Interpretable Control by Reinforcement Learning

Daniel Hein, Steffen Limmer, Thomas A. Runkler

Siemens AG, Corporate Technology, Otto-Hahn-Ring 6, 81739 Munich, Germany (e-mail: hein.daniel@siemens.com).

Abstract: In this paper, three recently introduced reinforcement learning (RL) methods are used to generate human-interpretable policies for the cart-pole balancing benchmark. The novel RL methods learn human-interpretable policies in the form of compact fuzzy controllers and simple algebraic equations. The representations as well as the achieved control performances are compared with two classical controller design methods and three non-interpretable RL methods. All eight methods utilize the same previously generated data batch and produce their controller offline - without interaction with the real benchmark dynamics. The experiments show that the novel RL methods are able to automatically generate well-performing policies which are at the same time human-interpretable. Furthermore, one of the methods is applied to automatically learn an equation-based policy for a hardware cart-pole demonstrator by using only human-player-generated batch data. The solution generated in the first attempt already represents a successful balancing policy, which demonstrates the methods applicability to real-world problems.

Keywords: Human supervised control, learning control, LQR, PID, fuzzy control.

1. INTRODUCTION

The search for interpretable reinforcement learning (RL) policies is of high academic and industrial interest. Interpretability is defined as the ability to explain or to present in understandable terms to a human. In the context of machine learning (ML), Doshi-Velez and Kim argue that the need for interpretability stems from an incompleteness in the problem formalization, creating a fundamental barrier to optimization and evaluation (Doshi-Velez and Kim, 2017). Since complex real-world tasks in industry are almost never completely testable, enumerating all possible outputs given all possible inputs is infeasible. Hence, we usually are unable to flag all undesirable outputs. Especially for industrial systems, domain experts are more likely to deploy autonomously learned controllers if they are understandable and convenient to assess. Moreover, legal frameworks such as the European Union's General Data Protection Regulation (EU, 2016) enforce interpretability of data processing systems.

In the context of this work, interpretability is evaluated via a quantifiable proxy. At first, a model class, e.g., fuzzy rules, algebraic equations, or high-level task descriptions, is claimed of being interpretable, and subsequently algorithms are developed in order to optimize within this class.

Since controllers from classical control theory, like proportional-integral-derivative (PID) controllers, traditionally have been parameterized by iterating response observation and adapting the respective parameters, they

Fig. 1. Hardware cart-pole system

are generally designed to be interpretable for humans. For this reason, PID control is still a very important and common strategy in order to realize stable, efficient, and understandable control strategies in industry applications. In 2002, Desborough and Miller conducted a survey of more than 11 000 controllers in the refining, chemicals, and pulp and paper industries which revealed that 97% of regulatory controllers had a PID structure (Desborough and Miller, 2002).

However, the field of application of regulating PID controllers is rather limited, since they are only linear, symmetric around the setpoint, do not incorporate planning, and cannot utilize direct knowledge of the process, for example.

RL on the other hand, is theoretically capable of solving any Markov decision process (MDP) which makes it applicable to a dramatically broader field of application compared to classical PID control. Moreover, since RL is not limited to linear control, can incorporate the full

observation of the system, and is able to solve planning tasks, it is capable of outperforming PID controllers even in classical industrial applications (Bischoff et al., 2013).

In this paper, we compare two methods from control theory, with three non-interpretable RL methods, and three recently introduced interpretable RL methods. The eight methods are tested using the same data generated by the cart-pole balancing (CPB) benchmark. Since for many real-world problems, the cost of interacting with the real environment on a trial-and-error basis is prohibitive, the methods were not allowed to perform exploration on the real benchmark dynamics but had to utilize existing state transition batch data generated by an arbitrary default exploration policy. To demonstrate the applicability on real-world applications, we learned a controller on batch data generated on a physical cart-pole system (Fig. 1).

2. MODEL-BASED BATCH REINFORCEMENT LEARNING

Generally, RL is distinguished from other computational approaches by its emphasis on an agent learning from direct interaction with its environment. This approach, which does not rely on exemplary supervision or complete models of the environment, is referred to as *online learning*. However, for many real-world problems online learning is prohibited for safety reasons. For example, it is not advisable to deploy an online RL agent, who starts by applying an arbitrary initial policy, which is subsequently improved by exploitation and exploration, on safety-critical systems like power plants or vehicles. For this reason, *offline learning* is a more suitable approach for applying RL methods on already collected training data and, if proving useful, existing models, to yield RL policies.

Offline RL is often referred to as *batch learning* because it is based solely on a previously generated batch of transition samples from the environment. The batch data set contains transition tuples of the form $(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}, r_{t+1})$, where the application of action $\boldsymbol{a}_t$ in state $\boldsymbol{s}_t$ resulted in a transition to state $\boldsymbol{s}_{t+1}$ and yielded a reward r_{t+1}. By applying a batch-mode RL algorithm called *fitted Q iteration* (FQI) (Ernst et al., 2005), an action-value function q_π can be estimated from the available data batch. FQI is an algorithm which uses the fixed point property of the Bellman equation (Bellman, 1962) to perform an *iterative policy evaluation*. Let's denote the initial action-value function of policy π with q_0 and choose its initial approximation arbitrarily. We can now compute each successive approximation by using the Bellman equation as an update rule:

$$q_{k+1}(\boldsymbol{s}_t, \boldsymbol{a}_t) \leftarrow r_{t+1} + \gamma q_k(\boldsymbol{s}_{t+1}, \pi(\boldsymbol{s}_{t+1})), \qquad (1)$$

for all transition tuples in the data batch. Depending on the function approximation applied for q, this algorithm is expected to converge to q_π as $k \to \infty$. The resulting action-value function is used as follows:

$$v_\pi(\boldsymbol{s}) = q_\pi(\boldsymbol{s}, \pi(\boldsymbol{s})), \qquad (2)$$

for all states $\boldsymbol{s}$ of the state space.

