# Genetic Programming For Interpretable Reinforcement Learning

**Genetische Programmierung für interpretierbares Reinforcement Learning**
Master thesis by Bane Janjus
Date of submission: January 23, 2023

1. Review: Dr. Davide Tateo
2. Review: Dr. Riad Akrour
3. Review: Prof. Dr. Jan Peters
Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Computer Science
Department

# Abstract

Reinforcement learning is a subfield of artificial intelligence where an agent learns to solve a task through interaction with an environment. The agent learns with the help of a special signal called the reward that can be sensed by the agent and is produced by the environment. The goal of the agent is to perform a sequence of actions in order to solve the task. An agent can be a robot solving a manufacturing task or an autonomously driving car. These tasks are extremely hard to solve but at the same time interesting from an industry point of view. Deep Reinforcement Learning uses deep neural networks and achieved impressive results on various tasks including autonomous driving. Despite the success of neural networks, their black-box nature complicates their deployment in the real world. Especially in high stake applications such as autonomous driving where human lives are involved. In order to solve this problem the literature proposes several approaches to replace neural networks through interpretable models that can be inspected before deployment. Examples of interpretable models are linear regression models or decision trees. Furthermore, recently genetic programming was proposed as a competitive technique to generate interpretable decision-making models. Motivated by the recent wave of work combining evolutionary with gradient-based algorithms we propose a new algorithm that combines genetic programming with the state-of-the-art Deep-Q-Network algorithm to generate interpretable action-value functions. We compare our algorithm to two genetic programming baselines and show that our algorithm beats the baselines in the most complex environment.

# Zusammenfassung

Reinforcement Learning ist ein Teilgebiet der künstlichen Intelligenz, bei dem ein Agent lernt, eine Aufgabe durch Interaktion mit der Umgebung zu lösen. Der Agent lernt mithilfe eines speziellen Signals auch Belohnung genannt, das vom Agenten wahrgenommen werden kann und von der Umgebung erzeugt wird. Das Ziel des Agenten ist es, eine Folge von Aktionen auszuführen, um die Aufgabe zu lösen. Ein Agent kann ein Roboter sein, der eine Fertigungsaufgabe löst, oder ein autonom fahrendes Auto. Diese Aufgaben sind extrem schwer zu lösen, aber gleichzeitig aus Sicht der Industrie interessant. Deep Reinforcement Learning verwendet tiefe neuronale Netze und hat bei verschiedenen Aufgaben, darunter auch beim autonomen Fahren, beeindruckende Ergebnisse erzielt. Trotz des Erfolgs neuronaler Netze erschwert ihre Blackbox-Natur ihren Einsatz in der realen Welt. Dies gilt insbesondere für Anwendungen wie das autonome Fahren, bei denen Menschenleben auf dem Spiel stehen. Um dieses Problem zu lösen, werden in der Literatur verschiedene Ansätze vorgeschlagen, um neuronale Netze durch interpretierbare Modelle zu ersetzten, die vor dem Einsatz überprüft werden können. Beispiele für interpretierbare Modelle sind lineare Regressionsmodelle oder Entscheidungsbäume. Außerdem wurde kürzlich genetische Programmierung als eine konkurrenzfähige Technik zur Erzeugung interpretierbarer Entscheidungsmodelle vorgeschlagen. Motiviert durch die jüngste Welle von Arbeiten, die evolutionäre und gradientenbasierte Algorithmen kombinieren, schlagen wir einen neuen Algorithmus vor, der genetische Programmierung mit dem State of the Art Deep-Q-Network-Algorithmus kombiniert, um interpretierbare Aktionswertfunktionen zu erzeugen. Wir vergleichen unseren Algorithmus mit zwei Baselines, die genetischen Programmierung verwenden und zeigen, dass unser Algorithmus die Baselines in der komplexesten Umgebung übertrifft.

## Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Bane Janjus, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbtständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, January 23, 2023

B. Janjus

# Contents

# 1 Introduction

Reinforcement learning is a subfield of artificial intelligence and machine learning where an agent learns to solve a task through interaction with the environment. The agent learns with the help of a special signal called the reward that can be sensed by the agent and is produced by the environment. For example, we can have a stock trading agent that interacts with the stock market (the environment) and that can buy different stocks. Buying a stock would represent an action that the agent can execute. Loosing or winning money can be seen as the reward signal. Even though the task is implicitly defined by the rewards normally we are not interested in optimizing the immediate reward we receive from the environment but the long-term reward also called return. Thus the goal is to figure out how to behave in the environment in order to maximize the return. To solve complex tasks such as autonomous driving function approximators is essential to be able to generalize. In the recent years Deep Reinforcement Learning achieved great results in several domains such as robotics [21], autonomous driving [4], and game playing [30]. These results were achieved with the help of powerful function approximators called neural networks. One success known by the broader audience was the breakthrough in the game of GO where the agent was able to beat the world's best player.

Although deep neural networks are the driving force behind these results at the same time their black-box nature is a bottleneck for the deployment of such models. Specifically in areas where human lives are involved such as healthcare and autonomous driving [16], [2]. In other applications where human lives are not involved a black-box model might not be trusted. One notion of trust in the context of machine learning is defined in [28] as being comfortable relinquishing control to a model. For example, given a neural network based trading agent on the stock market, a good performance in a test setting might not be enough to trust the model that there are no conditions under which it takes too much risk or starts losing money. To tackle this problem, the literature proposes interpretable models instead of neural networks. An example of an interpretable model is a small decision tree.

