\mathscr{N}$ to imitate $\mathscr{C}$ such that

- the closed-loop system $\mathscr{N}||S$ satisfies Φ, and
- the performance similarity $\sigma_{S,\Phi}(\mathscr{C},\mathscr{N})$ is as small as possible.

The learning guidance here is provided using positive examples, i.e. good behaviors, generated by the nominal controller which already satisfies the desired requirement. As we will see later, using parametric requirements allows more freedom in choosing nominal controllers satisfying some (minimal) performance, however to estimate the controller similarity in the learning process we estimate the actual Pareto-efficient parameters satisfied by a concrete nominal controller. Such a nominal controller may be complex and costly to execute and the ultimate goal is thus to use the learned NN controller to replace it. In the learning context, the nominal controller plays the role of a teacher that generates a desired control signal for a given system state which the NN should imitate.

In this paper, we mostly focus on the problem of how to iteratively train NN using examples (good behaviors) and counter-examples (bad behaviors).

III. Feedback Controller Learning Methodology

We will explain our approach for solving Problem 1 via an example that uses the PSTL formula Φ defined in (4). The major steps of our approach are as follows. Since initially we do not know the concrete performance of the nominal controller, we can assume that we conservatively choose a parameter valuation $\overline{p}$ so that $\Phi(\overline{p})$ is satisfied by the nominal controller. We train a NN controller satisfying $\Phi(\overline{p})$ using an iterative neural network training approach which is detailed in the subsequent subsections.

We generate and accumulate the traces generated by both controllers during the process. We use these traces to approximate their False domain. Then, we compute their volumes to estimate their policy similarity. Retraining is needed if this similarity is not as small as desired. To create data for such retraining, a finer grid can be used.

A. Neural Network Structure and Training

Intuitively, the NN controller is trained based on the good closed-loop behaviors we want it to learn. Let $\gamma = (x(\cdot), u(\cdot), y(\cdot))$ be a good closed-loop behavior, from which we extract the data of the form $d_\gamma = \{(x_k, y_k, u_k) \mid k < K\}$ where x_k, y_k, and u_k are respectively the state, output, and control values at time t_k; K is the discrete-time horizon (that is, the number of sampled time points). When many behaviors are considered, the data set $\mathscr{D}$ is the union of all d_γ. We generate a neural network $\mathscr{N}$ to fit the data set $\mathscr{D}$. The structure of the NN should capture the input-output relationships of the nominal controller $\mathscr{C}$. We keep some past values to represent the memory needed to compute the output at each discrete step. The input of the NN is $(y_{k-1}, \ldots, y_{k-m_y}, u_{k-1}, \ldots, u_{k-m_u})$ where $n_z = m_y + m_u$ represents the dimension of the state variable of the controller. The NN output is u_k. To train the NN, we use a loss function defined via the Root Mean Square Error (RMSE): $\sqrt{\frac{1}{n_d}\sum_{k=1}^{n_d}(\|u_k - \overline{u}_k\|)^2}$ where u is the output of the nominal controller and $\overline{u}$ is the output of the NN. It is possible to use other loss functions such as MSE (Mean Square Error); in our experiments so far the MRSE metric is more convenient in terms of interpreting the effects of control input error. A data point is a pair of input and output values, the total number n_d of data points is the number of data points per system behavior multiplied by the number of behaviors. The NN accuracy is defined by the *training* and *validation errors*, which are obtained from evaluating the loss function on the training and validation data.

B. Coverage based Data Generation

In order to achieve a robust NN controller, we need to provide data representing diverse settings that the NN should learn to cope with. Note that the set of reachable states of the system is infinite, we thus use a coverage measure based on ε-net [20] to quantify how well a finite set of sampled states covers the reachable set.