Another way of evaluating a policy offline from a batch of transition samples is referred to as *model-based value estimation*. In the first step, supervised ML is applied to
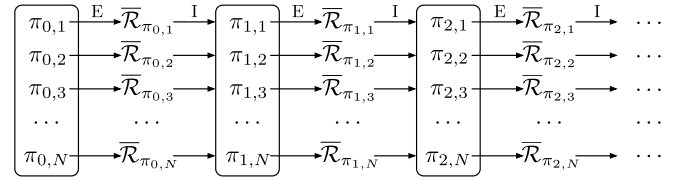

Fig. 2. Population-based policy search iterating policy evaluation (E) and policy improvement (I)

learn approximate models of the underlying environment from transition tuples $(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}, r_{t+1})$ as follows:

$$\tilde{g}(\boldsymbol{s}_t, \boldsymbol{a}_t) \leftarrow \boldsymbol{s}_{t+1}, \quad \tilde{r}(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}) \leftarrow r_{t+1}. \qquad (3)$$

Using models $\tilde{g}$ and $\tilde{r}$, the value for policy π of each state $\boldsymbol{s}$ in the data batch can be estimated by computing $\tilde{v}_\pi(\boldsymbol{s})$. Hence, using model-based value estimation means performing trajectory rollouts on system models for different starting states:

$$\tilde{v}_\pi(\boldsymbol{s}_t) = \sum_{k=0}^{\infty} \gamma^k \tilde{r}(\boldsymbol{s}_{t+k}, \boldsymbol{a}_{t+k}, \boldsymbol{s}_{t+k+1}), \qquad (4)$$

$$\text{with} \quad \boldsymbol{s}_{t+k+1} = \tilde{g}(\boldsymbol{s}_{t+k}, \boldsymbol{a}_{t+k}), \quad \boldsymbol{a}_{t+k} = \pi(\boldsymbol{s}_{t+k}). \qquad (5)$$

2.1 Population-based reinforcement learning

Since we are searching for interpretable solutions, the resulting policies have to be represented in an explicit form. *Policy search* yields explicit policies which can be enforced to be of an interpretable form. Furthermore, policy search is inherently well-suited for being used with population-based optimization techniques, like particle swarm optimization (PSO) and genetic programming (GP).

The goal of using policy search for learning interpretable RL policies is to find the best policy among a set of policies that is spanned by a parameter vector $\boldsymbol{x} \in \mathcal{X}$. Herein, a policy corresponding to a particular parameter value $\boldsymbol{x}$ is denoted by $\pi[\boldsymbol{x}]$. The policy's performance, when starting from $\boldsymbol{s}_t$ is measured by the value functions $v_{\pi[\boldsymbol{x}]}(\boldsymbol{s}_t)$ (Eq. (2)) or $\tilde{v}_{\pi[\boldsymbol{x}]}(\boldsymbol{s}_t)$ (Eq. (4)). Furthermore, including only a finite number of $T > 1$ future rewards for the model-based value estimation from Eq. (4) yields

$$\tilde{v}_{\pi[\boldsymbol{x}]}(\boldsymbol{s}_t) = \sum_{k=0}^{T-1} \gamma^k \tilde{r}(\boldsymbol{s}_{t+k}, \boldsymbol{a}_{t+k}, \boldsymbol{s}_{t+k+1}),$$

$$\text{with} \quad \boldsymbol{s}_{t+k+1} = \tilde{g}(\boldsymbol{s}_{t+k}, \boldsymbol{a}_{t+k}) \quad \text{and} \quad \boldsymbol{a}_{t+k} = \pi[\boldsymbol{x}](\boldsymbol{s}_{t+k}). \qquad (6)$$

To rate the performance of policy $\pi[\boldsymbol{x}]$, the value function is used as follows:

$$\overline{\mathcal{R}}_{\pi[\boldsymbol{x}]} = \frac{1}{|\mathcal{S}|} \sum_{\boldsymbol{s} \in \mathcal{S}} \tilde{v}_{\pi[\boldsymbol{x}]}(\boldsymbol{s}), \qquad (7)$$

with $\overline{\mathcal{R}}_{\pi[\boldsymbol{x}]}$ being the average discounted return of $\pi[\boldsymbol{x}]$ on a representative set of test states $\mathcal{S}$.

In population-based RL, the interest lies in populations of policies, which means that multiple different policies exist at the same time in one policy iteration step (Fig. 2). Applying population-based evolutionary methods to policy search has already been successfully achieved in the past (Chin and Jafari, 1998; Gomez et al., 2006; Chang et al., 2007; Wilson et al., 2018).

3. COMPARED METHODS

3.1 Linear quadratic regulator (LQR)

A well-established control strategy is the linear quadratic regulator (LQR) based on the assumption of a linear dynamical system

$$s_{t+1} = g(s_t, a_t) = U s_t + V a_t, \tag{8}$$

where the matrices U and V are of appropriate dimension and determine the state transition (Bertsekas, 1995). In practical settings, these matrices may either be determined by the governing equations of a physical system or have to be estimated based on observational data. In the latter case, the transition matrices are typically estimated from a sequence of state-action pairs of length $T+1$ by defining the matrices $S = [s_0, s_1, \ldots, s_T]$, $A = [a_0, a_1, \ldots, a_T]$ and $Y = [s_1, s_2, \ldots, s_{T+1}]$ and solving the least-squares problem in matrix form given by

$$\min_{U,V} \| Y - S^T[U, V] \|_F^2. \tag{9}$$

The LQR approach assumes a *quadratic* cost function of the form

$$J(s, a) = \sum_{t=0}^{\infty} s_t^T Q s_t + a_t^T R a_t, \tag{10}$$

where the matrices Q and R are positive semidefinite and account for costs related to the state s_t and action a_t at time t, respectively. The restriction to a quadratic cost function leads to favorable analytical properties and can be justified by choosing the matrices Q, R such that they reflect the desire to stabilize the state s and action a around the origin.