Furthermore, in the past years, it has been shown that evolutionary algorithms are a competitive alternative to modern gradient-based algorithms [39],[51],[46]. Another line of work combines evolutionary and gradient-based algorithms to benefit from the advantages of both worlds [36]. In [17] and [46], the authors use genetic programming to generate interpretable decision-making models.

Following this trend, this work introduces a new algorithm based on a state of the art gradient-based algorithm and genetic programming to learn interpretable action-value functions. We use genetic programming to search in the space of interpretable functional forms and apply gradient-based updates as in the DQN [30] algorithm to fine-tune the numerical constants.

This work is structured as follows: In the second chapter, the preliminary knowledge on reinforcement learning, genetic programming, and interpretability in reinforcement learning is introduced. Chapter 3 summarizes the related work in interpretable reinforcement learning and hybrid algorithms in reinforcement learning. The proposed algorithm is introduced in Chapter 4. Chapter 5 describes the conducted experiment. In the last chapter, we conclude this thesis and give directions for future work.

# 2 Background

In this chapter, we will review important concepts and theories needed for the rest of this work. First, we describe genetic programming as it is a constituent part of our algorithm. Thereafter, we describe Reinforcement Learning as it is the task we want to solve. Lastly, we describe what we consider interpretable in reinforcement learning.

## 2.1 Genetic Programming

The following section is based on [25] [44]. Genetic programming is a genetic algorithm where the population consists of computer programs. Usually, the individuals are represented as syntax trees. The individual in Fig 2.1is equivalent to the following expression: (x * 2) * (y + 1). Given the values of the variables x and y we can evaluate the expression.

The space of possible individuals is defined by the primitive set. The primitive set consists of a function set and a terminal set. The terminal set consists of variables, constants, and 0-arity functions while the function set can contain arithmetic functions (+, -, ...), boolean functions (and, or, ...), the conditional IF-THEN-ELSE, and others.

The first step of the algorithm consist in generating a random initial population. Two basic methods are Grow and Full. Both methods are based on a maximum depth parameter.

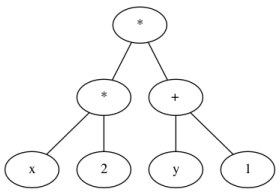


Figure 2.1: Syntax Tree Individual

The Grow method generates trees of various shapes. It constructs a tree by randomly choosing elements from the function and terminal sets until maximum depth is reached afterward it can only use terminals. The disadvantage of this method is that it is sensitive to the choice of both sets i.e. if the terminal set is larger than the function set it tends to produce smaller trees. On the other hand, the Full Method chooses randomly only functions in the function set until the maximum depth is reached afterward it can only use terminals. Therefore every leaf node will be on the maximum depth. An alternative choice is the ranked half-and-half method which creates a diverse set as it uses a range of depth limits and creates half of the individuals with the Grow and the other half with the Full method.

The fitness function determines how well an individual is performing with respect to the task at hand. Given a regression task, one may use the mean squared loss as a fitness of the individual. Besides the fitness function another important component are the genetic operations. There are four genetic operations: selection, crossover, mutation, and reproduction.

**Selection** The selection operator chooses individuals based on their fitness to create new individuals using crossover and mutation. Better fitness of an individual increases the chance of being selected. There are various selection operators, one of them is tournament selection where random individuals are chosen from the population to compete based on their fitness in tournaments and the winners of those tournaments are selected for reproduction. Another example is the fitness proportionate selection.

**Mutation** The mutation operator randomly changes a selected individual. Again there are many different mutation operators one of them is the subtree mutation where one subtree in the current individual is selected and replaced by a newly grown subtree. Another one is the Gaussian mutation where Gaussian noise is added to the numerical constants in the individual.

**Crossover** The crossover operator combines parts from two selected individuals to produce one or two new individuals. The newly created individuals are also called offspring. The subtree crossover selects one subtree of the first parent and one subtree of the second parent and exchanges them (see Fig. 2.3).

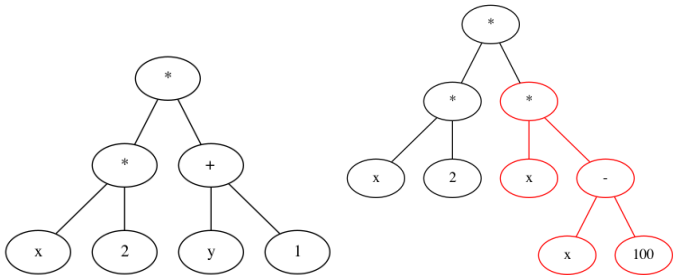


Figure 2.2: Subtree Mutation

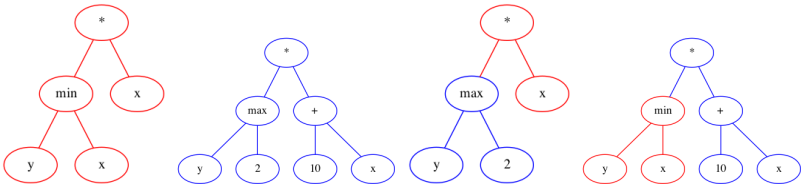


Figure 2.3: Subtree Crossover

**Reproduction** This operator simply copies a selected individual without modification.

There is one necessary property known as closure. The closure property requires type consistency and evaluation safety. Type consistency is required because i.e. a crossover can exchange arbitrary subtrees of selected individuals. One way to enforce type consistency is to introduce types for the primitives in the primitive set and the genetic operators have to respect the type constraints. Evaluation safety is needed to prevent i.e. division by zero. One way to handle the evaluation safety is to define protected functions i.e. before executing the division check if the denominator is zero and if so return a default value. Another desirable property is sufficiency. This property is described as the capability of expressing a solution with the given primitive set. Several hyperparameters must be set before running Alg. 1 such as the type and the probabilities of the genetic operators, the termination criteria (i.e. number of generations), and population size. The algorithm creates a new population from the existing one using the specified genetic operators and repeats this process until the termination criteria are fulfilled. The best individual found is the solution.

