Human-Readable Programs as Actors of Reinforcement Learning Agents Using Critic-Moderated Evolution

Senne Deproost$^{1[0009-0009-4757-0290]}$,
Denis Steckelmacher$^{1[0000-0003-1521-8494]}$, and Ann Nowé$^{1[0000-0001-6346-4564]}$

Vrije Universiteit Brussel, Pleinlaan 2, 1060 Elsene, Belgium

Abstract. With Deep Reinforcement Learning (DRL) being increasingly considered for the control of real-world systems, the lack of transparency of the neural network at the core of RL becomes a concern. Programmatic Reinforcement Learning (PRL) is able to to create representations of this black-box in the form of source code, not only increasing the explainability of the controller but also allowing for user adaptations. However, these methods focus on distilling a black-box policy into a program and do so after learning using the Mean Squared Error between produced and wanted behaviour, discarding other elements of the RL algorithm. The distilled policy may therefore perform significantly worse than the black-box learned policy. In this paper, we propose to directly learn a program as the policy of an RL agent. We build on TD3 and use its critics as the basis of the objective function of a genetic algorithm that syntheses the program. Our approach builds the program during training, as opposed to after the fact. This steers the program to actual high rewards, instead of a simple Mean Squared Error. Also, our approach leverages the TD3 critics to achieve high sample-efficiency, as opposed to pure genetic methods that rely on Monte-Carlo evaluations. Our experiments demonstrate the validity, explainability and sample-efficiency of our approach in a simple gridworld environment.

Keywords: reinforcement learning · genetic programming · explainability

1 Introduction

While Deep Reinforcement Learning (DRL) becomes a viable method to generate system controllers in an automatic manner [1, 2], their broader adoption becomes hindered by the lack of transparency and explainability. Since the DRL agent behaviour is computed by a black box model, such as a neural network, the exact method used by the network to map a state to an action is often too complex and beyond human comprehension [3, 4]. For control engineers who require specific guarantees from the controller (stability, robustness, ...) this lack of transparency is unacceptable, leading to low adoption of DRL in their workflow [5].

A decade after the first successes of merging deep and reinforcement learning [6], the emerging field of Explainable Reinforcement Learning (XRL) introduced both local and global explanations within the different stages of training an agent [7]. Whereas *global explanation* methods consider the policy as a whole, and describe its behavior in every state at once (such as "avoid obstacles by the North"), *local explanations* focus on a particular time-step, and offer explanations such as "I went up because there is an obstacle in front". To be able to generate good controllers, a human-readable *global* mapping from input to output is needed [8]. To represent this, different types of explanations have been considered, for instance rule-based [9]. These rules are queried as part of a rule set or selected in a hierarchical way using decision trees [10]. Recently, program-like representations have been considered, closely resembling regular computer programs [11, 12, 13]. They contain the same building blocks as regular source code (variables, control flow, operators, ...) with statements being executed in a sequential order. This improves on both readability of the explanation as well as adaptability by the user. Furthermore, if the program is expressed in a syntax compatible with a Programmable Logic Controller (PLC), their deployment onto actual hardware becomes straightforward [14].

The realization of programs within this Programmatic Reinforcement Learning (PRL) approach is still challenging. Current state-of-the-art methods such as generative networks [12] and search-based templates [11] have shortcomings in their ability to expand beyond a fixed set of possible solutions. To relax this restriction, another approach is to take inspiration from the field of program synthesis using Genetic Programming [13], motivated by the fact that producing a program is an optimization problem without gradients available, and Genetic Programming is both easy to apply and works well for those problems.

Contribution: In this paper, we improve on previous approaches closely related to model distillation [15] by proposing a reward-driven optimization. Instead of minimizing the error between the prediction error of the original model and the distilled one, we opt for the exploitation of the critic network in a TD3 agent [16] to compute the gradient of the program quality (average Q-Values it encounters) with regards to actions predicted by the program. We then steer the genetic algorithm to move the actions produced by the program in the direction of that gradient.

We observe that it is possible to generate programs for simple a simple grid-world environment, with highly promising sample-efficiency, policy quality and explainability.

2 Background

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm used to solve sequential decision problems where each step is taken at a timestep $t \in [0, t_{max}]$ [17] wit t_{max} as time horizon. These problems are modeled as a Markov Decision Process (MDP) [18], a control problem scheme represented by a tuple

(S, A, P_a, R_a) with S the state space, A the action space, P_a the probability distribution of state transitions given action a and R_a the immediate reward given after performing a in the environment. The transition between states is described by $\mathcal{P}^a_{ss'}$. For the formulation of an MDP, we assume the transitions of states to be Markovian, with $Prob\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\}$. This means the transitions of the states are not influenced by past transitions.