Under some mild assumptions on the matrices U, V, Q, R, the optimal controller is *linear* so that the optimal policy can be written as

$$a_t = \pi(s_t) = -K s_t. \tag{11}$$

Here, the matrix K satisfies the *discrete-time algebraic Riccati equation* (DARE)

$$K = U^T(K - KV(V^T KV + R)^{-1}V^T K)U + Q \tag{12}$$

and can be found by efficient solvers that are built around optimized linear algebra subroutines.

It is important to highlight the assumptions and idealization of the LQR approach compared to many practical control problems. Firstly, the approach is based on an idealized *linear* state transition function, cf. Eq. (8), which is often violated due to non-linear physical effects such as friction and force ripple. Secondly, the applied action a is penalized but assumed to be *unconstrained* so that in principle arbitrary forces may be applied in order to stabilize the system.

3.2 Proportional-integral-derivative (PID) controller

A PID controller is a control loop feedback mechanism which continuously calculates an error value as the difference between a setpoint and a measured process variable. Based on proportional, integral, and derivative terms of this error value, a corrective action is computed and applied to the system. Each of the three PID terms contributes to the action based on its individual gain values k_P, k_I, and k_D for proportional, integral, and derivative terms, respectively. Tuning these terms is the central task in PID controller design. Hence, a huge number of literature exists on how to tune PID controllers for various application domains.

The most common prescriptive rules used in manual PID tuning are the two Ziegler-Nichols response methods (Ziegler and Nichols, 1942). While the first method is applied to plants with step responses, the second method targets unstable plants in a closed loop system, as considered in this work.

In the CPB problem, the two process variables θ and ρ have to be regulated to their respective setpoints $\theta = 0$ and $\rho = 0$. To achieve this goal using PID control theory, a controller layout consisting of two independent PID controllers can be utilized (Wang, 2011). The error values e_θ and e_ρ are determined separately before they are passed on to the respective PID terms. The terms for each component of the controller are computed as follows:

$$P_s = k_{P_s} e_s, \ I_s = k_{I_s} \int_{t-T1}^{t} e_s dt, \ D_s = k_{D_s} \frac{de_s}{dt}, \tag{13}$$

with s as a placeholder for θ and ρ. Hence, the final output of the controller is determined by

$$a(t) = k_\theta(P_\theta + I_\theta + D_\theta) + k_\rho(P_\rho + I_\rho + D_\rho), \tag{14}$$

where k_θ and k_ρ are gains for balancing the contributions from θ and ρ, respectively. Note that tuning this PID controller for CPB using the manual method from Ziegler and Nichols is a feasible yet laborious process. It requires multiple simulation runs using a surrogate model of the CPB and subsequent investigations of the produced trajectories in order to converge to an adequate control performance.

3.3 Neural fitted Q iteration (NFQ)

NFQ is an algorithm for efficient and effective training of an action-value function (Eq. (1)). NFQ uses the ability of neural networks (NNs) to approximate non-linear functions to represent a value function (Riedmiller, 2005). A huge advantage of using a global function representation, like with NNs, is that it can exploit generalization effects by assigning similar values to related areas. However, since changing the weights of NNs has a global effect on the value function output, many difficulties have been reported (Boyan and Moore, 1995). In classical online RL, the value function is updated as soon as a new state-action experience has been made. However, adopting the weights of the NN according to this recent sensation usually not only changes the value of this state-action pair and its related area, but also has an unpredictable influence on values of unrelated state-action examples. With NFQ the malicious influence of a new update is constraint by offering previous knowledge explicitly.

In its original form, NFQ is a model-free RL approach. This means that no explicit model of the environment has to be learned, which makes the method data-efficient for many applications since transition tuples from the plant are directly used to find well-performing control policies. Generally, the method is also applicable for batch RL problems, where policies cannot be evaluated during the training and consequently cannot generate new transition samples. However, using an environment model to evaluate

and select adequate policies during training is very useful in order to yield adequate policy performance.

To adapt Q-learning for NNs, an error function can be derived from Eq. (1) to measure the difference between $q_k(\boldsymbol{s}_t, \boldsymbol{a}_t)$ and $q_{k+1}(\boldsymbol{s}_t, \boldsymbol{a}_t)$:

$$e_{k+1} = \left(q_{k+1}(\boldsymbol{s}_t, \boldsymbol{a}_t) - \left(\boldsymbol{r}_{t+1} + \gamma \max_{\boldsymbol{a}_{t+1}} q_k(\boldsymbol{s}_{t+1}, \boldsymbol{a}_{t+1}) \right) \right)^2. \tag{15}$$

Note that minimizing this mean squared error by common gradient descent techniques for adjusting the weights after each new sample can cause unpredictable effects on other areas in the state-action space. Therefore, Riedmiller proposed to update the neural function offline on the entire set of previous transitions (Riedmiller, 2005).

The result of NFQ is an action-value function which has to be maximized over the set of available actions w.r.t. the current state. Since this action-value function is represented by a multi-layer NN, it is generally very difficult to interpret how a certain action has been computed.

3.4 Particle swarm optimization policy (PSO-P)

PSO-P is a heuristic for solving RL problems by employing numerical online optimization of control action sequences (Hein et al., 2016). As an initial step, a system model is trained from observational data with standard methods. The problem of finding optimal control action sequences based on model predictions is solved with PSO, an established algorithm for non-convex optimization which does not require any gradient information (Kennedy and Eberhart, 1995). Specifically, PSO-P iterates over the following steps. (i) PSO is employed to search for an action sequence that maximizes the expected return when applied to the current system state by simulating its effects using the system model. (ii) The first action of the sequence with the highest expected return is applied to the real-world system. (iii) The system transitions to the subsequent state, and the optimization process is repeated based on the new state (go to step (i)).