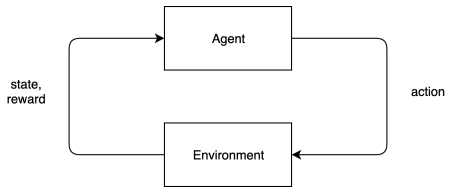


Figure 2.4: Interaction between the agent and the environment

## 2.2 Reinforcement Learning

The following section is based on [40] and [20]. Reinforcement learning is a subfield of machine learning and artificial intelligence in general. Unlike supervised machine learning where the goal is to learn a function for classification or regression on a labeled data set in reinforcement learning an agent has to learn to solve a task in an environment through trial and error learning. The interaction loop is shown in Fig. 2.4. The agent performs an action in the current state $s_t$ and the environment yields the new state $s_{t+1}$ and the reward $r_{t+1}$. There are two different types of tasks: episodic tasks and continuing tasks. In episodic tasks, an episode starts in an starting state and ends in a terminal state. Given that an agent is in a terminal state after a reset it restarts in a starting state or a sample of a starting state distribution. An example of an episodic task is a game of chess when a win or loss would be the terminal state.

The reward is a signal that implicitly describes the task at hand and the agent has to figure out which actions to pick to solve the task. Reinforcement learning is modeled through Markov decision processes (MDP). An MDP consists of an action set A, a state set S, a reward function r(s,a) that maps a state action pair to the expected return, and a state

**Algorithm 1** Abstract GP Algorithm

```
create initial population
while termination criteria not fullfiled do
    determine fitness of each individual
    select individuals (Selection)
    copy some of the selected individuals in the new population (Reproduction)
    create new individuals (Mutation and Crossover)
end while
return best individual found
```

transition function p(s' |s,a) that defines the probability of transitioning from state s to state s' by picking action a. A finite MDP has a finite A and S. The Markov property of an MDP requires that the state representation contains all the relevant information for decision making. This means that knowing some past state wouldn't give any advantage to the agent. Typically in RL the reward function and the state transition function are unknown. The goal of the agent is to maximize the return. The return is defined as $G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$. The $\gamma$ is the discount factor that determines the relevance of future rewards. The policy $\pi$ defines the behavior of the agent. It's a mapping of states to actions. A stochastic policy $\pi(a \mid s)$ maps states to a probability over actions. The value of a state defines how good it is to be in that state based on the expected return. The value function of a given policy is defined in Eq. 2.1. This function returns the expected return by following the policy $\pi$ for a given state.

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s] \tag{2.1}$$

Similarly, the state-action value function q(s,a) is defined as the expected return by following policy $\pi$ after taking action a in a given state s (see Eq. 2.2).

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid s_t = s, a_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a] \tag{2.2}$$

The goal in RL is to find the policy which achieves the best possible return. The optimal policy $\pi^*$ achieves better expected returns in all states than any other policy $\pi'$. This can be formalized with the help of the value function. A policy $\pi$ is better than or equal to a policy $\pi'$ if the following condition is fulfilled: $v_\pi(s) >= v_{\pi'}(s) \forall s \in S$. The optimal policy $\pi^*$ is better than or equal to all other policies. All the optimal policies have the same value function $v^*$ and the same state-action value function $q^*$. Given the optimal Q function $q^*$ we can obtain the optimal policy by choosing the action that has the highest value for a given state: $\pi^*(s) = \max_a q^*(s, a)$. This means we are acting greedy w.r.t. the outputs of q. One specific element that arises in RL is the so-called exploration-exploitation tradeoff. Given that the agent needs to maximize the return it needs to pick actions that yields the highest return. However, to know which action yield the highest return it has to try out different actions sufficiently often.

Reinforcement learning problems with a very small state space are considered as the tabular case. In this case, algorithms can approximate value functions or policies using

arrays or tables. For example, each entry of an array corresponds to the value of a given state. These algorithms are called tabular methods. An example of these tabular methods is the Q-Learning algorithm. This algorithm learns the optimal Q-function Q*. Given a sample from the environment $(s_t, a_t, r_{t+1}, s_{t+1})$ each entry of the Q function is updated by the following rule: $Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$. In this update rule the estimate of the return $r_t + \gamma \max_{a'} Q(s_{t+1}, a')$ is based on the current estimate of the Q-function thus the estimates are bootstrapped. In contrast, a Monte Carlo estimate of the return is the return of a complete episode in the environment. The samples from the environment are collected by a behavior policy e.g. an $\epsilon$-greedy policy. The $\epsilon$-greedy policy samples a random action with probability $\epsilon$ and takes the greedy action w.r.t. the Q-function with probability $1 - \epsilon$. It can be shown that Q converges to $Q^*$.

In the interesting problems, the state space can be very large or continuous e.g., in autonomous driving. Therefore tabular representations of the Q-function are not suitable. Under the assumption that similar states have similar optimal actions, this problem is solved through function approximation. Thus the tabular representation of the Q-function is replaced by a function approximator for example a neural network. In the following, $\theta$ represents the parameters of a function approximator i.e. the weights of a neural network or linear regression.