At any timestep in the MDP a state s_t is observed. The policy π_t predicts an action a_t for that observation. A new state of the environment $s_{t+1} \in S$ is observed together with reward signal $r_{t+1} \in R$. This can consist of both positive and negative values and is provided by the user as reward function $\mathcal{R}^a_{ss'}$ expressing the control objective. The objective of a Reinforcement Learning agent is to learn a policy that maximizes the discounted sum of rewards $R(\tau) = \sum\limits_t \gamma^t r_{t+1}$, with $\gamma \in [0, 1[$ as the discount factor.

2.2 Q-learning

A well-known approach to Reinforcement Learning, at the basis of our work, is Q-Learning. The quality of an action a_t under a policy π is given by a Q-Value $Q_\pi(s, a)$, the expected return obtained by executing the a_t in s_t, and then following policy π. The Reinforcement Learning agent iteratively refines estimates of the Q-Values. The update rule is given by:

$$\delta = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \qquad \text{(TD error)}$$
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta$$

with learning rate $\alpha \in]0, 1]$. Selecting an action a_t under the optimal policy $\pi^*(s_t)$ is obtained by $a_t = \operatorname{argmax}_a Q^*(s_t, a)$ with Q^* the converged optimal Q-Value function.

2.3 Policy gradient

Q-Learning is a value-based RL method because it learns the value of actions (how good they are). Another approach to RL is Policy Gradient, a policy-based RL method, that directly learns the best parameters of a parametric policy π_θ, such that the agent achieves the highest-possible returns [19]:

$$R_t = \sum_{t'=t}^{T_{max}} r_{t'}$$
$$\nabla_\theta = \sum_t R_t \log \pi(a_t|s_t)$$

with T_{max} the time-step at which one episode finishes. Policy gradient using ∇_θ is applied after the episode, as it needs the Monte-Carlo sum of rewards obtained

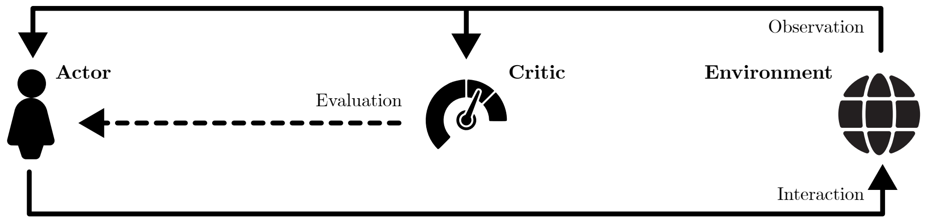

Fig. 1: Schematic overview Actor-Critic architecture. The critic provides Q-values to the made interactions while the actor updates the policy via policy gradient [16].

during the episode. After each application of Policy Gradient, any samples of states, actions and rewards needs to be discarded and new ones need to be collected by executing the updated policy in the environment. This causes Policy Gradient to have low sample-efficiency.

Recent advances in policy-based methods led to Actor-Critic methods that combine an explicit policy π_θ with Q-Values (the critic). An example of Actor-Critic algorithm is TD3 [20], graphically summarized in Figure 1. TD3 trains two separate critic networks with SARSA, and then queries those for a gradient of the actions in a batch of states: $\nabla_a Q(s, \cdot)$. This gradient is then back-propagated into the actor so its parameters are changed in a way that produce actions that are evaluated as more promising by the critics. The use of multiple critics rather than one is one of the improvements of TD3 on Deep Deterministic Policy Gradient (DDPG) [21], reducing value overestimation.

2.4 Use of black-box models in RL

Traditionally, an RL policy is represented by a table of Q-values for discrete state and action spaces. To learn in continuous spaces, a discretization function needs to be applied. The resulting discretization error could be minimized by extending the table size at the cost of memory, which explodes for multi-variable observation spaces. Replacing the table with an artificial neural network addresses this issue since it is capable to generalize between encountered inputs and have been considered universal function approximators[22].

However, due to their outputs being the result of a large number of simple operations (additions, multiplications, simple non-linear functions), using neural networks comes at the cost of losing model transparency, due to the sheer number of operations and seemingly-arbitrary numbers appearing in neural networks. To address this lack of *explainability*, the recent field of Explainable Reinforcement Learning (XRL) has prompted many researchers to come up novel methods of representing policies or critics [7]. Various algorithm generate explanations at different moments (from the beginning, during or after training), and consider either global or local explanations. In this paper, we focus on global explanations, with the aim of producing a program that can be "copy-pasted" in a

Programmable Logic Controller, read, understood and trusted by the control engineer.

2.5 Genetic Programming

Optimizing via evolution is performed by executing iterations of the Genetic Programming (GP) process or generation loop. The starting condition is a population of individuals with g_i being the ith individual's genome. This genome represents model parameters to be optimized (polynomials, weights, ...) or an encoding of the model itself (program statements). *It is up to the user to translate the problem into a representation that fits the GP process*, and one contribution of our paper is the representation of programs as a sequence of real-valued genes. A fitness function is defined, encapsulating the objective to be optimized in order to solve the problem.