Algorithm 1 Dataset aggregation-based training algorithm.

```
1: $\mathscr{N}_0 \leftarrow \emptyset$, $\mathscr{D}_0 \leftarrow \emptyset$, $k \leftarrow 1$
2: repeat
3:     ($\mathscr{D}_k$, Status) $\leftarrow$ getNewData ($\mathscr{N}_{k-1}, \mathscr{D}_{k-1}$)
4:     if $\mathscr{D}_k \neq \mathscr{D}_{k-1}$ then          ▷ In this case $\mathscr{D}_{k-1} \subset \mathscr{D}_k$
5:         $\mathscr{N}_k \leftarrow$ Train ($\mathscr{D}_k$)
6:     $k \leftarrow k+1$
7: until $\mathscr{D}_k = \mathscr{D}_{k-1}$ or $k > k_{max}$
8: return $\mathscr{N}_k$, Status
```

We propose a simple grid-based method to construct ε-nets satisfying a separation requirement. These notions of ε-net and ε-separated sets are important to measure function approximation quality, expressed roughly as how to obtain an accurate function approximation with a small number of function evaluations.

Let us assume that the state space is a box $B_r = [\underline{r}, \overline{r}]^n$. We use a grid $\mathscr{G}$ to partition B_r into a set G of rectangular cells with equal side length 2ε, assuming for simplicity that $(\overline{r} - \underline{r})/(2\varepsilon)$ is an integer. The set C of center points of all the cells in G is an ε-net of the state space. It can be proved that it is an ε-net with minimal cardinality and additionally a ε-separated set with maximal cardinality[2].

Datasets constructed using ε-nets guarantee ε coverage. However, as we shall see later, the falsification procedure (to check if a neural network satisfies the requirement) uses optimization-based search algorithms which can explore many states outside the ε-nets. To estimate a coverage measure that better reflects the portion of the tested behaviors, we can use a finer grid $\mathscr{G}_c$ and take the ratio between the number of cells visited by both the sampling and falsification procedures and the total number of cells in this grid $\mathscr{G}_c$.

IV. Dataset Aggregation-based Training

The top-level iterative training algorithm implements a data aggregation method, inspired by [10], which uses the current NN to generate new data (from examples and counter-examples) that will be aggregated to the whole dataset to train a new NN. This dataset aggregation method considers the coverage and separation measures discussed in the previous section, to enable the NN to generalize well and provide high confidence in the correctness of the final NN.

The dataset aggregation algorithm (Algorithm 1) determines whether it should generate new data for retraining or stop. It stops after a maximum number of iterations k_{max} or as soon as the procedure responsible for providing new training samples fails to do so.

The training procedure is successful if it terminates with $\mathscr{D}_k = \mathscr{D}_{k-1}$, *i.e.*, no new data is found, and the *Status* returned by `GetNewData` is "No counter-example found". This `GetNewData` procedure is described in Algorithm 2.

[2]This cardinality determines the ε-entropy and ε-capacity of the state space [20].

Algorithm 2 New data acquisition procedure.

```
1: procedure GETNEWDATA($\mathscr{D}$, $\mathscr{N}$)
2:     if $\mathscr{D}$ is $\emptyset$ then                      ▷ Sample traces from initial set
3:         InitSamples ← getInitSamples()
4:         InitTraces ← simNominal(InitSamples)
5:         $\mathscr{D}_{new}$ ← gridFilter(InitTraces)
6:         Status ← "New data available for training."
7:     else                                                ▷ Search for counter-examples
8:         CexTraces ← falsify($\mathscr{N}$)
9:         if CexTraces $\neq \emptyset$ then
10:            ($\mathscr{D}_{new}$, Status) ← fixAndMerge($\mathscr{D}$, CexTraces)
11:        else                                            ▷ Success.
12:            $\mathscr{D}_{new}$ ← $\mathscr{D}$
13:            Status ← "No counter-example found."
```

Algorithm 3 Augmenting the training data from bad traces.

```
1: procedure FIXANDMERGE($\mathscr{D}$, CexTraces)
2:     NeighSamples ← gridFilter(CexTraces)$\setminus$Cover($\mathscr{D}$)
3:     if NeighSamples $= \emptyset$ then
4:         Status ← "Counter-examples do not add new data."
5:         $\mathscr{D}_{new}$ ← $\mathscr{D}$
6:     else                                    ▷ Get fixed traces from some bad samples
7:         NeighSamples ← selectNeigh(NeighSamples)
8:         FixedTraces ← simNominal(NeighSamples)
9:         $\mathscr{D}_{new}$ ← gridFilter($\mathscr{D}$ ∪ FixedTraces)
10:        Status ←"New data available for training."
```