Q-Learning with function approximation is described in [15] as follows. Like in the tabular version, this algorithm learns the optimal Q-function Q*. Given a target value of Q* $y_t$ the algorithm minimizes the mean squared error (MSE) loss function: $\sum_j (y_t - Q_\theta(s_t, a_t))^2$ Given a sample of the environment $(s_t, a_t, r_{t+1}, s_{t+1})$ the parameters of the function updated by the following rule: $\theta_i = \theta_{i-1} + \alpha(\nabla_{\theta_{i-1}} Q_{\theta_{i-1}}(s_t, a_t))(r_{t+1}) + \max_a Q_{\theta_{i-1}}(s_{t+1}, a) - Q_{\theta_{i-1}}(s_t, a_t)$. Here the target $y_t = r_{t+1} + \max_a Q_{\theta_{i-1}}(s_{t+1}, a)$ uses bootstrapping again.

Motivated by the fact that learning $Q^*$ using a neural network, which is a nonlinear model, is unstable and can even diverge [30] proposes a new variant of the Q-Learning algorithm. This algorithm is based on two ideas: a target network and a replay buffer. Instead of having only one neural network, there is a second one that is used to compute the targets. The network used to compute the targets is called the target network while the network that is learning is called Q-network. The target network parameters are updated periodically every C steps and in between, they are fixed. In the following the target network parameters are denoted as $\theta^-$ and the parameter of the Q-network are denoted as $\theta_i$. Given a sample $(s_t, a_t, r_{t+1}, s_{t+1})$ the targets are computed as follows: $r_{t+1} + \max_a Q_{\theta^-}(s_{t+1}, a)$.

The replay buffer is a data set with a given size that stores samples in the form of $(s_t, a_t, r_{t+1}, s_{t+1})$. The updates of the parameters $\theta_i$ are performed based on samples or

**Algorithm 2** DQN

Initialize replay buffer D with capacity N
Initialize $Q_\theta$ with random parameters $\theta$
Initialize $Q_{\theta^-}$ with parameters $\theta$
**for** episode = 1, M **do**
  Initialize starting state $s_1$
  **for** t = 1, T **do**
    Select random action with probability $\epsilon$
    otherwise select $\max_a Q_{\theta_i}(s_t, a)$
    Execute action $a_t$ and observe reward $r_{t+1}$ and next state $s_{t+1}$
    Store sample $(s_t, a_t, r_{t+1}, s_{t+1})$ in the replay buffer D
    sample mini-batch $(s_t, a_t, r_{t+1}, s_{t+1})$ from D
    **if** $s_t + 1$ is terminal **then**
      $y_t = r_{t+1}$
    **else**
      $y_t = r_{t+1} + \gamma \max_a Q_{\theta^-}(s_{t+1}, a)$
    **end if**
    perform a gradient step on $(y_t - Q_{\theta_i}(s_t, a_t))^2$
  **end for**
**end for**

mini-batches sampled from the replay buffer. The complete algorithm is shown in Alg. 2. A version of this algorithm without target networks was proposed in [31].

## 2.3 Interpretability

In the context of machine learning [3] describes interpretability as a passive quality of a model and as the transparency of a model. Furthermore, the authors distinguish between interpretability and explainability. Explainability is achieved through the use of post-hoc techniques to explain a black-box model. As described in [28] a transparent model is the opposite of a black box or opaque model. Lipton[28] describes three different levels of transparency: simulatability, decomposability, and algorithmic transparency. Simulatability considers the entire model and requires that a person is able to simulate the model in a reasonable amount of time. A simulatable model is a simple model i.e. a tiny neural network could be considered simulatable while a huge decision tree is not.

Decomposability considers the individual parts (inputs, parameters, and calculations). A model is decomposable if each of its parts can be explained i.e. given interpretable features we can explain each node in a decision tree. Algorithmic transparency considers the properties of the learning algorithm i.e. while the error shape and the convergence of linear regression are well understood this is not the case for the neural networks. Following [16] in the context of Reinforcement Learning to achieve interpretability all of the following components need to be interpretable: inputs and their processing, transition, and reward models and the decision-making model (policy and value function). Instead of applying one of the above-mentioned definitions of interpretability (simulatability, decomposability and algorithmic transparency) to all of the reinforcement learning components the interpretability is analyzed for each component separately. For example, an RL approach may be based on a non-interpretable input model and a simulatable policy. This work is situated in the interpretable decision-making as we are learning an interpretable Q function.

# 3 Related Work

In this chapter we will first cover the related work in interpretable reinforcement learning and afterwards the related work that combines evolutionary algorithms with reinforcement learning.

## 3.1 Interpretable Reinforcement Learning

First, we will consider previous work that uses decision trees. Silva et al.[37] uses differentiable decision trees to learn interpretable policies. In addition, a user study is conducted that shows that the resulting policies are more interpretable than a neural network. Topin et al.[43] wraps a MDP in a new MDP representation called IBMDP. Given the proposed training procedure an IBMDP policy is guaranteed to correspond to a decision tree policy for the original an MDP even though any function approximator may be used during training. Additionally, they demonstrate the advantage of their approach when a depth limit for the decision tree is impose which is an important criterion in the context of interpretability. In [34] the authors propose an algorithm that only splits a node, thus increasing the tree size, if the expected return would increase above a given threshold. Hein et al. [18] learns a fuzzy policy with particle swarm optimization that can be represented as a set of IF-THEN rules. In [17] genetic programming is used to obtain policies in the form of simple formulas. Given a function and a terminal set that is used by the algorithm, an expert assigns a complexity weight to each element of this set. The complexity of one individual is the sum of all complexities of its components. The algorithm returns a Pareto Front consisting of the best individuals for each complexity. In both approaches In [17] and [18] a world model is approximated by a neural network. The individuals are evaluated on this world model. Jiang et al.[19] represents policies in first-order logic. The approach is based on policy gradient methods and differentiable inductive models. The policy returned by the algorithm proposed by [1] can be seen as a