At the beginning of each generation, a number of individuals are selected in order to perform *crossover*. These *parents* will exchange genetic material with each other to produce offspring that share their characteristics. Afterwards, *mutation* is applied to the whole population to encourage random exploration in the genome. From this phase, advantageous behaviour can arise that is passed through the generations. It is often used to look beyond the peaks of local optima. In the *selection* phase, the fitness function is applied on each individual. Those that perform worst are eliminated. The best scoring individuals survive and will contribute to the next generational loop.

The optimization ends when the user-specified number of generations is met or some performance threshold. The individual achieving the highest fitness is considered as the best individual. However, in the context of generating programs, individuals with similar performances could also be considered if other criteria are considered (program length, limited nesting, ...).

3 Related work

3.1 Programmatic Reinforcement Learning

In the recent past, several attempts have been made to synthesize programs from an RL agent. Verma et al. introduced a template-based search over a set of programmatic policies [11]. By using an *oracle* network from a trained DRL agent, they steered the search for fitting variable values. However, this method is not so flexible as the templates are user-defined at the start of training, which could be non-optimal. Nevertheless, they showed the approach is well suited for PID-like programs on both a racing game and a classic control environments. Trivedi et al. propose to learn program embeddings using a variational autoencoder (VAE) [12]. This model first learns program embedding by reconstructing source code using several loss functions. Afterwards, it can be used to directly learn a programmatic policy by interacting in the environment and suggesting candidate programs based on return maximization. Hein et al. incorporate genetic programming to generate programmatic trees that represent simple algebraic equations [13]. Mutation of genes happen in the nodes of the trees while crossover switches position between two subtrees within the parents, ensuring the program is kept valid. This method was applied on both the CartPole and MountainCar environment with well performing programs emerging at certain complexity levels.

3.2 Genetic Reinforcement Learning

Genetic programming, besides evolving surrogate models, has been used to optimize other components of the Reinforcement Learning process. Niekum et al. created new reward functions using PushGP, a stack-based programming language [23]. Using a set of simple operators, they could formulate a better reward for dynamic gridworld environments. This approach captures common features among the different environments, allowing for hierarchical decomposition. Finally, Sehgal et al. used GP to optimize the hyperparameters of a DDPG agent on MuJoCo robotic environments [24, 25].

4 Evolving programs

Our contribution builds on TD3 and Genetic Programming to implement the actor of a Reinforcement Learning agent as a program, expressible in source code. Our method differs from related work in two key aspects:

1. The program replaces the TD3 actor, and is optimized using gradients of actions produced by the critics. This allows the actor to directly optimize the returns obtained by the agent, as opposed to approximate some black-box policy, leading to high-quality programs being produced in terms of reward.
2. The program is trained using gradients produced by the TD3 critics, not by direct interaction with the environment. This makes our method several orders of magnitude more sample-efficient than other programmatic RL approaches, that evaluate individuals in the Genetic Algorithm population by performing rollouts in the environment.

We now introduce how we represent programs as a list of real values (the genome), execute these programs, and integrate them as the actor of TD3.

4.1 Representing programs as sequences of real values

At the start of the training process, a population of genomes is initialized as a two-dimensional array of size `num_individuals` $\times$ `num_genes`. Each ith genome $g_i \in G_{min}^{max}$ is a selection of `num_genes` random floats with values in the range of $]-$`len(operators)`$,$`max`$]$ from the gene space G which represents an encoding

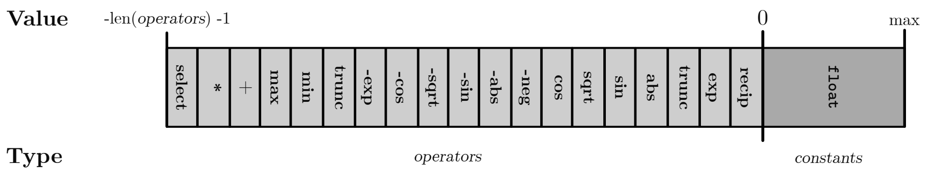

Fig. 2: Values in the gene space and their encoding. Positive values represent literals while negative values the different operators. Literals have a sign randomly sampled at run time. The sign is fixed with *abs* or $-abs$.

of an arbitrary program. As shown in figure 2, we opted for a simple set of operators, encoded as negative values, and literals of randomly-sampled sign, encoded as positive values.

The sequence of operators and literals that make a program are interpreted as the postfix notation of a program, directly linked to the evaluation of the program by a stack machine. For instance, "3 2 sin +" represents $sin(2) + 3$ in the usual infix notation: it pushes 3, pushes 2, replaces the 2 with sin(2), then pops sin(2) and 3 and replaces it with $\sin(2) + 3$.

Some operators have a limited domain, such as the square root operator. We want to ensure that every program is runnable, and thus implement some default value for the operators that have inputs outside of their domain. `reciprocal`, or $\frac{1}{x}$, resolves to the value 20 when x is between -0.05 and 0.05. `exp` clamps its input to at most 10. For the square root, the return value is 0 should the operand value be negative.