The goal of PSO-P is to find an action sequence $\boldsymbol{x} = (\boldsymbol{a}_t, \boldsymbol{a}_{t+1}, \ldots, \boldsymbol{a}_{t+T-1})$ that maximizes the expected return $\mathcal{R}$ for state $\boldsymbol{s}_t$. Adopting the model-based value estimation from Eq. (6), the expected return $\mathcal{R}_{\boldsymbol{x}}$ of action sequence $\boldsymbol{x}$ starting from state $\boldsymbol{s}_t$ is computed by the following value function:

$$v_{\boldsymbol{x}}(\boldsymbol{s}_t) = \sum_{k=0}^{T-1} \gamma^k r(\boldsymbol{s}_{t+k}, \boldsymbol{a}_{t+k}, \boldsymbol{s}_{t+k+1}), \tag{16}$$

$$\text{with } \boldsymbol{s}_{t+k+1} = g(\boldsymbol{s}_{t+k}, \boldsymbol{a}_{t+k}). \tag{17}$$

Solving the RL problem corresponds to finding the optimal action sequence $\hat{\boldsymbol{x}}$ by maximizing

$$\hat{\boldsymbol{x}} \in \arg\max_{\boldsymbol{x} \in \mathcal{A}^T} v_{\boldsymbol{x}}(\boldsymbol{s}_t). \tag{18}$$

3.5 Particle swarm optimization neural network (PSONN)

In PSONN, the policy is represented by an NN whose weights are optimized by PSO. The inputs of the NN policy are the state features and the output is the respective action. Using PSO to optimize the weights of NNs has been successfully applied in several publications (Zhang et al., 2000; Juang, 2004).

Herein, the networks' topologies, as well as the neurons' transfer functions, are fixed and not part of the swarm optimization. Parameter vector $\boldsymbol{x}$ defines the weights of the network graph edges which interconnect each neuron of one layer with all the neurons of the next layer. Each particle of the swarm represents one solution within the parameter space. Using model-based RL, the swarm searches for parameters yielding a policy with maximum average return (Eq. (7)).

The PSONN policy $\pi[\boldsymbol{x}](\boldsymbol{s}) = \boldsymbol{a}$ receives a state $\boldsymbol{s}$ as input, transforms the state features' values according to the pre-defined activation functions and weights in $\boldsymbol{x}$, and passes the neurons' outputs further on to the next layer in the network. The signals produced by the final layer are interpreted as action $\boldsymbol{a}$, which is subsequently applied to the system.

3.6 Fuzzy particle swarm reinforcement learning (FPSRL)

FPSRL utilizes a model-based RL approach similar to PSONN, but instead of optimizing the weights of an NN policy, PSO is used to search for optimal fuzzy-rule-based policy parameters (Hein et al., 2017).

Based on fuzzy set theory, Mamdani and Assilian introduced a so-called fuzzy controller, specified by a set of linguistic if-then rules whose membership functions can be activated independently and produce a combined output computed by a suitable defuzzification function (Mamdani and Assilian, 1975).

In a D-inputs-single-output system with C rules, a fuzzy rule $R^{(i)}$ can be expressed as follows:

$$R^{(i)} : \text{ IF } \boldsymbol{s} \text{ is } m^{(i)} \text{ THEN } o^{(i)}, \quad \text{with } i \in \{1, \ldots, C\}, \tag{19}$$

where $\boldsymbol{s} \in \mathbb{R}^D$ denotes the input vector (the environment state in our setting), $m^{(i)}$ is the membership of a fuzzy set of the input vector in the premise part, and $o^{(i)}$ is a real number in the consequent part.

Herein, Gaussian membership functions (Wang and Mendel, 1992) have been applied:

$$m^{(i)}(\boldsymbol{s}) = \text{m}[\boldsymbol{c}^{(i)}, \boldsymbol{\sigma}^{(i)}](\boldsymbol{s}) = \prod_{j=1}^{D} \exp\left\{ -\frac{(c_j^{(i)} - s_j)^2}{2\sigma_j^{(i)^2}} \right\}, \tag{20}$$

where $m^{(i)}$ is the i-th parameterized Gaussian $\text{m}[\boldsymbol{c}, \boldsymbol{\sigma}]$ with its center at $\boldsymbol{c}^{(i)}$ and width $\boldsymbol{\sigma}^{(i)}$.

The parameter vector $\boldsymbol{x} \in \mathcal{X}$, where $\mathcal{X}$ is the set of valid Gaussian fuzzy parameterizations, is represented by

$$\begin{aligned} \boldsymbol{x} = (&c_1^{(1)}, c_2^{(1)}, \ldots, c_D^{(1)}, \sigma_1^{(1)}, \sigma_2^{(1)}, \ldots, \sigma_D^{(1)}, o^{(1)}, \\ &c_1^{(2)}, c_2^{(2)}, \ldots, c_D^{(2)}, \sigma_1^{(2)}, \sigma_2^{(2)}, \ldots, \sigma_D^{(2)}, o^{(2)}, \ldots, \\ &c_1^{(C)}, c_2^{(C)}, \ldots, c_D^{(C)}, \sigma_1^{(C)}, \sigma_2^{(C)}, \ldots, \sigma_D^{(C)}, o^{(C)}, \alpha). \end{aligned} \tag{21}$$

The output is determined using the following formula:

$$\pi[\boldsymbol{x}](\boldsymbol{s}) = \tanh\left(\alpha \cdot \frac{\sum_{i=1}^{C} m^{(i)}(\boldsymbol{s}) \cdot o^{(i)}}{\sum_{i=1}^{C} m^{(i)}(\boldsymbol{s})} \right), \tag{22}$$

where the hyperbolic tangent limits the output to between -1 and 1, and parameter α can be used to change the slope of the function. The normalization applied in Eq. (22) in combination with the Gaussian membership function ensures that even a relatively small number of membership functions can produce policy outputs for all possible inputs.

