Symbolic method for deriving policy in reinforcement learning

Alibekov, Eduard; Kubalìk, Jiřì; Babuska, Robert

DOI
10.1109/CDC.2016.7798684

Publication date
2016

Document Version
Accepted author manuscript

Published in
Proceedings of the 2016 IEEE 55th Conference on Decision and Control (CDC)

* This is the accepted author manuscript of the following article: Alibekov, E., Kubalík, J.. and Babuška, R., 2016, December. Symbolic method for deriving policy in reinforcement learning. In Proceedings of the 2016 IEEE 55th Conference on Decision and Control (CDC), Las Vegas, NV, 2016, pp. 2789-2795. DOI: 10.1109/CDC.2016.7798684. The published version can be found here: http://ieeexplore.ieee.org/document/7798684/

Symbolic Method for Deriving Policy in Reinforcement Learning*

Eduard Alibekov and Jiří Kubalík and Robert Babuška

***Abstract*— This paper addresses the problem of deriving a policy from the value function in the context of reinforcement learning in continuous state and input spaces. We propose a novel method based on genetic programming to construct a symbolic function, which serves as a proxy to the value function and from which a continuous policy is derived. The symbolic proxy function is constructed such that it maximizes the number of correct choices of the control input for a set of selected states. Maximization methods can then be used to derive a control policy that performs better than the policy derived from the original approximate value function. The method was experimentally evaluated on two control problems with continuous spaces, pendulum swing-up and magnetic manipulation, and compared to a standard policy derivation method using the value function approximation. The results show that the proposed method and its variants outperform the standard method.**

I. Introduction

Reinforcement Learning (RL) algorithms provide a way to optimally solve decision-making and control problems involving dynamic systems [1]. An RL agent interacts with the system by measuring the states and applying actions according to a certain policy. The agent receives a scalar reward signal as an evaluation of its immediate performance. The goal is to find an optimal policy which maximizes the long-term cumulative reward.

In this paper, we consider the critic-only, model-based variant of RL in continuous spaces. Critic-only methods first find the optimal value function (abbreviated as V-function) and then derive an optimal policy from this value function. The typical learning process consists of three steps:

1) Data collection – using a model of the system or the system itself, samples in the form $(x_k, u_k, x_{k+1}, r_{k+1})$ are collected. Here, x_k is the system state, u_k is the control input (action), x_{k+1} is the state that the system reaches from state x_k after applying action u_k, and r_{k+1} is the immediate reward for that transition.
2) Computation of the optimal V-function – based on the samples, an approximation of the V-function is found, which for each system state predicts the cumulative long-term reward obtained under the optimal policy.
3) Policy derivation – based on the computed V-function, the policy is derived at each sampling time instant, so that the system can be controlled in a closed loop.

The policy derivation step can be understood as a hill climbing process. At each step, the agent applies the control input that leads toward a higher point on the V-function surface. An advantage of this control law is its inherent stability – the value function is analogous to the control Lyapunov function [2], [3]. However, the hill climbing process is affected by the approximate nature of the V-function. A typical approximation by means of basis functions exhibits artifacts, which lead to the chattering of the control input and even to limit cycles. This problem is illustrated in Fig. 1.

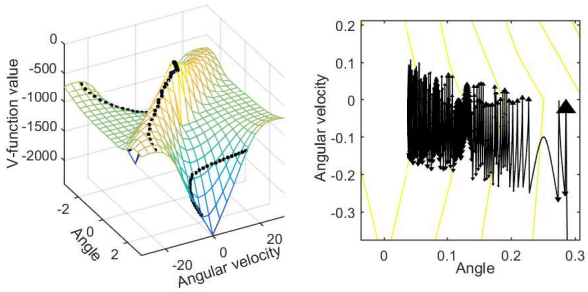

Fig. 1: A sample state trajectory obtained via hill-climbing on the approximate V-function surface for the pendulum swing-up task (see section IV-A). Left: the state trajectory on the V-function surface. Right: the state trajectory in the vicinity of the goal state in [0,0].

This undesired behavior typically occurs in the vicinity of the goal state and leads not only to suboptimal performance, but it can also render the goal state unreachable. An obvious approach to alleviate these problems would be to use a smooth approximation of the V-function. An ideal choice for such a purpose is a symbolic approximator, which can be constructed, e.g., by means of genetic programming. However, we cannot rely on minimizing an error measure between the given V-function data and the resulting symbolic function. Genetic programming has no information about the purpose of the function and can, therefore, ignore small but important parts of the V-function while still achieving the least possible error. Consequently, the resulting smooth approximation can have virtually the same shape as the V-function, but it can yield a completely different, sub-optimal policy.