In the first call of Algorithm 2, when no neural network has been trained yet and the data set is empty, it samples states using an ε-net over the initial set via the `getInitSamples` function and computes nominal traces from these initial states, that is traces generated by the nominal controller via the `simNominal` function. The `gridFilter` function collects samples from these traces and ensures that only one sample per grid cell is kept so that the data remains ε-separated. Indeed the `gridFilter` function "filters" the sampled states by removing the newly sampled states from the cells that are covered by existing data already. A subset of the remaining samples is then used to compute new nominal traces, producing new data. This step is illustrated in Fig. 2. At the subsequent calls, new data is obtained by testing the current neural network $\mathscr{N}_k$, looking for traces that violate the requirement using falsification (`falsify` function). The procedure `fixAndMerge` described in Algorithm 3 is called to produce new nominal traces and add data from these new nominal traces, in order to "fix" these counter-examples. Again, to keep samples well separated, new nominal traces are generated only from the samples resulting after applying `gridFilter` on the bad traces (*CexTraces* - Line 2). We also remove samples that are already covered by $\mathscr{D}$ (*Cover*($\mathscr{D}$)), represented as blue cells in Fig. 2).

Furthermore, in `selectNeigh` the samples close to the cells which are already covered have higher priority to be selected, which helps the neural net to generalize more easily. This can be seen in Fig. 2 where the green cells (containing new data selected for retraining) are close to the blue cells (containing current training data).

Algorithm 1 stops either when no new data is returned by Algorithm 2 or when k_{max} is reached. In the first case (no new data), there are two situations:

- "No counter-example found": Successful case. The falsification procedure has failed to falsify the neural network controller, providing our strongest evidence that we have obtained a correct controller with test coverage at least ε.
- "Counter-examples do not add new data.". This means that the grid is likely too coarse and the counter-examples do not allow visiting new cells.

If the maximum number of iterations is reached, the two statuses above are still possible (both stopping conditions are met), otherwise, the status returned is: "New data available for training." which indicates that the user can resume the process using the data aggregated so far.

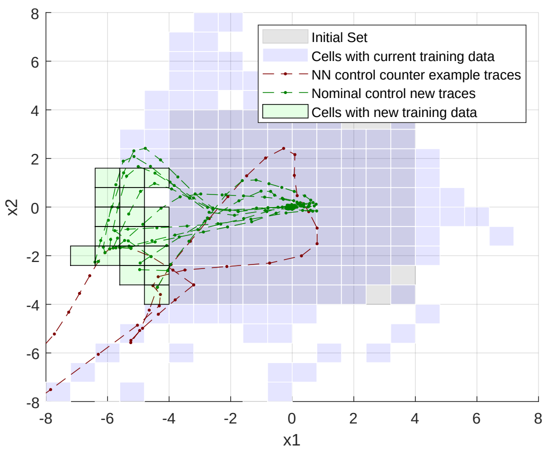

Fig. 2. One iteration of the learning algorithm.

V. Flying Robot Case Study

This example is taken from Mathworks [21] and describes a model of a flying robot that is driven by a nonlinear model predictive controller. The flying robot has two thrusters to move it in a 2-D space. The state of the flying robot, denoted by x, consists of six components: x_1, x_2 (horizontal and vertical coordinate) θ (robot thrust direction), and their derivatives $\dot{x}_1$, $\dot{x}_2$ and $\dot{\theta}$. The thrusts are represented by $u = (u_1, u_2)$. The dynamics of the robot are given by $\ddot{x_1} = (u_1 + u_2) \cdot \cos(\theta)$, $\ddot{x_2} = (u_1 + u_2) \cdot \sin(\theta)$ and $\ddot{\theta} = \alpha \cdot u_1 + \beta \cdot u_2$. The parameters are $\alpha = 0.2$ and $\beta = 0.2$. For each thrust, there is an operating range $[-u_{max}, u_{max}]$ with $u_{max} = 3$. Considering the following set X_0 of initial conditions