3.7 [Fuzzy] Genetic programming reinforcement learning ([F]GPRL)

Applying the aforementioned FPSRL to systems with many state features is prone to yield non-interpretable fuzzy systems since every rule contains all the state dimensions, including redundant or irrelevant ones, in its membership function by default. By creating fuzzy rules using GP rather than tuning the fuzzy rule parameters via PSO, FGPRL eliminates the manual feature selection process (Hein et al., 2018a). GP is able to automatically select the most informative features as well as the most compact fuzzy rule representation for a certain level of performance. Moreover, it returns not just one solution to the problem but a whole Pareto front containing the best-performing solutions for many different levels of complexity.

GP has been utilized for creating system controllers since its introduction (Koza, 1992). Since then, the field of GP has grown significantly and has produced numerous results that can compete with human-produced solutions (Koza, 2010), including system controllers (Keane et al., 2002; Shimooka and Fujimoto, 1999), game playing (Gearhart, 2003; Wilson et al., 2018), and robotics (Downing, 2001; Kamio and Iba, 2005).

GP encodes computer programs as sets of genes and then modifies (evolves) them using a so-called genetic algorithm (GA) to drive the optimization of the population by applying selection and reproduction to the population. The basis for both concepts is a fitness value which represents the quality of performing the predefined task for each individual. Selection means that only the best portion of the current generation will survive each iteration and continue existing in the next generation. Analogous to biological sexual breeding, two individuals are selected for reproduction based on their fitness, and two offspring individuals are created by crossing their chromosomes. Technically, this is realized by selecting compatible cutting points in the function trees and subsequently interchanging the subtrees beneath these cuts. The two resulting individuals are introduced to the population of the next generation (Fig. 3). Herein, we applied tournament selection (Blickle and Thiele, 1995) for selecting the individuals to be crossed.

Since with FGPRL we are interested in using interpretable fuzzy controllers as RL policies, the genes include membership and defuzzification functions, as well as constant floating-point numbers and state variables. These fuzzy policies can be represented as function trees and stored efficiently in arrays (Hein et al., 2018a).

The interpretable policies which are generated by applying the GPRL approach are basic algebraic equations (Hein et al., 2018b). Given that GPRL can find rather short (non-complex) equations, it is expected to reveal sub-
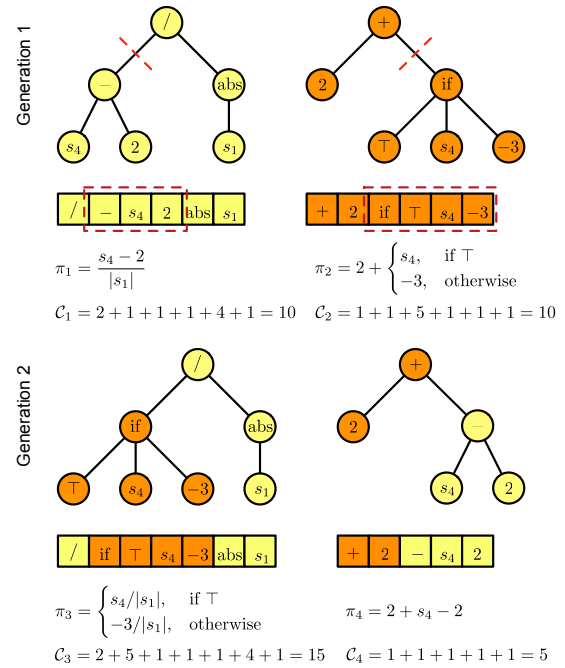

Fig. 3. GP individuals as function trees and memory arrays. Depicted are four exemplary GP individuals π_i from two consecutive generations with their respective complexity measures $\mathcal{C}_i$. Crossover cutting points are marked in the tree diagrams of policies π_1 and π_2.

stantial knowledge about underlying coherencies between available state variables and well-performing control policies for a certain RL problem.

To rate the quality of each policy candidate, a fitness value has to be provided for the GP algorithm to advance. For FGPRL and GPRL, the fitness of each individual is calculated by generating trajectories using the model-based return estimation from Eq. (7).

4. EXPERIMENTS

4.1 Cart-pole balancing (CPB) benchmark

The cart-pole experiments were conducted using the CLS^2 software [1] with default application parameters. The objective of the CPB benchmark is to apply forces to a cart moving on a one-dimensional track to keep a pole hinged to the cart in an upright position. Here, the four Markov state variables are the pole angle θ, the pole angular velocity $\dot{\theta}$, the cart position ρ, and the cart velocity $\dot{\rho}$. These variables describe the Markov state completely, i.e., no additional information about the system's past behavior is required. The task for the CPB controller is to find a sequence of force actions $a_t, a_{t+1}, a_{t+2}, \ldots$ that prevent the pole from falling over.

In the CPB task, the angle of the pole and the cart's position are restricted to intervals of $[-0.7, 0.7]$ and $[-2.4, 2.4]$ respectively. Once the cart has left the restricted area, the episode is considered a failure, i.e., velocities become zero, the cart's position and pole's angle become fixed, and the system remains in the failure state for the rest of the

---

[1] http://ml.informatik.uni-freiburg.de/former/research/clsquare.html

episode. The controller can apply force actions on the cart from -10 N to $+10$ N in time intervals of 0.025 s.

The reward function (negative cost function) for CPB is given as follows:

$$r(\boldsymbol{s}_{t+1}) = \begin{cases} 0.0, & \text{if } |\theta_{t+1}| < 0.25 \text{ and } |\rho_{t+1}| < 0.5, \\ -1.0, & \text{if } |\theta_{t+1}| > 0.7 \text{ or } |\rho_{t+1}| > 2.4, \\ -0.1, & \text{otherwise.} \end{cases}$$
(23)

Based on this reward function, the primary goal of the policy is to avoid reaching the failure state. The secondary goal is to drive the system to the goal state region where $r = 0$ and balance it there for the rest of the episode.