E. Alibekov and J. Kubalík are with the Czech Institute of Informatics, Robotics, and Cybernetics, Czech Technical University in Prague, Czech Republic, {eduard.alibekov,jiri.kubalik}@cvut.cz

E. Alibekov is also with the Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague

R. Babuška is with the Delft Center for Systems and Control, Delft University of Technology, the Netherlands and also with the Czech Institute of Informatics, Robotics, and Cybernetics, Czech Technical University in Prague, Czech Republic, r.babuska@tudelft.nl

In this paper we present a novel method that uses genetic programming, in particular, a variant of a Single Node Genetic Programming [12], to evolve a smooth proxy to the V-function, which is then used for a continuous policy derivation. A concept similar to *advantage updating* [5] is used. The genetic programming algorithm evolves the symbolic proxy-function that maximizes the number of correct choices of the control action for a set of training states. In this way, the search is biased toward a symbolic proxy-function that would be suited for the policy derivation than a symbolic V-function evolved by minimizing an error measure between the V-function data and the symbolic V-function.

This distinguishes our approach from several works in the literature dealing with a use of genetic programming for V-function fitting. For instance, in [6] a method called Value Function Discovery is proposed that uses GP to evolve algebraic description of the V-function. In [7] an evolutionary algorithm is used to accelerate the convergence of Q-tables. However, both approaches use an error measure between the given V-function data and the resulting symbolic function as an optimization criterion.

The paper is organized as follows. Section II provides a brief introduction to reinforcement learning and genetic programming. The proposed method for finding the proxy-function and four policy derivation methods used in this paper are described in Section III. The benefits of the proposed method are experimentally demonstrated in Section IV and discussed in detail in Section V. Section VI concludes the paper.

II. Preliminaries

A. Reinforcement learning

Define an n-dimensional state space $\mathcal{X} \subset \mathbb{R}^n$, and m-dimensional action space $\mathcal{U} \subset \mathbb{R}^m$. The system to be controlled is described by the state transition function $x_{k+1} = f(x_k, u_k)$, with $x_k, x_{k+1} \in \mathcal{X}$ and $u_k \in \mathcal{U}$. The reward function assigns a scalar reward $r_{k+1} \in \mathbb{R}$ to the state transition from x_k to x_{k+1}:

$$\begin{aligned} x_{k+1} &= f(x_k, u_k) \\ r_{k+1} &= \rho(x_k, u_k) \end{aligned} \tag{1}$$

Define a finite set of discrete control input values $U = \left\{u^1, u^2, \ldots, u^M\right\}$ drawn from $\mathcal{U}$. An approximate V-function denoted by $\hat{V}(x)$ can be computed by solving the Bellman equation:

$$\hat{V}(x) = \max_{u \in U}[r(x,u) + \gamma \hat{V}(f(x,u))] \tag{2}$$

where γ is the discount factor, a user-defined parameter. We employ the fuzzy V-iteration algorithm [4] to find $\hat{V}(x)$. The policy is the mapping:

$$h : \mathcal{X} \to \mathcal{U} \tag{3}$$

and the optimal discrete-valued policy corresponding to $\hat{V}(x)$ is:

$$\hat{h}(x) \in \underset{u \in U}{\operatorname{argmax}} \left[r(x,u) + \gamma \hat{V}(f(x,u))\right], \forall x \tag{4}$$

In the sequel we use $RHS(x,u)$ to refer to the $\left[r(x,u) + \gamma \hat{V}(f(x,u))\right]$ part of (4).

B. Genetic programming

Genetic programming (GP) belongs to methods frequently used to solve the symbolic regression problem. Besides the standard Koza's tree-based GP [11], many other variants have been proposed such as Grammatical Evolution [15] which evolves programs whose syntax is defined by a user-specified grammar, Gene Expression Programming [8] that evolves linear chromosomes that are expressed as tree structures through a genotype-phenotype mapping or graph-based Cartesian GP (CGP) that uses a linear integer representation for expressing programs of the form of a directed graph [14].

A Single Node Genetic Programming (SNGP) [9], [10], used in this work, is a graph-based GP method that evolves a population of individuals, each consisting of a single program node. The node can be either terminal, i.e. a constant or a variable in case of the symbolic regression problem, or a function chosen from a set of functions defined for the problem at hand. Importantly, the individuals are not entirely distinct, they are interlinked in a graph structure similar to that of CGP, so some individuals act as input operands of other individuals.

Formally, a SNGP population is a set of L individuals $M = \{m_0, m_1, \ldots, m_{L-1}\}$, with each individual m_i being a single node represented by the tuple $m_i = \langle e_i, f_i, Succ_i, Pred_i, O_i \rangle$, where

- $e_i \in T \cup F$ is either an element chosen from a function set F or a terminal set T defined for the problem;
- f_i is the fitness of the individual, with the fitness function used in this work described in Section III-B.2;
- $Succ_i$ is a set of successors of this node, i.e. the nodes whose output serves as the input to the node;
- $Pred_i$ is a set of predecessors of this node, i.e. the nodes that use this node as an operand;
- O_i is a vector of outputs produced by this node.