4.2 Optimization landscape

The representation of the programs has been designed with some specific aspects that improve learning:

1. Abrupt changes in program behavior following a small mutation of a gene makes the optimization landscape rigid, with many plateaus. To address this issue, we add some stochastic aspects in the program 1.
2. To prevent random programs (when the agent has not learned yet) from biasing the behavior of the agent, we ensure that random programs produce an expected value of 0. We do that by ensuring that literals have no definite sign (the sign is sampled at random at runtime), so they are not biased towards positive values. Functions that have an image biased towards positive or negative numbers also have a negated version. For instance, there exist sqrt and -sqrt.

To smooth the optimization landscape, we introduce stochasticity in the mapping from real value to operators. Instead of casting the real value to an integer, and using that integer to identify an operator, we identify the operator with:

$$o = \lfloor g + x \sim \mathcal{U}(-0.5, 0.5) \rfloor \quad (1)$$

with o the integer index of an operator, g the corresponding real-valued gene, and x a value uniformly sampled from the -0.5 to 0.5 range. When programs are evaluated, they are actually sampled and run 10 times, and the output of the 10 runs is averaged. Slowly changing the value of genes now has the effect of slowly changing that average value, leading to a smooth optimization landscape.

4.3 Execution

To execute a program, a simple stack-based execution is used. Given is the input observed state s, and a genome g_i. The stack is pre-populated with $[s] \times 20$, so input values are available many times on the stack. This encourages operators at the beginning of the genome to use state variables, leading to more state-dependent and *reactive* programs. Genes are then interpreted as described in the previous section: either as literals of random sign, or as operators sampled as described above.

Literals are pushed onto the stack. When an operator is encountered, as many values as required operands are popped from the stack. If a stack underflow happens, the program is considered invalid and given a low fitness. When the proper amount of operands is retrieved, the operator is applied and the result pushed back on the stack.

At the end of execution, the result of the program is the current top of stack value.

4.4 Critic-Moderated evolution

We provide an overview of our method in figure 3 together with pseudocode 1. To perform the evolutionary loop, we use the PyGad library [26]. Used hyper-parameters during the evolutionary loop can be found in appendix 1.

In this paper, we assume both the state space and action spaces to be continuous. When the action space has several dimensions, we learn one program per action dimension.

Every `policy_freq` time-steps, we optimize the programs according to the following procedure. We first use both critics of TD3 to produce *improved actions* A^* for a batch of states S. For this, we query the current programs for current actions $\hat{A}$ for S. We then ask the two critics for Q-Value estimates $Q_A(S)$ and $Q_B(S)$. We compute an overall *program quality* metric by averaging the Q-Values over Q_A and Q_B, and over states. This produces a single real value. All these operations are performed with autograd enabled (using PyTorch in our case [27]) which automatically computes gradients when performing backward passes through the critics. We can then retrieve the gradient of that real value with regards to the actions $\hat{A}$ produced by the programs, leading to ∇_A. Slightly improved actions A^* are then produced by computing $A^* = \hat{A} + \nabla_A$.

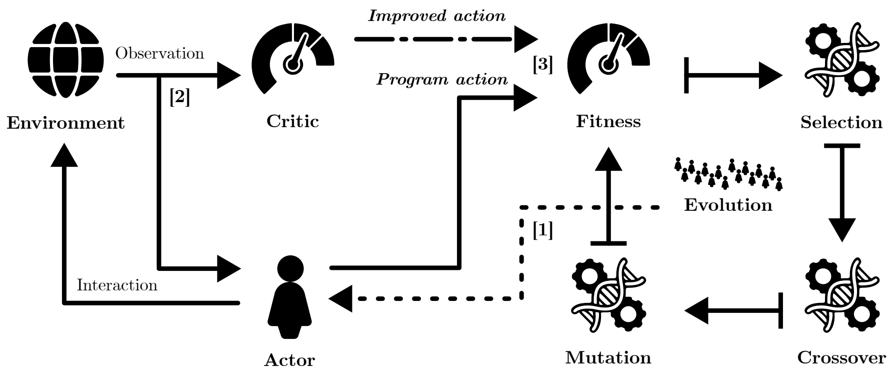

Fig. 3: Our extension of TD3 with the evolution-based optimizer. With critic-improved actions, the evolutionary loop is performed to optimize the population of candidates.