$$X_0 = \{ \quad \big(x_1(0), x_2(0), \theta(0), \dot{x}_1(0), \dot{x}_2(0), \dot{\theta}(0)\big) \ \ s.t. \\ x_1(0), x_2(0) \in [3.8, 3.8], \theta(0) \in [-2.5, 2.5], \\ \dot{x}_1(0), \dot{x}_2(0) \in [-1.6, 1.6], \dot{\theta}(0) \in [-0.8, 0.8] \},$$

the goal of the control input is to drive the flying robot from any state in X_0 to a region close to the origin and maintain it in this region indefinitely. We define y as $y(t) = \|x(t)\|$, so that the problem becomes to stabilize y close to 0. The nominal controller is provided by a model predictive control (MPC) scheme which computes inputs at each step that

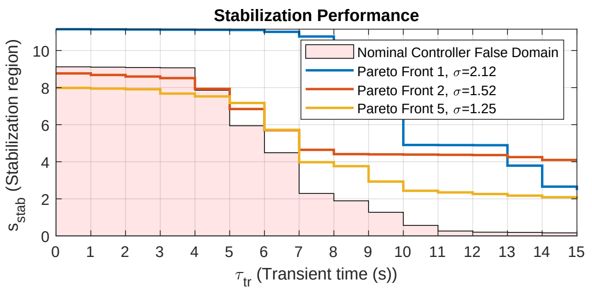

Fig. 3. Result of training NN controllers for the flying robot. Good performance is obtained after only 5 iterations. The red region (Nominal Controller False Domain) represents the valuations p for which $\Phi(p)$ are not satisfied by the nominal controller. The boundary of this region represents the Pareto front of the nominal controller. Other plots represent the Pareto fronts for several instances of the NN controllers computed for different iterations. Similarity with the nominal MPC controller is indicated in the legend.

minimizes $y(t)$ over a given horizon. The simulation time is 15 seconds. To evaluate the performance of a controller for this problem, we used the PSTL formula Φ defined in Equation 4, where we set τ_{st} to $+\infty$ and considered overshoot s_{ov}, transient time τ_{tr} and stabilization region s_{st} as variables to measure. We applied our algorithm with the STL formula $\Phi(\overline{p})$ with $\overline{p} = \{s_{ov} \to 15, \tau_{tr} \to 14, s_{st} \to 2\}$.

The inputs of NNs we trained correspond to the state variables of the system and the previous control input variables. They have RELU as activation functions, 6 hidden layers (each with 256 neurons), one scaling layer and one output layer. Figure 3 shows the performance evaluations of the nominal controller and a sample of different trained NN controllers obtained after 6 iterations, computed in around 120 minutes of computation time on a standard PC with an Intel Core i7 10700 processor and 64 Go of memory. After only two iterations, the NN controller manages to improve on the overshoot. However, looking at the False domain for (τ_{st}, τ_{tr}), we can see that the stabilization region is larger than that of the MPC controller, meaning that neural networks have trouble stabilizing the robot close to the origin due to the inherent instability of the system at this state.

VI. Conclusion

In this paper, we have presented a framework for efficiently training a neural network-based controller by imitation learning using a dataset aggregation approach with several novel aspects. Most notably, the collection of data from the nominal (expert) controller is done at states where the trained controller caused the plant to fail according to a specification expressed in Signal Temporal Logic. Moreover, the same specification is parameterized and can be used to evaluate the performance of the trained controller and how far or how differently it behaves with respect to the nominal controller. The method was evaluated on a nonlinear robotic system with promising results. Further experiments will be conducted and different practical and theoretical questions remain to be explored but we believe that this work represents an interesting step in the direction of safer and more efficient imitation learning methods of complex control systems.