Typically, the population is partitioned so that the first L_{term} nodes, at positions 0 to $L_{term} - 1$, are terminals, followed by function nodes. Importantly, a function node at position i can use as its successor (i.e. the operand) any node that is positioned lower down in the population relative to the node i. This means that for each $s \in Succ_i$ we have $0 \leq s < i$. Similarly, predecessors of individual i must occupy higher positions in the population, i.e. for each $p \in Pred_i$ we have $i < p < L$. The population model used in this work is described in Section III-B.1

In [9], a single evolutionary operator called *successor mutate* (*smut*) has been proposed. It picks one individual of the population at random and then one of its successors is replaced by a reference to another individual of the population making sure that the constraint imposed on successors is satisfied. Predecessor lists of all affected individuals are updated accordingly. Moreover, all individuals affected by this action must be reevaluated as well. In this work, nodes to be mutated are chosen using a *depthwise* selection proposed

in [12], which takes into account both the quality and depth of nodes.

The evolution is carried out via a hill-climbing mechanism using the *smut* operator and an acceptance rule, which can have various forms. Here, the new population is accepted if and only if the best fitness in the population has not been worsened by the mutation operation. Otherwise, the modifications made by the mutation are reversed.

III. Proposed method

A. V-function proxy

Define a set of samples $X = \{x^1, x^2, \ldots, x^N\} \in \mathcal{X}$. The genetic programming algorithm searches for a symbolic function $P(\cdot)$ that satisfies the following condition:

$$\underset{u \in U}{\operatorname{argmax}} \ [P(f(x,u))] = \underset{u \in U}{\operatorname{argmax}} \ [RHS(x,u)], \forall x \in X \tag{5}$$

Observe by comparing (4) and (5) that for the set U of discrete inputs, $P(\cdot)$ yields the same optimal policy as the V-function. Note, that $P(\cdot)$ does not have to satisfy the Bellman equation. Its purpose is to merely provide at each state the same preferred control action. This, however, works under the assumption, that $P(f(x,u))$ has the same distinguishing properties as $RHS(x,u)$. It is only possible when $r(\cdot)$ part of $RHS(x,u)$ does not explicitly depend on u (rather than on $f(x,u)$). Otherwise $P(\cdot)$ will not be capable to robustly work in situations where several different control inputs lead to the same state.

To improve the robustness of the genetic search, we strengthen condition (5) by replacing the $argmax$ operator by the $order$ operator. The $order$ operator produces a partially ordered set of the control input indices so that the corresponding control actions are ordered with respect to their evaluation by the given function. The purpose of this modification is to make sure that if the genetic search finds a suboptimal solution, a high-ranked sub-optimal action is chosen instead of the optimal one.

When $P(\cdot)$ is found, the policy can be derived by using (4), where $RHS(x,u)$ is replaced with $P(f(x,u))$. The overall setting is schematically depicted in Fig. 2.

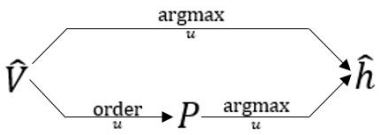

Fig. 2: Standard policy derivation and the proposed method.

The $order$ operator can be formalized as follows:

$$\underset{u \in U}{\operatorname{order}} \ (RHS(x,u)) = \{i, j, ..., \ell\} \tag{6}$$

with

$$RHS(x,u^i) \geq RHS(x,u^j) \geq \cdots \geq RHS(x,u^\ell) \tag{7}$$

B. SNGP for evolving the proxy function

We propose a variant of SNGP using a population model and fitness function as described in the following paragraphs.

1) Population model: Typically, the function set contains functions that might produce invalid output values such as a division by zero. In order to avoid such cases, protected versions of these functions are used instead. These functions are forced to produce a valid output for any input. For example, the protected division outputs a predefined value whenever the denominator is zero. Thus, the output of any candidate expression is ensured to be a valid number. However, due to such hard-coded irregular behavior of the protected functions, expressions making use of the protected functions can still exhibit undesired behavior, e.g., the expressions might become non-differentiable at some data points or can contain local approximation artifacts. Such a symbolic function, even it has a very small error on the training data, can be effectively useless when applied to new previously unseen data.

Here, we propose a *partitioned population* that is divided into two parts – *head* and *tail*. The head part contains nodes that are roots of constant-valued expressions only. It uses extended function set F_e including the protected functions. Each head node can use other head nodes and constant terminal nodes as its input. The tail part contains nodes that can only be chosen from a set F_s of simple non-conflicting functions (i.e. no protected functions). Tail nodes can use all preceding head and tail nodes and both constants and variables as their input. In this way, a "reasonable" behavior of expressions rooted in tail nodes is ensured, since the protected functions are used only for produce constants.