Algorithm 1 Critic-Moderated Genetic Programming (CM-GP)

```
Require: TD3 critics $Q_A$ and $Q_B$, population $g$, rollouts buffer, environment env
 1: for $n = 0$ to n_steps do
 2:     $g^* =$ best_individual($g$)                  ▷ [1] Realize best program
 3:     $p_{g^*} \leftarrow g^*$
 4:     buffer $\leftarrow$ collect_rollout(env) using $p_{g^*}$
 5:     Sample $rollout$ from buffer
 6:     train($Q_A, Q_B$) with $rollout$
 7:     if $step$ mod policy_freq $= 0$ then
 8:         $S \leftarrow \text{states}(rollout)$
 9:         $\hat{A} \leftarrow p_{g^*}(S)$              ▷ Actions of the current best program
10:         repeat 50 times                          ▷ [2] Compute improved actions
11:             $\nabla_A = \text{mean}(Q_A(S), Q_B(S))$
12:             $A^* = \hat{A} + \nabla_A$
13:             $\hat{A} \leftarrow A^*$
14:         end
15:         optimize($g, \hat{A}, S$)                ▷ [3] Generation of evolution
16:     end if
17: end for
```

We now set $\hat{A} \leftarrow A^*$ and repeat the process above 50 times. This forms a sort of 50-step gradient ascent algorithm that, starting from the current output of the programs, follows the TD3 critics to lead to better actions. This optimization process on a critic is akin to the CACLA method proposed in [28]. We stop that process if A^* becomes too different from the actions produced by the programs (when the L1 norm is above 1), akin to the trust region of TRPO proposed in [29].

The improved actions A^* can now be used to optimize the programs with the Genetic Algorithm. The fitness function on a batch of states S for an individual g_i in the population is the mean-squared error (MSE) between program actions $p_{g_i}(S) = \hat{A}$ and improved actions A^* from the critic 2.

$$F_{mse}(g_i, A, S) = \frac{1}{n} \sum_{j=0}^{n} \left(p_{g_i}(S) - A_j^*\right)^2 \tag{2}$$

$$F_{var} = \text{state\_variables}(p_{g_i})/|S| \tag{3}$$

$$\text{fitness} = (1 - F_{mse}) \times F_{var} \tag{4}$$

with F_{var} a fitness term that encourages programs to look at the state variables (as opposed to producing constants). It is computed by looking at how many state variables the program looks at, divided by the number of state dimensions.

The optimized programs can now be used to predict actions for new states, as the TD3 agent continues. We stress that our method allows the programs to influence exploration (as opposed to *a posteriori* distillation), and that the Genetic Algorithm runs on values that do not require numerous rollouts to be performed in the environment (thus allowing for high sample-efficiency). Our contribution is graphically summarized in Figure 3.

5 Results

5.1 Environment

To validate our approach, we used an environment called `SimpleGoal` (fig. 4). This navigation task is performed in a bounded continuous space of size 1×1 where the agent needs to take steps towards a goal area located at $x < 0.1, y < 0.1$. The initial starting position is random. The observation space is the current (x, y) coordinate of the agent. The action space is in the range $[-1, 1]$ and defines the change in x and y for the next time-step, with $dx = 0.1a_0$ and $dy = 0.1a_1$. At each timestep, the reward $r_t = 10 * ($`old_distance`$-$`new_distance`$)$ is calculated based on progress in lowering Euclidean distance towards the goal. If the goal area is reached, an additional reward of 10 is given and the episode terminates. A forbidden area exists at the center of the environment, at coordinates $0.4 < x < 0.6, 0.4 < y < 0.6$. Entering this region terminates the episode with a reward of -10. Otherwise, episodes terminate after 50 time-steps.

5.2 Training

To evaluate our method, we trained on several runs using nodes on HPC infrastructure. We got allocated 40 cores out of a a 2x 32-core AMD EPYC 9384X node with 377 GB of memory. A run of 15.000 steps with our method took roughly 8 hours.

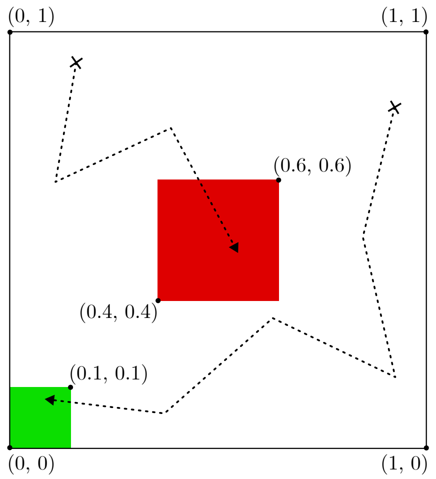

Fig. 4: The `SimpleGoal` environment with the green square the goal region and red the forbidden region. The agent starts at a random position in the environment and tries to reach the goal as quickly as possible.

To compare to the methods that inspired ours, we set up the same experiments for a vanilla TD3 agent and a pure Genetic Programming approach. The TD3 algorithm is the original one from the CleanRL implementation [30]. Our method builds on that TD3 implementation, hence this comparison allows to measure how was TD3 before we introduce our programmatic policy. Learning starts at the same timestep as our method, 2.000. No changes to the original hyperparameters were made.