sequence of fuzzy IF close(state, center[k]) DO action[k] rules. where action[k] is the associated action with cluster k. To ensure interpretability the number of clusters is kept small and the cluster centers are picked from encountered states. Wilson et al.[51] evolves simple program policies with cartesian genetic programming for Atari games. Verma et al.[45] proposes a framework where policies are represented as a program of a high-level programming language. Given a 'sketch' (prior on the shape of the policy) the goal is to find a policy that fits this sketch and has an optimal long-term reward. The proposed solution is based on imitation learning and local search. The programming language and the sketch induce what is considered to be interpretable. Maes et al.[29] performs a search in the space of simple formulas.

## 3.2 Hybrid Reinforcement Learning

In [22] Evolutionary Reinforcement Learning (ERL) is introduced. ERL is a hybrid algorithm that uses a population of actors for the evolution and uses the samples from evolution evaluations to train an RL Agent in parallel. On the other hand, gradient information flows into the evolution through the periodical insertion of the RL actor. DDPG is used as the RL Agent but it is mentioned that any off-policy actor-critic architecture could be used. Khadka et al.[23] uses multiple learners next to the neuroevolutionary population of actors. Each learner consists of an actor, critic, and algorithm with hyperparameters. All the actors from the population and from the learners are filling a shared replay buffer. This replay buffer is used by the learners to update their parameters via gradient descent. Similar to ERL actor networks from learners are inserted into the population periodically. Furthermore, computational resources are assigned dynamically to the learners based on UCB scores. Given that an actor in ERL is a neural network encoded as a vector of weights [6] shows that standard genetic operators(n-Point Crossover and Gaussian Mutation) are destructive. To overcome this problem in ERL they introduce a transition buffer for each individual in the population and two novel operators: Q-filtered behavior distillation crossover and proximal mutation. The distillation crossover uses the transition buffers of the selected parents to train a child policy via a form of imitation learning. The proximity mutation normalizes the gaussian mutation by the sensitivity of the actions to weight perturbations. To determine the fitness of the actors in the genetic population e.g.in ERL every actor must interact with an environment. To reduce the computational cost, [47] partially replaces the real fitness with a critic-based surrogate. Leite et al.[26] evolves a population of critics. To obtain the fitness of an individual in the population a corresponding actor is fitted to the critic and afterward evaluated in the environment.

NEAT [38] evolves parameters and topologies of networks. NEAT+Q [49] combines NEAT and Q-Learning such that during the evaluation of an individual after each step the parameters are updated toward the target. In [14] a genetic algorithm is used for policy optimization. The population consists of neural network policies with a fixed architecture. A policy gradient algorithm is used as a mutation. A behavior cloning based crossover is used to combine two parent policies in the state space. Pourchot et al.[33] combines the Cross-Entropy Method (CEM)[35] with either DDPG or TD3[13]. CEM samples a population of actors using the current mean vector and covariance matrix. One half of the population is directly evaluated while the other half is first trained using a critic network managed by one of the algorithms mentioned above and afterward evaluated. Samples from all actor evaluations are stored in a common replay buffer. Similarly, [41] combines CEM with ACER[48]. Downing[11] combines genetic programming with Q-learning. An individual is represented by a decision tree where the leaves are action selection points. The rewards of the environment are used for the fitness evaluation of an individual but also to learn what action to take in a given leaf node. Chen et al.[8] proposes a two-stage framework that combines a multi-policy actor-critic with the multi-objective covariance matrix adaptation evolution strategy to approximate a well-distributed Pareto frontier. Based on a state-transition model [24] proposes several methods to find the value function based on genetic programming. One version of these methods uses gradient-based tuning of weights.

# 4 Combining Genetic Programming with Reinforcement Learning

In this chapter, we first describe the software architecture used for our approach. Afterward, we introduce our approach. Then we describe theoretical and practical issues that arise.

## 4.1 Software Architecture

To conduct our experiments we implemented several key components inspired by the MushroomRL[9] library. We have an agent that passes a vector of actions in each step to a vector environment. The actions in the vector correspond to the actions picked by each agent in the population. The vector environment consists of multiple environments, given a vector of actions, each action is executed in the corresponding environment. In order to speed up the computation we use multiprocessing s.t. the different environment are distributed on a number of processes. The vector environment returns the next states and rewards for each environment. These two components are glued together by the Core class. The Core controls the number of episodes the agents can train on.