2) Fitness function: When fitting the symbolic proxy-function $P(\cdot)$, a set of M distinct actions $U = \{u^1, \ldots, u^M\}$ sampled from the original continuous action space $\mathcal{U}$ is used to generate a set of N training samples. A training sample generated for each state $x^i \in X$, has the following structure $t^i = [x^i, f(x^i,u^1), \ldots, f(x^i,u^M), o^1, \ldots, o^M]$ where o^j denotes the order class to which the next state obtained by applying the action u^j to the state x^i is assigned according to $RHS(x^i,u^j)$. The best next state is assigned to order class 0, the second best next state is assigned to order class 1, and so on. Note that multiple states can be assigned to the same order class.

A candidate function $P(\cdot)$ produces values $P(f(x^i,u^1))$, $\ldots$, $P(f(x^i,u^M))$ for each x^i in the training set. Denote by $o'^1, \ldots, o'^M$ the order classes derived from these values. The SNGP searches for the proxy-function $P(\cdot)$ that is optimal according to the following fitness function

$$fitness(P(f(x,u))) = \sum_{i=0}^{N} w(x^i) \sum_{j=0}^{M} (fn(j) + fp(j)) \tag{8}$$

where

$$fn(j) = \begin{cases} 1 + 0.1 \ dist(j), & \text{if } o^j = 0 \text{ and } o'^j \neq 0 \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

and

$$fp(j) = \begin{cases} 1, & \text{if } o^j \neq 0 \text{ and } o'^j = 0 \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

The function *false positive*, $fp(j)$, penalizes with penalty 1 the cases where the state $f(x^i, u^j)$ should not belong to the best order class according to $o^j \neq 0$, but it does belong to it (i.e., $o'^j = 0$).

Similarly, the function *false negative*, $fn(j)$, penalizes with penalty 1 the cases where the state $f(x^i, u^j)$ should be the best one according to $o^j = 0$, but it is not (i.e., $o'^j \neq 0$). In order to refine the fitness function, each false negative case is further penalized with the term $dist(j)$ that is calculated as the absolute difference between the $P(f(x^i, u^k))$ of actually the best next state achieved with action u^k applied to the state x^i, and the $P(f(x^i, u^j))$. This way the fitness can distinguish between candidate proxy-function producing the same number of false negative and false positive cases. The value of $dist(j)$ is bounded from above by 1.0. By weighting the $dist(j)$ with the factor 0.1, we make sure that it plays a secondary role in the fitness.

As mentioned in Section I not all states are equally important. The effect of chattering is much stronger in the vicinity of the goal state. Thus, it can be beneficial to weight the overall penalty calculated for a given state with respect to its distance to the goal state. To avoid negative effects, caused by magnitudes of different variables in a state space, each variable of the state space must be mapped into the same range $[0, 1]$. The weight function $w(x^i)$ returns a square of the reciprocal of the Euclidean distance between the state x^i and the goal state. Thus, the errors made in states far from the goal state are penalized less than the errors made in states close to the goal state. When x^i is the goal state, the $w(x^i)$ returns a weight of the state nearest to the goal state.

Since the fitness function expresses how far the candidate proxy-function is from the ideal, the resulting fitness function is to be minimized.

C. Policy derivation using raw function

The first algorithm we consider is a direct usage of the computed proxy-function $P(f(x, u))$, as stated in (4). Algorithm 1 formalizes this procedure.

Algorithm 1: Policy derivation from the raw symbolic function (in the sequel denoted as Raw)

Input: $f(x, u),\ P(f(x, u)),\ \ U,\ x_0$
$k \leftarrow 0$
while *control experiment not terminated* **do**
  $u_k \leftarrow \underset{u' \in U}{\operatorname{argmax}}\ P(f(x_k, u'))$
  $x_{k+1} \leftarrow f(x_k, u_k)$;
  $k \leftarrow k + 1$
end
Output: trajectory $[x_0, x_1, ...]$, $[u_0, u_1, ...]$

D. Policy derivation using hybrid symbolic function

The main purpose of this method is to provide a robust policy. Due to the stochastic nature of genetic programming, successful finding of the optimal proxy-function is not guaranteed. Therefore, all areas, for which $P(f(x, u))$ provides non-optimal policy, should be covered by another approximation. The computed $\hat{V}$ is used for this purpose. Define a vector $C = [c^1, c^2, ..., c^N]^T$ of boolean flags, where N is a number of training samples. Each flag c^i is true if the corresponding training sample t^i is successfully fitted by the proxy-function $P(f(x, u))$, otherwise it is false. At a policy derivation step k, a training sample t^j with its state x^j nearest to the current state x_k is found among all N training samples. This can be easily done by using *k-d tree* for the nearest neighbor search. Then, the optimal input to be applied in state x_k is derived using the $P(f(x, u))$ if the corresponding c^j is true. Otherwise, the policy in state x_k is derived using the $RHS(x, u)$. A pseudo-code for this policy derivation method is shown in Algorithm 2 where the function $NN(x)$ provides the nearest neighbor state to the state x.