For the vanilla Genetic Programming (without TD3 critics, using only rollouts), we used the settings as described in table 1. During a learning iteration, the best performing program stands in for the agent in the RL loop. When the policy is updated, the GP process is performed and a new best program is selected. In this case, the fitness function is the performance of one episode in the environment. We already can note that the amount of interactions will increase dramatically with an increase in both `num_generations` and `num_individuals`. We show our results in figure 5.

The vanilla GP approach is sample-inefficient, needing an order of magnitude more interactions with the environment to produce good programs. However, when GP is steered by our critic-moderated approach, learning becomes drastically more efficient. This is because of the lack of environment interactions we need to perform to calculate the fitness function since we do this based on the produced gradients of the critic. TD3 has an even higher sample-efficiency. From these results, we can conclude that:

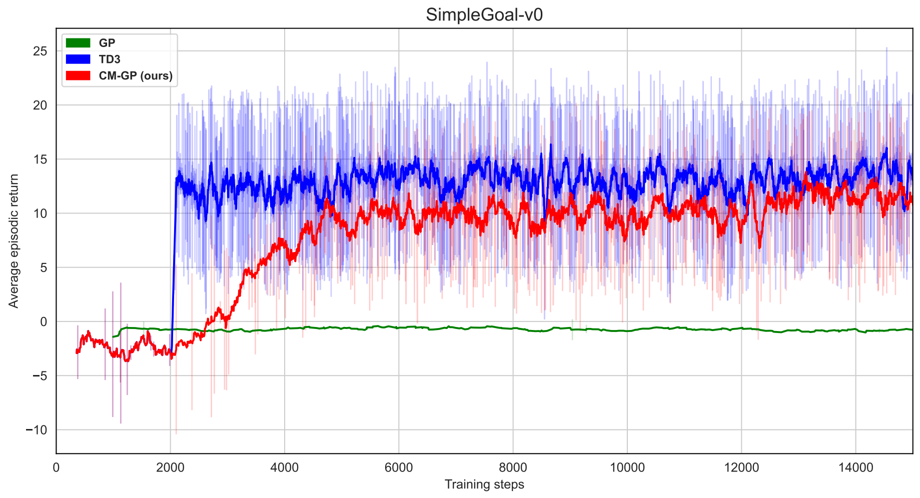

Fig. 5: Comparison between vanilla TD3 (blue), Genetic Programming (green) and TD3 + Genetic Programming (ours, red). The intervals indicate the standard error across 5 runs.

- Our method produces explainable programs (see next section) at no cost of final policy quality, and with much higher sample-efficiency than vanilla Genetic Programming.

5.3 Produced programs

We took a selection of produced programs at the end and plotted their behaviour in an arrow plot (fig. 6). The full programs of each plot are listed in the appendix. At first, we can see that the arrows tend to point towards the goal area in the left bottom corner. The distance of actions taken are quite small, leading to the agent taking a large amount of steps to get to the goal relative to the total distance it has to traverse. For almost all programs (except `prog_1`) the closer the agent is at the goal the larger the step it will take towards it. Since the environment is not strict on going out of range, the agent can take a big step towards the wall and just move along an angle at its edge. Where `prog_2` and `prog_4` tend to avoid the pitfall area, `prog_1` and `prog_3` both have the tendency to enter the area from the right. In the former ones, we also notice a tendency to get stuck in the bottom-right corner.

When we examine the program notations, we can observe some patterns. First, programs tend to be quite complicated in their raw forms, with always-true conditions and a general tendency for producing constants. However, automated or manual constant propagation can be used to make the programs more readable. For all action variables we see the regular incorporation of `sin` and `cos`, probably to have a bending curve effect on the action. In general, we notice

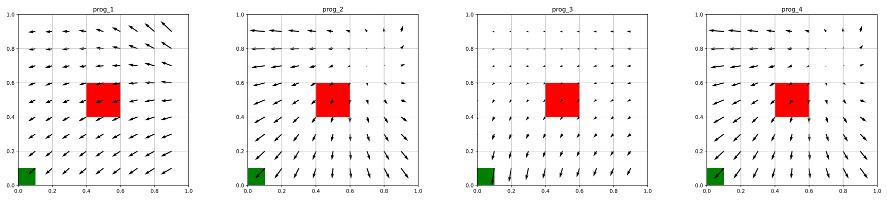

Fig. 6: Arrow plots indicating a sample of policies on the gridworld. The longer the arrow, the bigger the step taken in the direction it points to. Most programs have a tendency to move towards the goal, most of them avoiding the forbidden area.

both action variables produce negative value most of the time, resulting in a direction towards $(0, 0)$.

6 Conclusion

We introduced a new method, building on TD3 and Genetic Programming, for generating programs out of Reinforcement Learning agents based on a critic network. The programs are produced and tuned as part of the RL agent learning (they are its policy). Our experiments show that the learned policies are of comparable quality to black-box vanilla TD3 policies, with a sample-efficiency several orders of magnitude higher than Genetic Programming without TD3.