## 4.2 Algorithm

This work combines several existing ideas into a new algorithm. The first idea comes from the NEAT+Q[49] algorithm that evolves Q-functions. Each individual in NEAT+Q is a neural network. While the fitness of the individual is evaluated standard gradient Q-updates are applied to the Q-networks of the population. Given the black-box nature of neural networks, we use trees from genetic programming as Q-functions. As in [42] we treat the numerical constants in the trees as parameters that can be tuned via gradient

descent. As we are using gradient descent, in addition to the before-mentioned properties in genetic programming closure and sufficiency we also require that the functions in the function set are differentiable. Furthermore, as in [46] and [17], we use the complexity of a tree as a measure of interpretability. A complexity score is assigned to each primitive in the primitive set. For example, the complexity of the sinus function might be two and the complexity of an addition one. In this case, addition is considered more interpretable because its complexity score is lower. The complexity of one tree is the sum of the complexities of its primitives. In the case where all primitives have the same complexity score, trees with fewer nodes are considered more interpretable. Furthermore, we use the NSGA-II[10] multi-objective selection operator that is based on the average return and the complexity of an individual as in [46]. The $\mu + \lambda$ evolution strategy is used. This means that in every generation $\lambda$ new individuals are generated from the current population and $\mu$ individuals are selected for the new population from the current population and the generated offspring. We use the ideas from the DQN algorithm and introduce target trees and replay buffers. The update of the targets can either be done as in DQN or as a soft update as in the DDPG[27] algorithm. In DDPG the targets are updated every step using the following update rule: $\theta^- = \tau\theta_i + (1 - \tau)\theta^-$. Thus the target update method can be seen as a hyperparameter of the algorithm. Furthermore, we tried the standard online updates (no replay buffer), the use of one replay buffer per individual, and the use of a collective buffer that all individuals share. This can also be seen as a hyperparameter of the algorithm. The complete algorithm using one separate buffer per individual is shown in Alg. 3.

First, we generate an initial population and train every individual for several episodes. Afterward, we evaluate each individual for several episodes to determine their fitness. Then we generate an offspring using crossover and mutation. The offspring is trained and evaluated and a new population is created out of the current population and the offspring. This process is repeated for a given number of generations.

In this algorithm every individual is trained for a given number of episodes. Alternatively, one could also choose to continue training the current population in each generation. The third option is to use the reproduction operation. In this case one fraction of the population determined by the reproduction rate would be in the offspring and thus would continue to be trained.

**Algorithm 3**

initialize population of Q-trees of size $\mu$
initialize target Q-trees for the population
initialize buffer of capacity C for each tree in the population
**for** each Q in population in parallel **do**
    train Q for M episodes according to DQN algorithm
**end for**
**for** each Q in population in parallel **do**
    evaluate Q for E epsisodes to determine average return
    determine complexity of Q
**end for**
**for** `g = 1,...,G` **do**
    create offspring of size $\lambda$ using mutation and crossover
    initialize targets for each individual in offspring
    **for** each Q in offspring in parallel **do**
        train Q for M episodes according to DQN algorithm
    **end for**
    **for** each Q in offspring in parallel **do**
        evaluate Q for E epsisodes to determine average return
        determine complexity of Q
    **end for**
    select new population of size $\mu$ out of old population and offspring based on the average return and the complexity
**end for**
select individual manually from the population

## 4.3 Theoretical and Practical Issues

From a theoretical point of view, one problem of our approach is that we use bootstrapping, an off-policy method, and a function approximator (differentiable trees). These three elements combined form the deadly triad described in [40]. In the case of the deadly triad, the danger of instability and divergence is present. Indeed, what we observed during training is that some of the individuals diverge and do not recover independently of parameters and training episodes (see Fig. 4.1). To escape the deadly triad we have to give up one of the before-mentioned elements. For example by replacing the bootstrapped estimates of the targets with Monte Carlo Estimates. However, this implies other drawbacks

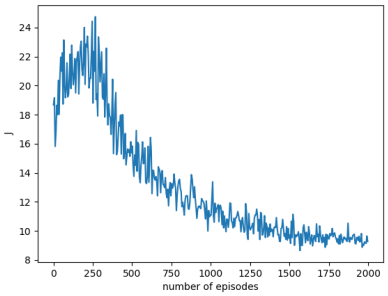


Figure 4.1: Divergence of an individual

regarding data efficiency.

In practice, we observe several problems with our approach. First, as we combined genetic programming with Q-learning we have the hyperparameters of both algorithms which makes some kind of hyperparameter search nearly impossible. Furthermore, when tuning the hyperparameters for Q-Learning one problem that arises is that we need to find parameters that work well for different architectures that are created present during the evolution. The only way to find good hyperparameters in this case is to generate a random population of a given size, find parameters that work well on this population, and assume that they will work for future generations. However, this is a strong assumption. Additionally searching for hyperparameters on an initial population is again computationally expensive and only possible to a given extent.

Furthermore, our first idea was not to use a fitness evaluation step, but rather assess the fitness during the learning episodes. This would make our approach more sample efficient. However, it was not possible due to the instability in the training. For example, if an individual performs well and in the last episode it starts diverging, its average return would be high, and it would be selected into the next generation although its true performance is worse, due to the divergence in the last episode.

# 5 Experiments

In this chapter, we describe the experiments conducted. First, we describe the environments we use for the evaluation. Thereafter, we describe the baselines that we used to evaluate the performance of our algorithm. Finally, we present the results.

## 5.1 Environments

We evaluate our approach on three environments from the OpenAI Gym[7]: Lunar Lander, Cart Pole and Mountain Car. Given that our approach works only for a discrete action space, we use the discrete versions of Lunar Lander and Cart Pole.

**Cart Pole** In this environment, a pole is joined to the cart. Initially, the pole is upright and the goal is to balance the pole by controlling the cart. The agent can push the cart to the left or to the right. The state consists of the cart position, cart velocity, pole angle, and pole angular velocity. In each time step the agent receives a reward of $+1$. The episode terminates if one of the following conditions is fulfilled: the absolute value of the pole angle exceeds 12°, the middle of the cart reaches the end of the display, or after 200 steps.

**Mountain Car** In this environment, a car is placed in a valley between two hills with the goal to reach the top of the right hill as fast as possible. The state consists of the position and the velocity of the car. In each time step the agent receives a reward of -1 and the episode ends if the car reaches the goal or after 200 steps. The agent can choose between accelerating to left, to the right, or not accelerating at all.