Algorithm 2: Policy derivation from the hybrid approximation (in the sequel denoted as Hybrid)

Input: $f(x, u),\ r(x, u),\ C,\ P(f(x, u)),\ \gamma,\ \hat{V},\ U,\ x_0$
$k \leftarrow 0$
while *control experiment not terminated* **do**
  if $C[NN(x_k)]$ **then**
    $u_k \leftarrow \underset{u' \in U}{\operatorname{argmax}}\ P(f(x_k, u'))$
  else
    $u_k \leftarrow \underset{u' \in U}{\operatorname{argmax}}\ [r(x_k, u') + \gamma \hat{V}(f(x_k, u'))]$
  end
  $x_{k+1} \leftarrow f(x_k, u_k)$;
  $k \leftarrow k + 1$
end
Output: trajectory $[x_0, x_1, ...]$, $[u_0, u_1, ...]$

E. Policy derivation using hybrid symbolic function and a fine grid of actions

This method combines the previous method with a fine-sampled set of actions (see [13] for further details).

Define a set of actions A as

$$A = \bar{U}_1 \times \bar{U}_2 \times \cdots \times \bar{U}_m, \quad A \subseteq \mathcal{U} \tag{11}$$

where each set $\bar{U}_i$ contains points equidistantly distributed along the ith dimension of the action space. The set A contains the control inputs that will be considered in the policy derivation using the $P(f(x, u))$ and $RHS(x, u)$, respectively. The policy derivation algorithm is essentially equal to the Algorithm 2 with the action set U being replaced with the set A. In the sequel we call this algorithm HybridGrid.

The size of the set A is given by a vector $A_s = [a_1, a_2, \ldots, a_m]^T$ where each a_i corresponds to the number of points along the ith dimension of the action space. By default we have selected $a_1 = a_2 = \cdots = a_m = 11$.

F. Classical policy derivation

The classical policy derivation uses equation (4) at every policy derivation step k, where the maximization is computed over the same discrete action set U on which $\hat{V}(x)$ has been learned. Formally, this can be described by Algorithm 1 with $RHS(x,u)$ substituted for $P(f(x,u))$. In the sequel we call this algorithm Baseline.

IV. Experimental study

A. Pendulum swing-up

The inverted pendulum consists of a weight of mass m attached to an actuated link that rotates in a vertical plane (see Fig. 3). The available torque is taken insufficient to push the pendulum up in a single rotation from every initial state. Instead, from certain state (e.g., pointing down), the pendulum needs to be swung back and forth to gather energy, prior to being pushed up and stabilized.

The continuous-time model of the pendulum dynamics is:

$$\ddot{\alpha} = \frac{1}{J} \cdot \left[mgl \sin(\alpha) - b\dot{\alpha} - \frac{K^2}{R}\dot{\alpha} + \frac{K}{R}u \right] \qquad (12)$$

where $J = 1.91 \cdot 10^{-4}$ (kgm^2), $m = 0.055$ (kg), $g = 9.81$ (ms^{-2}), $l = 0.042$ (m), $b = 3 \cdot 10^{-6}$ (Nms/rad), $K = 0.0536$ (Nm/A), $R = 9.5$ Ω. The angle α varies in the interval $[-\pi, \pi]$, with $\alpha = 0$ pointing up, and 'wraps around' so that e.g. a rotation of $3\pi/2$ corresponds to $\alpha = -\pi/2$. The state is $x = [\alpha, \dot{\alpha}]$. The sampling period is $T_s = 0.01$ (s), and the discrete-time transitions are obtained by numerically integrating the continuous-time dynamics between consecutive time steps. The control action u is limited to $[-2, 2]$ (V), which is insufficient to push up the pendulum in one go. The set of discrete control inputs is $U = \{-2, -1, -0.05, 0, 0.05, 1, 2\}$.

The control goal is to stabilize the pendulum in the unstable equilibrium $\alpha = \dot{\alpha} = 0$, which is expressed by the following quadratic reward function:

$$r(x,u) = -f^T(x,u) Q f(x,u), \text{ where } Q = \text{diag}[5, 0] \quad (13)$$

B. Magnetic manipulation

Magnetic manipulation (abbreviated as Magman) has several advantages compared to traditional robotic manipulation approaches. First of all, it is contactless, which opens new possibilities for actuation on a micro scale and in environments where it is not possible to use traditional actuators. In addition, magnetic manipulation is not constrained by the robot arm morphology, and it is less constrained by obstacles.