Our current program representation is quite simple compared to other structures used in GP. Future research avenues include looking into tree-based or graph-based program representations and their dedicated operations. We also designed our algorithm such that the optimization landscape of the GP algorithm is smoothed and without too many local optima. Future program representations may further improve the search landscape, hopefully allowing for more readable yet more expressive policies to be learned in challenging environments.

Finally, we note that the selection of operators is domain-dependent. Selecting a more suiting set of primitives would benefit the interpretability of the produced programs by the end user.

Acknowledgements This research received funding from the Flanders Research Foundation via FWO S007723N (CTRLxAI) and FWO G062819N (Explainable Reinforcement Learning). We acknowledge financial support from the Flemish Government (AI Research Program).

Disclosure of Interests The authors of this dissemination declare to have no conflict of interest with any other party.

Reproduction Code is made available for reproduction at `https://github.com/SenneDeproost/CM-GP.git` under MIT licence.

References

[1] Marc G. Bellemare et al. “Autonomous Navigation of Stratospheric Balloons Using Reinforcement Learning”. In: *Nature* 588.7836 (Dec. 2020), pp. 77–82. ISSN: 1476-4687. DOI: 10.1038/s41586-020-2939-8.

[2] Jonas Degrave et al. “Magnetic Control of Tokamak Plasmas through Deep Reinforcement Learning”. In: *Nature* 602.7897 (Feb. 2022), pp. 414–419. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-021-04301-9.

[3] Cynthia Rudin. “Stop Explaining Black Box Machine Learning Models for High Stakes Decisions and Use Interpretable Models Instead”. In: *Nature Machine Intelligence* 1.5 (May 2019), pp. 206–215. ISSN: 2522-5839. DOI: 10.1038/s42256-019-0048-x.

[4] Tim Miller. “Explanation in Artificial Intelligence: Insights from the Social Sciences”. In: *Artificial Intelligence* 267 (Feb. 2019), pp. 1–38. ISSN: 0004-3702. DOI: 10.1016/j.artint.2018.07.007.

[5] Pavel Osinenko, Dmitrii Dobriborsci, and Wolfgang Aumer. “Reinforcement Learning with Guarantees: A Review”. In: *IFAC-PapersOnLine* 55.15 (2022), pp. 123–128. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2022.07.619.

[6] Volodymyr Mnih et al. “Human-Level Control through Deep Reinforcement Learning”. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14236.

[7] Yanzhe Bekkemoen. “Explainable Reinforcement Learning (XRL): A Systematic Literature Review and Taxonomy”. In: *Machine Learning* (Nov. 2023). ISSN: 0885-6125, 1573-0565. DOI: 10.1007/s10994-023-06479-7.

[8] Markus Langer et al. “What Do We Want from Explainable Artificial Intelligence (XAI)? – A Stakeholder Perspective on XAI and a Conceptual Model Guiding Interdisciplinary XAI Research”. In: *Artificial Intelligence* 296 (July 2021), p. 103473. ISSN: 0004-3702. DOI: 10.1016/j.artint.2021.103473.

[9] Raphael C. Engelhardt et al. “Sample-Based Rule Extraction for Explainable Reinforcement Learning”. In: *Machine Learning, Optimization, and Data Science*. Ed. by Giuseppe Nicosia et al. Cham: Springer Nature Switzerland, 2023, pp. 330–345. ISBN: 978-3-031-25599-1.

[10] Youri Coppens et al. “Distilling Deep Reinforcement Learning Policies in Soft Decision Trees”. In: *International Joint Conference on Artificial Intelligence*. 2019.

[11] Abhinav Verma et al. “Programmatically Interpretable Reinforcement Learning”. In: *Proceedings of the 35th International Conference on Machine Learning*. PMLR, July 2018, pp. 5045–5054.

[12] Dweep Trivedi et al. *Learning to Synthesize Programs as Interpretable and Generalizable Policies*. Jan. 2022. arXiv: 2108.13643 [cs].

[13] Daniel Hein, Steffen Udluft, and Thomas A. Runkler. “Interpretable Policies for Reinforcement Learning by Genetic Programming”. In: *Engineering Applications of Artificial Intelligence* 76 (Nov. 2018), pp. 158–169. ISSN: 09521976. DOI: 10.1016/j.engappai.2018.09.007.

[14] Shengjian Guo, Meng Wu, and Chao Wang. "Symbolic Execution of Programmable Logic Controller Code". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Paderborn Germany: ACM, Aug. 2017, pp. 326–336. ISBN: 978-1-4503-5105-8. DOI: `10.1145/3106237.3106245`.

[15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. Mar. 2015. arXiv: `1503.02531 [cs, stat]`.

[16] Scott Fujimoto, Herke van Hoof, and David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. Oct. 2018. arXiv: `1802.09477 [cs, stat]`.

[17] Richard S. Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. Nachdruck. Adaptive Computation and Machine Learning. Cambridge, Massachusetts: The MIT Press, 2014. ISBN: 978-0-262-19398-6.