In the experiments in the following section, discount factor $\gamma = 0.97$ and time horizon $T = 100$ have been used. All of the methods utilized the same previously generated transition data batch of size $10\,000$.

4.2 Neural network surrogate model

The required model in Eq. (3) is an NN of the form 5-10-10-10-1 (from input to output) with rectifier activation functions on the three hidden layers. The model computes the deltas of the state variables $\Delta \boldsymbol{s}_{t+1}$ from which the resulting state is calculated according to $\boldsymbol{s}_{t+1} = \boldsymbol{s}_t + \Delta \boldsymbol{s}_{t+1}$. Since CPB yields discrete rewards, one-hot encoding, which transforms the categorical rewards (Eq. (23)) to a more conveniently predictable problem representation, is used. The rewards are mapped to the following binary vectors: $0.0 \rightarrow [1,0,0], -0.1 \rightarrow [0,1,0]$, and $-1.0 \rightarrow [0,0,1]$. Note that such a representation can be interpreted as a probability vector for each observed reward with respect to one of the reward classes. Consequently, the NN does not predict reward r directly but it is rather optimized to predict the probabilities for each reward class by means of the one-hot vector.

4.3 Results and comparison

LQR produced the following linear controller:
$$\pi(\boldsymbol{s}_t) = -\boldsymbol{K}\boldsymbol{s}_t = 38.8\theta + 10.1\dot{\theta} + 2.8\rho + 3.9\dot{\rho}. \quad (24)$$
The penalty of this solution was 2.23 on the model and 3.01 on the real system, which represents a very good control performance for the CPB task.

The best controller found by the Ziegler-Nichols tuning is represented by:

$$\begin{aligned} a(t) = & 0.95 \bigg(0.6 k_{\text{C1}} \cdot e_\theta(t) + 1.2 k_{\text{C1}}/p_{\text{C1}} \cdot \int_{t-40}^{t} e_\theta(t)dt \\ & + 0.6 k_{\text{C1}} p_{\text{C1}}/8 \cdot \frac{de_\theta(t)}{dt} \bigg) \\ & + 0.05 \bigg(0.6 k_{\text{C2}} \cdot e_\rho(t) + 1.2 k_{\text{C2}}/p_{\text{C2}} \cdot \int_{t-28}^{t} e_\rho(t)dt \\ & + 0.6 k_{\text{C2}} p_{\text{C2}}/8 \cdot \frac{de_\rho(t)}{dt} \bigg), \end{aligned}$$
(25)

with critical values $k_{\text{C1}} = -14.3$ and $k_{\text{C2}} = -42.7$, and critical periods $p_{\text{C1}} = 113$ and $p_{\text{C2}} = 331$. The penalty of this solution was 2.03 on the model and 3.04 on the real system. Fig. 4 depicts an example trajectory of this
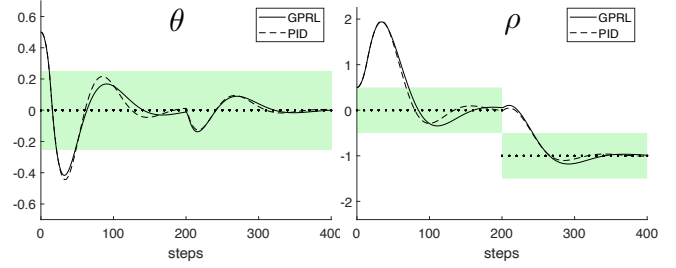

Fig. 4. CPB trajectories for PID and GPRL. Both controllers successfully react to the change in setpoint and the respective goal area after 200 steps.
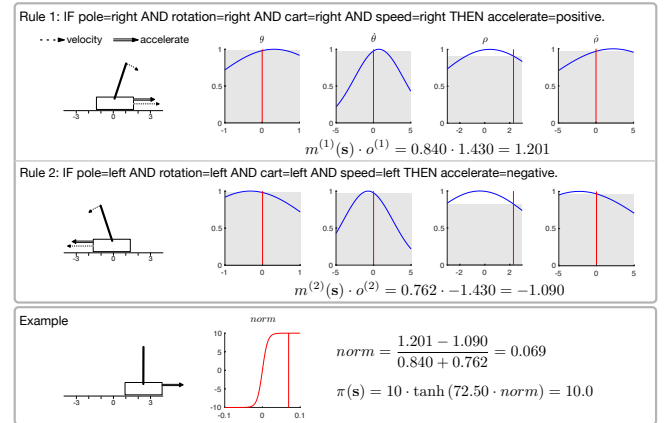

Fig. 5. CPB rules by FPSRL

PID controller. Note how it quickly drives the system into the goal regions and adjusts its action to the changed cart position setpoint after 200 steps.

Fig. 5 depicts a fuzzy rule policy generated by FPSRL. The penalty of this solution was 2.02 on the model and 2.72 on the real system. Only two interpretable fuzzy rules are sufficient to solve the CPB task. However, allowing for higher complexity by tuning the parameters of 4 and 6 fuzzy rules increased the performance up to median penalties of 1.82 on the model and 2.69 on the real system dynamics.

In contrast to FPSRL, FGPRL has the possibility to reduce the complexity by automatically selecting only the most relevant input features. Fig. 6 depicts a fuzzy rule policy generated by FGPRL, where only three out of four input features are used. The penalty of this solution was 2.31 on the model and 3.30 on the real system, which means a small decline in performance compared to the more complex FPSRL solution. Note that FGPRL produces a whole Pareto front (between the functions performance and complexity) of solutions for many levels of complexity incl. the solution with two rules and four input features. The performance of the best policies irrespective of their complexity is shown in Table 1.