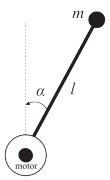

Fig. 3: Inverted pendulum schematic.

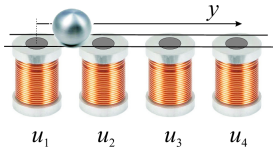

Fig. 4: Magman schematic.

Our magnetic manipulation setup (see Fig. 4) has four electromagnets in a line, but for the experiments presented in this work first two coils, at the positions 0 (m) and 0.025 (m), respectively, have been used. The current through the electromagnet coils is controlled to dynamically shape the magnetic field above the magnets and so to position a steel ball accurately and quickly to a desired set point. The horizontal acceleration of the ball is given by:

$$\ddot{y} = -\frac{b}{m}\dot{y} + \frac{1}{m}\sum_{i=0}^{1} g(y,i)\, u_i \qquad (14)$$

with

$$g(y,i) = \frac{-c_1\ (y - 0.025i)}{\left((y - 0.025i)^2 + c_2\right)^3}. \qquad (15)$$

Here, y denotes the position of the ball, $\dot{y}$ its velocity and $\ddot{y}$ the acceleration. With u_i the current through coil $i = 0, 1$, $g(y,i)$ is the nonlinear magnetic force equation, m (kg) the ball mass, and b ($\frac{Ns}{m}$) the viscous friction of the ball on the rail. The model parameters are listed in Table I.

TABLE I: Magnetic manipulation system parameters

Model parameter	Symbol	Value	Unit
Ball mass	m	$3.200 \cdot 10^{-2}$	kg
Viscous damping	b	$1.613 \cdot 10^{-2}$	Nms
Empirical parameter	c_1	$5.520 \cdot 10^{-10}$	Nm^{5}A^{-1}
Empirical parameter	c_2	$1.750 \cdot 10^{-4}$	m^2
Sampling period	T_s	0.005	s
Set of control inputs	U	$\left\{\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0.6\\0\end{bmatrix}, \begin{bmatrix}0\\0.6\end{bmatrix}\right\}$	V

State x is given by the position and velocity of the ball. The control input u is defined as the vector of currents $[u_1 u_2] \in [0, 0.6]$ to the coils. The reward function is defined as:

$$r(x,u) = |(x_d - f(x,u))|\, Q, \text{ where } Q = \text{diag}[100, 5] \quad (16)$$

The desired position x^d is set to $x^d = 0.01$ (m).

C. V-function learning algorithm

To compute $\hat{V}(x)$, the fuzzy V-iteration algorithm [4] is used. The learning process can be briefly described as follows. Define a set of samples $S = \{s^1, s^2, \ldots, s^N\}$ on an equidistant grid in $\mathcal{X}$. The number of samples per dimension is described by vector $B = [b_1, b_2, \ldots, b_n]^T$ with the total number of samples $N = \prod_{i=1}^{n} b_i$. Further define a vector of fixed triangular basis functions $\phi = [\phi_1(x), \phi_2(x), \ldots, \phi_N(x)]^T$ where each $\phi_i(x)$ is centered in s^i, i.e., $\phi_i(s^i) = 1$ and $\phi_j(s^i) = 0, \forall j \neq i$. The basis functions are normalized so that $\sum_{j=1}^{N} \phi_j(x) = 1, \forall x \in \mathcal{X}$.

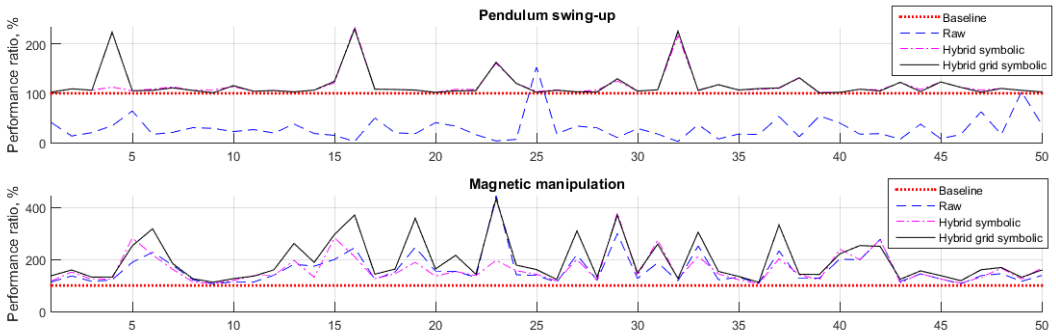

Fig. 5: The performance ratio in percents for 50 simulations of policy derivation. Simulations are performed with randomly chosen initial conditions. The upper and lower plots show the pendulum swing-up benchmark and the magnetic manipulation benchmark, respectively. For both figures the horizontal axis displays separate simulations, which are connected by lines for better visualization.