[18] Richard Bellman. "A Markovian Decision Process". In: *Indiana University Mathematics Journal* 6 (1957), pp. 679–684.

[19] Richard S Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems* 12 (1999).

[20] Scott Fujimoto, Herke Hoof, and David Meger. "Addressing function approximation error in actor-critic methods". In: *International conference on machine learning*. PMLR. 2018, pp. 1587–1596.

[21] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning". In: *CoRR* abs/1509.02971 (2015).

[22] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer Feedforward Networks Are Universal Approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: `10.1016/0893-6080(89)90020-8`.

[23] Scott Niekum, Andrew G. Barto, and Lee Spector. "Genetic Programming for Reward Function Search". In: *IEEE Transactions on Autonomous Mental Development* 2.2 (June 2010), pp. 83–90. ISSN: 1943-0612. DOI: `10.1109/TAMD.2010.2051436`.

[24] Adarsh Sehgal et al. "Deep Reinforcement Learning Using Genetic Algorithm for Parameter Optimization". In: *2019 Third IEEE International Conference on Robotic Computing (IRC)* (Feb. 2019), pp. 596–601. DOI: `10.1109/IRC.2019.00121`.

[25] Emanuel Todorov, Tom Erez, and Yuval Tassa. "MuJoCo: A Physics Engine for Model-Based Control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033. DOI: `10.1109/IROS.2012.6386109`.

[26] Ahmed Fawzy Gad. "Pygad: An Intuitive Genetic Algorithm Python Library". In: *Multimedia Tools and Applications* (2023), pp. 1–14.

[27] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035.

[28] Hado Van Hasselt and Marco A Wiering. “Reinforcement learning in continuous action spaces”. In: *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*. IEEE. 2007, pp. 272–279.

[29] John Schulman et al. “Trust Region Policy Optimization”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, July 2015, pp. 1889–1897. URL: `https://proceedings.mlr.press/v37/schulman15.html`.

[30] Shengyi Huang et al. “CleanRL: High-quality Single-file Implementations of Deep Reinforcement Learning Algorithms”. In: *Journal of Machine Learning Research* 23.274 (2022), pp. 1–18. URL: `http://jmlr.org/papers/v23/21-1342.html`.

A Experimental setup

Hyperparameter	Value
`num_genes`	5
`num_individuals`	50
`num_generations`	20
`num_parents_mating`	20
`mutation_probability`	0.1
`parent_selection_type`	`sss`
`crossover_type`	`single_point`
`mutation_type`	`random`
`random_mutation_min_value`	-10
`random_mutation_max_value`	10
`policy_freq`	128

Table 1: Used hyperparameters for the evolution strategy

B Operators

Operators	Description	Operands
`abs / -abs`	Absolute value	1
`sin / -sin`	Sine value	1
`cos / -cos`	Cosine value	1
`exp / -exp`	Exponent with max operand value 10	1
`neg`	Negation of the value	1
`+`	Addition	2
`*`	Multiplication	2
`select`	Conditional with test and two cases	3
`max`	Maximum of two operands	2
`min`	Minimum of two operands	2
`id`	Identity function	1
`reciprocal`	Inverse value	1
`trunc`	Truncation of the value	1

Table 2: Available operators

C Produced programs

The produced programs are listed here in their raw form, and then after constant propagation (given a domain of x in $[-1, 1]$).

```
prog_1
   a[0] = -exp(max(-sin((x[0] if trunc(x[1]) > 0 else x[1])), x[0]))
   a[1] = -cos(((((x[1] + x[0]) + x[1]) * x[0]) * x[1]))

prog_1 simplified
   a[0] = -exp(x[0])
   a[1] = -cos(((((x[1] + x[0]) + x[1]) * x[0]) * x[1]))


prog_2
   a[0] = -cos(cos(x[1])) if abs(±66.31885466661134) > 0 else x[0]
   a[1] = -abs(cos(max(cos(-sqrt(x[1])), x[0])))

prog_2 simplified
   a[0] = -cos(cos(x[1]))
   a[1] = -abs(cos(max(cos(-sqrt(x[1])), x[0])))


prog_3
   a[0] = -exp(max((max(-abs(x[1]), x[0]) * x[1]), x[0]))
   a[1] = -abs(reciprocal(-sqrt(((x[1] + x[0]) * x[1]))))

prog_3 simplified
   a[0] = -exp(max((max(-abs(x[1]), x[0]) * x[1]), x[0]))
   a[1] = -abs(reciprocal(-sqrt(((x[1] + x[0]) * x[1]))))


prog_4
   a[0] = -cos(cos(x[1])) if exp(±64.18861262866074) > 0 else x[0]
   a[1] = neg(cos(max(cos(-sqrt(x[1])), x[0])))

prog_4 simplified
   a[0] = -cos(cos(x[1]))
   a[1] = neg(cos(max(cos(-sqrt(x[1])), x[0])))
```