A solution of low complexity and high performance generated by GPRL is the following linear equation:
$$\pi(\boldsymbol{s}) = 6.98\theta + 2\dot{\theta} + \rho + 0.94\dot{\rho}. \quad (26)$$
The penalty of this solution was 2.00 on the model and 2.73 on the real system. Note the similarities with the LQR solution from Eq. (24). The ratios between θ and $\dot{\theta}$ are 3.8
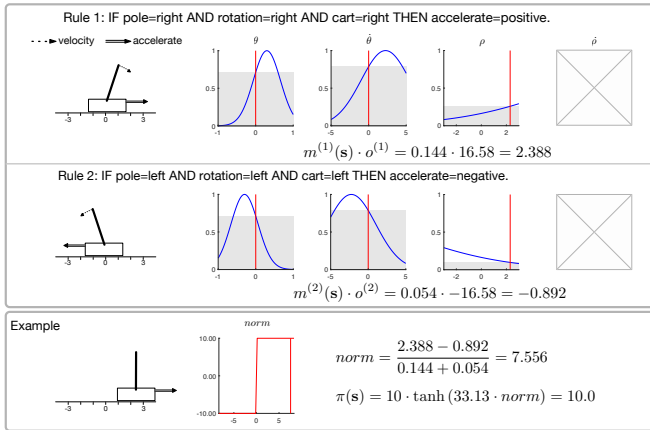
Fig. 6. CPB rules by FGPRL

and 3.5 for LQR and GPRL, respectively, while ρ and $\dot{\rho}$ are weighted by significantly smaller gains. Fig. 4 compares GPRL trajectories for an example start state with that of the PID controller. Note how the fully automatically generated solution from GPRL yields a very similar control performance compared to the manually tuned PID controller. Similarly to FGPRL, GPRL generates a whole Pareto front of solutions. The performance of the best policies is given in Table 1.

Since the policies of NFQ, PSO-P, and PSONN are non-interpretable, only their performance is compared with the other methods in Table 1. PSO-P achieves outstanding performance values as long as its actions are applied to the same model on which the optimization is performed on. Applying the actions to the real system dynamics produces high penalty values. NFQ in a pure model-free setting (*last*) was not able to produce well-performing policies. However, using a surrogate model for selecting the policy (*selected*) yields adequate results. The population-based RL method PSONN produced very good non-interpretable NN controllers for the CPB benchmark. Parameters for all the compared methods evaluated on CPB are published in Hein (2019).

4.4 Evaluation on a hardware cart-pole system

To further evaluate the performance of population-based RL and to demonstrate the applicability to real dynamic systems, we applied GPRL on a hardware cart-pole system. This cart-pole system (Fig. 1) is built from inexpensive components, like an ultrasonic sensor (Elegoo HC-SR04) and a low-cost angle sensor (pwb encoders MEC 22). The inexact information provided by these sensors yields noisy observations from the system. Moreover, the velocities can only be estimated from differences of the positions. Thus, the state of the hardware cart-pole is no longer a true Markovian state, which means we have to estimate the Markovian state from past observations. The estimated state is given by $\tilde{s}_t = (\theta_{t-9}, \dot{\theta}_{t-9}, \rho_{t-9}, \dot{\rho}_{t-9}, a_{t-9}, \ldots, \theta_t, \dot{\theta}_t, \rho_t, \dot{\rho}_t, a_t)$, which yields 50 available input features for a policy.

Approximately 17 000 transition samples have been generated by a human player driving the cart by using a game pad. Two NNs $\tilde{g}_\theta(\tilde{s}_t) = \dot{\theta}_{t+1}$ and $\tilde{g}_\rho(\tilde{s}_t) = \dot{\rho}_{t+1}$ are learned by supervised ML. Angle and position are updated by computing $\theta_{t+1} = \theta_t + \dot{\theta}_{t+1}$ and $\rho_{t+1} = \rho_t + \dot{\rho}_{t+1}$, respectively.

A controller found by GPRL is represented by the following human-interpretable equation:

$$\pi(\tilde{s}_t) = 25\theta_t - \frac{\theta_{t-9}}{0.65 + \dot{\theta}_{t-8}} + \rho_{t-7} + \dot{\rho}_{t-1}. \qquad (27)$$

It is clearly visible that the controller has a strong proportional term w.r.t. θ but also utilizes a past value of cart position ρ.

A video showing the data generation process and the learned controller in action is available at `https://www.youtube.com/watch?v=5ot3xUyFPq4`.

5. CONCLUSION

The experiments with both a CPB software simulation and a hardware demonstrator have shown that the population-based RL methods FPSRL, FGPRL, and GPRL are able to generate human-interpretable controllers from available transition samples. In contrast to LQR and PID controllers which are only linear, symmetric around the setpoint, do not incorporate planning, and cannot utilize direct knowledge of the process, RL is theoretically capable of solving any Markov decision process (MDP) which makes it suitable for a broader field of application.

In real-world industry use cases, data batches of transitions generated by default controllers are often available, whereas online RL is prohibited for safety reasons. Therefore, all of the investigated RL methods are offline approaches which learn directly from batch data or learn a surrogate model first using supervised learning and subsequently perform Monte Carlo rollouts to estimate the value function. Since the space of solutions of GP consists of computer programs, it can search a wide variety of controller classes for domain specific use cases. Examples include precisely defined signal plans for traffic control solutions or optimal scheduling policies for manufacturing industries.

REFERENCES

Bellman, R.E. (1962). *Adaptive Control Processes: A Guided Tour*. Princeton University Press.

Bertsekas, D.P. (1995). *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA.

Bischoff, B., Nguyen-Tuong, D., Koller, T., Markert, H., and Knoll, A. (2013). Learning throttle valve control using policy search. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 49–64. Springer.

Blickle, T. and Thiele, L. (1995). A mathematical analysis of tournament selection. In *ICGA*, 9–16.