**Lunar Lander** This environment consists of a spacecraft and a landing pad. The spacecraft should land on the landing pad. The agent can choose between firing the left engine, the right engine, the main engine, or doing nothing. The state consists of the x and y coordinates, the velocities in the x and y directions, the angle, the angular velocity of the lunar lander and two flags indicating if the left, and right legs are on the ground. An episode ends if the lunar lander leaves the viewport, crashes, comes to rest, or after 1000 steps. When the lander crashes the agent receives -100 points while when it comes to rest it receives 100 points. It loses reward when it moves away from the landing pad. Firing the main or one of the side engines gives -0.3 or -0.03 points respectively. Touching the ground gives 10 points per leg.

## 5.2 Baselines

In this section, we describe the baselines used to compare our algorithm. Videau et al.[46] uses genetic programming to evolve policies for several OpenAI Gym benchmark tasks. Both algorithms are based on the $\mu + \lambda$ evolution strategy and run for several generations. The fitness of each individual is based on the average return achieved while interacting with the environment. Every individual is evaluated on at least one episode. Furthermore, there is a simulation budget T assigned to each generation. This means T times a batch of k individuals will be evaluated in the given generation. The k individuals are chosen based on a multi-arm-bandit-like strategy. Each individual is assigned the following fitness value: $\overline{x} + c\sqrt{\frac{ln(n'+T)}{n}}$. $\overline{x}$ is the average return. $n$ is the number of episode evaluations of the given policy. $c$ is the exploration constant and $n'$ is the number of episode evaluations until the current generation. $\overline{x}$ and $n$ are updated after every evaluation. The k policies with the highest value from the population and the offspring are selected for an additional evaluation in the environment. These two mechanisms are used to evaluate the promising individuals and augment the sample efficiency. Furthermore, the simulation budget is increased over time motivated by the fact that if the differences in performance of the individuals decrease then the variance should decrease as well. We set the exploration constant $c$ to zero thus the above-described UCB-like score becomes simply the average return.

Based on this main loop described above Tree Genetic Programming (Tree GP) and Linear Genetic Programming (Linear GP[5]) are used. In Tree GP policies are represented as syntax trees as described above. The mutation operator is defined as a combination of the gaussian mutation and the subtree mutation. NSGA-II selection operator is used to select

| Primitive | Complexity |
|---|---|
| Variable | 1 |
| Constant | 1 |
| + - × | 1 |
| / | 2 |
| exp, log, pow | 3 |
| <, > | 4 |
| if | 5 |

Table 5.1: Operator complexity

individuals based on two objectives: the average return achieved and the complexity of the individual. The complexity of the individual is obtained as the sum of the complexities of the individual primitives. (Table 5.1). Table 5.1 is not exhaustive.

The second algorithm is based on Linear GP. The individuals are represented in an imperative language. An example of an individual is shown in Fig. 5.1. This individual corresponds to the same individual shown in Fig. 2.1 or to the expression (x * 2) * (y * 1). The array r = x, y, 1.0, 1.0, 1.0, 2.0 represents the registers. r[0] and r[1] are the input registers r[4] and r[5] are used to store constants. r[3] and r[4] are used for calculations. Furthermore in this case r[2] is used as the output.

Listing 5.1: Linear GP Individual

```
r = {x, y, 1.0, 1.0, 1.0, 2.0}
void individual(r){
    r[2] = r[0] * r[5]
    r[3] = r[3] * r[4]
    r[2] = r[2] * r[3]
}
```

The mutation is either the insertion, modification, or removal of an instruction each of these mutations is occurring independently of the others. The probability of removing an instruction is twice the probability of inserting an instruction thus simpler programs are favored. A Tournament selection of size five is used.

### 5.2.1 Implementation

The proposed algorithms and the baselines are implemented in Python 3. Even though the code of the baselines was made available we reimplemented some parts to evaluate the performance of the algorithm based on the number of episodes instead of the number of generations. The implementation of the proposed algorithms is based on DEAP[12] for the genetic operations and PyTorch[32] for the calculation of the gradients.

## 5.3 Results

In this section, we present the results of the before-mentioned environments. We tried several hyperparameter configurations of our algorithm for each environment and report the results of the best one per environment. For the baselines best hyperparameters from the provided repository are used. To evaluate the performance of our algorithm and the baseline we evaluate the best-performing individual in each generation. We ran each experiment on five different seeds to obtain a representative result. In the Mountain Car environment, the best-performing configuration used the common buffer that all individuals share. This can be motivated by the fact that a common buffer enhances the exploration of the state space i.e. if one individual reaches the top of the right hill all the other individuals that haven't explored enough will be able to use this experience. In the following, J represents the mean reward achieved by an individual. In Fig. 5.3 in the right plot, we see that our approach outperforms the Tree GP baseline. However, in the left plot, we see that Linear GP achieves superior performance. In the cart pole environment (see Fig. 5.2) the best performing version of our approach was the one without using a buffer. We see that in this environment even though our approach achieves the same performance it is clearly outperformed by both baselines as the sample efficiency is incomparable. The Lunar Lander environment is considered to be a more complex environment than the previous two. For this environment, we only compared our approach to the Tree GP baseline as in a smaller experiment (see Fig. 5.6 with three seeds the two baselines achieved a similar performance however Tree GP converged faster. The best-performing version of our algorithm didn't use a buffer. In Fig. 5.4 in the left plot, we see that our approach needs a lot of samples to achieve similar performance to the baseline however afterward it starts outperforming the baseline. In the right plot, we can clearly see the difference in the last episodes. In all environments, we see that one drawback of our approach is that it is sample inefficient in comparison with the baseline. This comes from the design of our algorithm. Even though we use a smaller population in our algorithm the individuals