Finally, define a parameter vector $\theta = [\theta_1, \theta_2, \ldots, \theta_N]^T, \theta \in \mathbb{R}^N$. The V-function approximation is defined as:

$$\hat{V}(x) = \theta^T \phi(x) \tag{17}$$

The fuzzy value iteration is:

$$\theta_i \leftarrow \max_{u \in U} \left[r(s_i, u) + \gamma \ (\theta)^T \ \phi\left(f(s_i, u)\right) \right], i = 1, ..., N \tag{18}$$

The iteration terminates when the following convergence criterion has been satisfied:

$$\epsilon \geq ||\theta - \theta^-||_\infty \tag{19}$$

where θ^- represents the parameter vector in the previous iteration and ϵ is a convergence threshold.

The learning parameters used for both benchmarks are listed in Table II.

D. Complexity of the proxy-function computation

The complexity of proxy-function computation can be estimated only approximately due to the stochastic nature of the genetic programming. The whole evolution process is running for a limited number I of iterations. In each iteration L candidate solutions (see II-B) are evaluated on every given sample point. The total number of samples is denoted as N. Therefore, the complexity linearly depends on the parameters I, L and N.

E. Results

SNGP with a population of size 200 equally divided into the head and tail parts and function sets $F_e = \{+, -, *, /, square, cube, sqrt, exp, sin, ln\}$ and $F_s = \{+, -, *, square, cube, sqrt, sin\}$ were used in this work. Evolution of the proxy-function was run for 5000 iterations, with a maximum time limit of 300 (s).

TABLE II: Fuzzy V-iteration parameters

Parameter	Pendulum swing-up	Magman
Discount factor, γ	0.99999	0.999999
Samples per dimension, B	$[21, 21]^T$	$[21, 27]^T$
Convergence threshold, ϵ	10^{-5}	10^{-8}

To measure the performance of the proxy-function the coverage ratio between correctly fitted samples and the total number of training samples N was used. The coverage map of the pendulum swing-up task is depicted in Fig. 6. Several proxy-functions were fitted for each benchmark. The policy derivation methods were tested 50 times on each proxy-function using the same set of randomly chosen initial states. Using that proxy-functions the policy derivation methods have been tested 50 times with randomly chosen initial conditions. Simulation time was set to 3 (s) for the pendulum swing-up and to 1 (s) for the magnetic manipulator, respectively. To measure the performance of the algorithms the following criteria are defined:

- Average return $R_a = \frac{\sum_{j=1}^{50} \sum_{i=0}^{K} r(x_i, u)}{50}$, where K denotes the number of time steps in a control experiment.
- Performance ratio P_r – the ratio between returns obtained by classical policy derivation and the tested algorithm.
- Average performance ratio in percents (denoted as Average P_r) – a mean of P_r values over 50 simulations.

The results obtained with the best-performing proxy-function with respect to the average P_r for each benchmark are presented in Tables III and IV, respectively. The performance ratios for both benchmarks are depicted in Fig. 5.

TABLE III: Pendulum swing-up simulation results

Criterion	Baseline	Raw	Hybrid	HybridGrid
Average return	-642.09	-2452.10	-601.83	-588.45
Average P_r	100%	31.52%	114.75%	117.64%
Coverage	—	88.8%	88.8%	88.8%

TABLE IV: Magnetic manipulation simulation results

Criterion	Baseline	Raw	Hybrid	HybridGrid
Average return	-106.61	-79.86	-77.98	-71.02
Average P_r	100%	162.47%	163.91%	193.81%
Coverage	—	89.7%	89.7%	89.7%

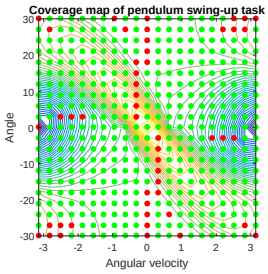

Fig. 6: Coverage map of the pendulum swing-up task. The green dots represent the samples in which the proxy-function is fitted correctly and the red dots otherwise.

V. Discussion

The Hybrid and HybridGrid algorithms significantly outperform the Baseline algorithm as shown in Fig. 5 and Tables III and IV. Fig. 5-top shows poor results for the Raw algorithm on the Pendulum swing-up problem. The reason is quite straightforward. As stated in Section I, states are not equally important and genetic programming cannot guarantee finding the optimal proxy-function, as illustrated in Fig. 6. It shows the coverage map in which green and red dots represent correctly and incorrectly fitted samples, respectively. An incorrectly fitted sample is a sample, in which proxy-function choses a different control input than the V-function. We have empirically found that the regions in the vicinity of the two red dots, nearest to the state space center, must be fitted correctly in order to reach the goal state. Hybrid and HybridGrid algorithms successfully use $RHS(x, u)$ in those regions, while the Raw algorithm failed since it had no ability to do so.