Boyan, J.A. and Moore, A.W. (1995). Generalization in reinforcement learning: Safely approximating the value function. In *Advances in neural information processing systems*, 369–376.

Chang, H.S., Hu, J., Fu, M.C., and Marcus, S.I. (2007). Population-based evolutionary approaches. In *Simulation-Based Algorithms for Markov Decision Processes*, chapter 3, 61–87. Springer.

Chin, H.H. and Jafari, A.A. (1998). Genetic algorithm methods for solving the best stationary policy of finite

	LQR	PID	PSO-P	NFQ (last)	NFQ (selected)	PSONN	FPSRL	FGPRL	GPRL
Model	2.23	2.0285	1.2535 ± 0.0014	15.0 ± 1.0	2.71 ± 0.04	1.833 ± 0.002	1.819 ± 0.005	1.814 ± 0.003	1.88 ± 0.02
System	3.01	3.0376	4.20 ± 0.04	15.5 ± 0.9	5.4 ± 0.2	2.831 ± 0.003	2.69 ± 0.03	2.798 ± 0.006	2.71 ± 0.03

Table 1. Penalty comparison of all evaluated methods. Depicted are the mean penalties computed from ten independent experiments. The number after $\pm$ represents the standard error. LQR and PID controllers are single manually constructed solutions.

Markov decision processes. In *System Theory, 1998. Proceedings of the Thirtieth Southeastern Symposium on*, 538–543. IEEE.

Desborough, L. and Miller, R. (2002). Increasing customer value of industrial control performance monitoring - Honeywell's experience. In *AIChE symposium series*, 326, 169–189. New York; American Institute of Chemical Engineers; 1998.

Doshi-Velez, F. and Kim, B. (2017). Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*.

Downing, K.L. (2001). Adaptive genetic programs via reinforcement learning. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, GECCO'01, 19–26. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Ernst, D., Geurts, P., and Wehenkel, L. (2005). Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6, 503–556.

EU (2016). Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *OJ*, L 119, 1–88.

Gearhart, C. (2003). Genetic programming as policy search in Markov decision processes. In J.R. Koza (ed.), *Genetic Algorithms and Genetic Programming at Stanford*, 61–67. Stanford Bookstore, Stanford, USA.

Gomez, F., Schmidhuber, J., and Miikkulainen, R. (2006). Efficient non-linear control through neuroevolution. In *European Conference on Machine Learning*, 654–662. Springer.

Hein, D., Hentschel, A., Runkler, T.A., and Udluft, S. (2017). Particle swarm optimization for generating interpretable fuzzy reinforcement learning policies. *Engineering Applications of Artificial Intelligence*, 65, 87–98.

Hein, D., Hentschel, A., Runkler, T.A., and Udluft, S. (2016). Reinforcement learning with particle swarm optimization policy (PSO-P) in continuous state and action spaces. *International Journal of Swarm Intelligence Research (IJSIR)*, 7(3), 23–42.

Hein, D. (2019). *Interpretable Reinforcement Learning Policies by Evolutionary Computation*. Dissertation, Technical University Munich, Munich.

Hein, D., Udluft, S., and Runkler, T.A. (2018a). Generating interpretable fuzzy controllers using particle swarm optimization and genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '18, 1268–1275. ACM, New York, NY, USA.

Hein, D., Udluft, S., and Runkler, T.A. (2018b). Interpretable policies for reinforcement learning by genetic programming. *Engineering Applications of Artificial Intelligence*, 76, 158–169.

Juang, C.F. (2004). A hybrid of genetic algorithm and particle swarm optimization for recurrent network design. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(2), 997–1006.

Kamio, S. and Iba, H. (2005). Adaptation technique for integrating genetic programming and reinforcement learning for real robots. *Trans. Evol. Comp*, 9(3), 318–333.

Keane, M.A., Koza, J.R., and Streeter, M.J. (2002). Automatic synthesis using genetic programming of an improved general-purpose controller for industrially representative plants. In *Proceedings of the 2002 NASA/DoD Conference on Evolvable Hardware (EH'02)*, EH '02, 113–123. IEEE, Washington, DC, USA.

Kennedy, J. and Eberhart, R.C. (1995). Particle swarm optimization. *Proceedings of the IEEE International Joint Conference on Neural Networks*, 1942–1948.

Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Koza, J.R. (2010). Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3), 251–284.

Mamdani, E.H. and Assilian, S. (1975). An experiment in linguistic synthesis with a fuzzy logic controller. *Int. Journal of Man-Machine Studies*, 7(1), 1–13.

Riedmiller, M. (2005). Neural fitted Q iteration - First experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005*, volume 3720, 317–328. Springer.

Shimooka, H. and Fujimoto, Y. (1999). Generating equations with genetic programming for control of a movable inverted pendulum. In *Selected Papers from the Second Asia-Pacific Conference on Simulated Evolution and Learning on Simulated Evolution and Learning*, SEAL'98, 179–186. Springer-Verlag, London, UK.

Wang, J.J. (2011). Simulation studies of inverted pendulum based on PID controllers. *Simulation Modelling Practice and Theory*, 19(1), 440–449.

Wang, L.X. and Mendel, J.M. (1992). Fuzzy basis functions, universal approximation, and orthogonal least-squares learning. *IEEE Transactions on Neural Networks*, 3(5), 807–814.

Wilson, D.G., Cussat-Blanc, S., Luga, H., and Miller, J.F. (2018). Evolving simple programs for playing Atari games. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, 229–236. ACM, New York, NY, USA.

Zhang, C., Shao, H., and Li, Y. (2000). Particle swarm optimisation for evolving artificial neural network. In *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, volume 4, 2487–2490. IEEE.

Ziegler, J.G. and Nichols, N.B. (1942). Optimum settings for automatic controllers. *Transactions of the A.S.M.E.*, 64(11).