References

[1] S. Schaal, "Learning from demonstration," in *Advances in Neural Information Processing Systems*, 1996.

[2] M. T. Hagan, H. B. Demuth, and O. D. Jesús, "An introduction to the use of neural networks in control systems," *International Journal of Robust and Nonlinear Control: IFAC-Affiliated Journal*, vol. 12, no. 11, pp. 959–985, 2002.

[3] C. Nicol, C. J. B. Macnab, and A. Ramirez-Serrano, "Robust neural network control of a quadrotor helicopter," in *Canadian Conference on Electrical and Computer Engineering*, 2008, pp. 1233–1238.

[4] C. E. Garcia, D. M. Prett, and M. Morari, "Model predictive control: Theory and practice—a survey," *Automatica*, vol. 25, no. 3, pp. 335–348, 1989.

[5] P. Varshney, G. Nagar, and I. Saha, "Deepcontrol: Energy-efficient control of a quadrotor using a deep neural network," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2019, pp. 43–50.

[6] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, "Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications," in *Lectures on Runtime Verification*, 2018, pp. 135–175.

[7] A. Donzé, "Breach, A toolbox for verification and parameter synthesis of hybrid systems," in *Computer Aided Verification, 22nd International Conference*. Springer, 2010, pp. 167–170.

[8] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, J. Peters *et al.*, "An algorithmic perspective on imitation learning," *Foundations and Trends in Robotics*, vol. 7, no. 1-2, pp. 1–179, 2018.

[9] D. Michie, M. Bain, and J. Hayes-Miches, "Cognitive models from subcognitive skills," *IEE control engineering series*, vol. 44, pp. 71–99, 1990.

[10] S. Ross, G. Gordon, and D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," in *International conference on artificial intelligence and statistics*, 2011, pp. 627–635.

[11] F. S. Acerbo, M. Alirczaei, H. Van der Auweraer, and T. D. Son, "Safe imitation learning on real-life highway data for human-like autonomous driving," in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2021, pp. 3903–3908.

[12] A. Tagliabue, D.-K. Kim, M. Everett, and J. P. How, "Efficient guided policy search via imitation of robust tube MPC," in *International Conference on Robotics and Automation*, 2022, pp. 462–468.

[13] S. Chow, D. Chang, and G. A. Hollinger, "Parallelized control-aware motion planning with learned controller proxies," *IEEE Robotics and Automation Letters*, 2023.

[14] S. Yaghoubi and G. Fainekos, "Worst-case Satisfaction of STL Specifications using Feedforward Neural Network Controllers: A Lagrange Multipliers Approach," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–20, 2019.

[15] A. Clavière, S. Dutta, and S. Sankaranarayanan, "Trajectory tracking control for robotic vehicles using counterexample guided training of neural networks," in *International Conference on Automated Planning and Scheduling*, 2019, pp. 680–688.

[16] E. Asarin, A. Donzé, O. Maler, and D. Nickovic, "Parametric Identification of Temporal Properties," in *Runtime Verification - Second International Conference*, 2011, pp. 147–160.

[17] X. Jin, A. Donzé, J. V. Deshmukh, and S. A. Seshia, "Mining Requirements From Closed-Loop Control Models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 11, pp. 1704–1717, 2015.

[18] T. Osa, J. Pajarinen, G. Neumann, J. A. Bagnell, P. Abbeel, and J. Peters, "An algorithmic perspective on imitation learning," *CoRR*, vol. abs/1811.06711, 2018.

[19] V. Pareto, "Manuel d'é conomie politique," *Bull. Amer. Math. Soc*, vol. 18, p. 462–474, 1912.

[20] A. N. Kolmogorov and V. M. Tikhomirov, "ε-entropy and ε-capacity of sets in function spaces," *Uspekhi Matematicheskikh Nauk*, vol. 14, no. 2, pp. 3–86, 1959.

[21] Mathworks, "Imitate nonlinear MPC controller for flying robot," [Online] Available at https://in.mathworks.com/help/reinforcement-learning/ug/imitate-nonlinear-mpc-controller-for-flying-robot.html.