Fig. 7 shows typical simulations of Baseline and HybridGrid algorithms on the magnetic manipulation task. It can be seen that the chattering in the state space is significantly suppressed, but chattering in the control inputs is still present. Input chattering can usually be reduced by penalizing the control input in the reward function, which is, however, not possible here due to the choice of the proxy function structure as $P(f(x, u))$. One way to overcome this limitation is to reformulate the proxy function as $P(x, u)$ and then use it in policy derivation in the same way as a Q-function. This will be a part of our future work.

VI. Conclusion

The proposed method offers an alternative way to the policy derivation. The simulation results show that the proposed approach improves upon the policy derived from the original value function approximation. Moreover, when hybridized with the original value function approximation even better policy has been derived. This approach can be used with any kind of value function approximation.

VII. Acknowledgment

This research was supported by the Grant Agency of the Czech Republic (GAČR) with the grant no. 15-22731S titled "Symbolic Regression for Reinforcement Learning in Continuous Spaces" and by the Grant Agency of the Czech Technical University in Prague, grant no. SGS16/228/OHK3/3T/13 titled "Knowledge extraction from reinforcement learners in continuous spaces".

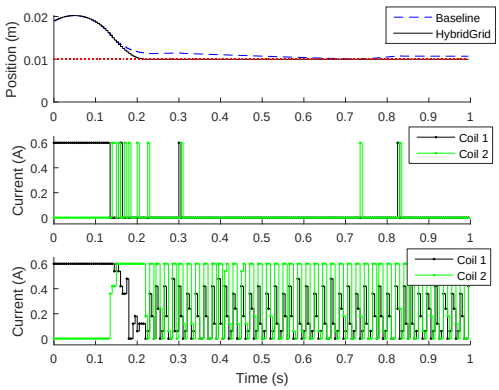

Fig. 7: Simulation of the magnetic manipulation task. The top panel represents position of the ball, where 0.01 m is the desired position. The middle and bottom panels show the control inputs, obtained by applying Baseline and HybridGrid algorithms, respectively.

References

[1] Sutton, R.S. and Barto, A.G. (1998). Reinforcement learning: An introduction. Cambridge Univ Press.

[2] Lewis, F.L. and Vrabie, D. and Syrmos, V.L. (2012). Optimal Control. Wiley, 3rd edition.

[3] Primbs, J.A. and Nevistic, V. and Doyle, J.C. (1999). Nonlinear Optimal Control: A Control Lyapunov Function and Receding Horizon Perspective. *Asian Journal of Control, vol. 1*, pages 14–24.

[4] Busoniu L., Babuska R., Schutter B. D., Ernst D. (2010). Reinforcement Learning and Dynamic Programming Using Function Approximators. CRC Press, Inc.

[5] Baird L. C. (1994). Reinforcement learning in continuous time: advantage updating. In *IEEE World Congress on Computational Intelligence*, vol.4. pages 2448–2453.

[6] Onderwater M., Bhulai S., der van Mei, R. (2015). Value Function Discovery in Markov Decision Processes With Evolutionary Algorithms In *IEEE Transactions on Systems, Man, and Cybernetics: Systems*.

[7] Davarynejad M., van Ast J., Vrancken J., van den Berg J. (2011). Evolutionary value function approximation In *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2011 IEEE Symposium on*, pages 151–155.

[8] Ferreira, C. (2001). Gene expression programming: A new adaptive algorithm for solving problems. *Complex Systems*, 13(2):87–129.

[9] Jackson, D. (2012a). A new, node-focused model for genetic programming. In *EuroGP 2012*, pages 49–60.

[10] Jackson, D. (2012b). Single node genetic programming on problems with side effects. In *PPSN XII*, pages 327–336.

[11] Koza, J. (1992). *Genetic programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, 2nd edition.

[12] Kubalik, J. and Babuska, R. (2015). An Improved Single Node Genetic Programming for Symbolic Regression. In IJCCI (ECTA), pages 244 – 251.

[13] Alibekov, E., Kubalik, J. and Babuska, R. (2016). Policy Derivation Methods for Critic-Only Reinforcement Learning in Continuous Action Spaces. In *IFAC-PapersOnLine, 49(5)*, pages 285–290. Elsevier.

[14] Miller, J. and Thomson, P. (2000). Cartesian genetic programming. In *Poli, R. et al. (eds.) EuroGP 2000, LNCS, vol. 1802, pages 121–132*. Springer.

[15] Ryan, C., Collins, J., Collins, J., and O'Neill, M. (1998). Grammatical evolution: Evolving programs for an arbitrary language. In *LNCS 1391, Proceedings of the First European Workshop on Genetic Programming*, pages 83–95. Springer-Verlag.