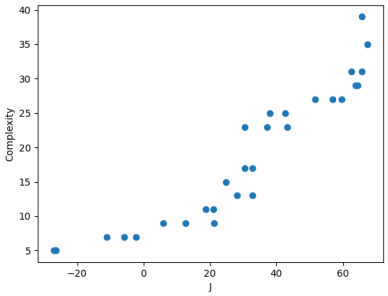


Figure 5.1: Final population average return vs. complexity

| Primitive | Complexity |
|---|---|
| Variable | 1 |
| Constant | 1 |
| + - × | 1 |
| sin | 4 |
| differentiable if | 5 |

Table 5.2: Operator complexity

must be first trained on several episodes and additionally evaluated. Furthermore, the baselines employ a dynamic strategy to reduce the number of episodes used for the fitness assessment of its individuals and in this way increase the sample efficiency. In contrast, our algorithm uses a static number of episodes to evaluate the individuals. In order to determine the complexity of an individual the primitive complexities in Table 5.2 were used. Given a final population, we can choose an individual based on our subjective importance of the complexity and average return. One example of a final population in the Lunar Lander environment can be seen in Fig. 5.1. If we take a closer look we can see it is not the most complex expression (according to Table 5.2 that performs the best. The most complex individual (see Fig. 5.5 left) has a complexity score of 39 and achieves an average return of 65.715. The second most complex(see Fig. 5.5 right) individual has a complexity score of 35 and achieves an average reward of 67.55. Given the simple structure and the small size of the individuals generated in Fig. 5.5 we can consider them simulatable as we can assume that a human being should be able to simulate them.

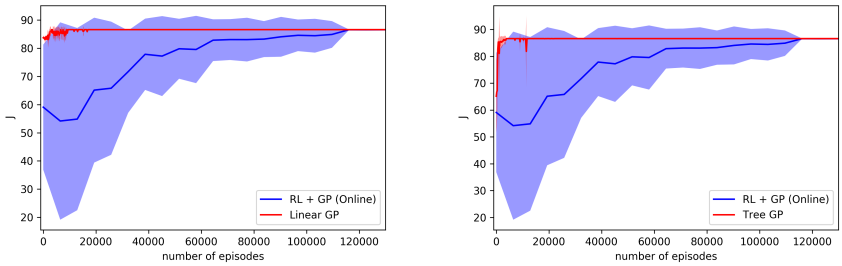


Figure 5.2: Cart Pole

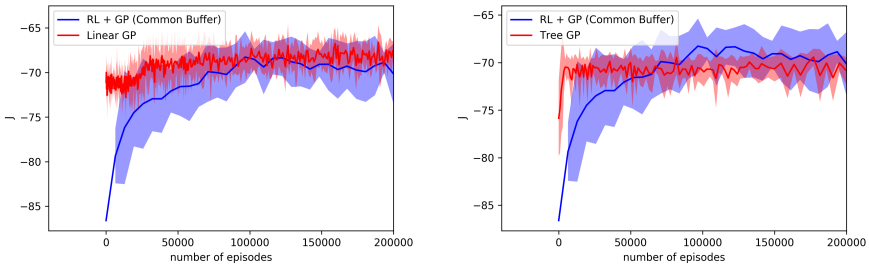


Figure 5.3: Mountain Car

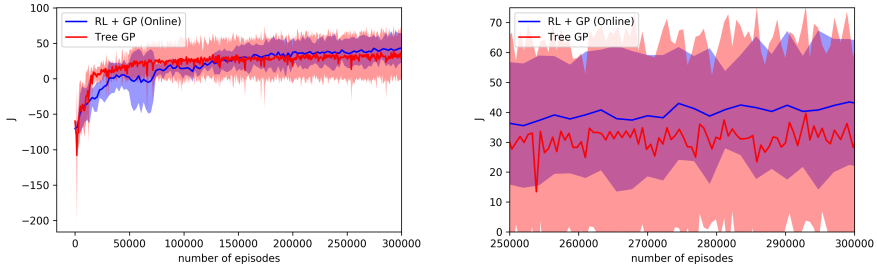


Figure 5.4: Lunar Lander

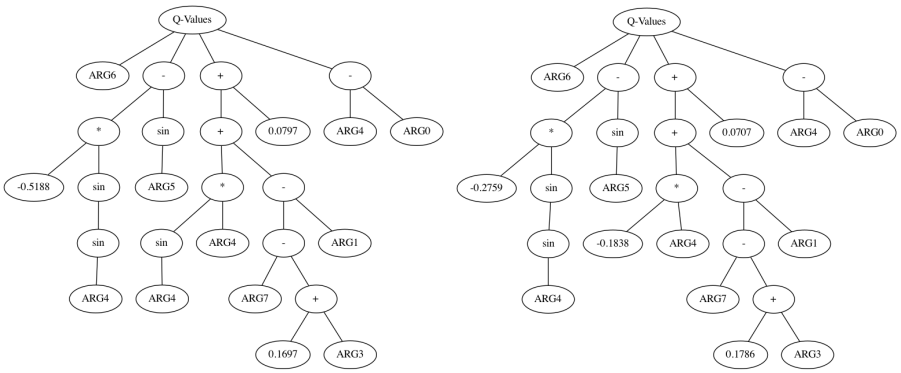


Figure 5.5: Individuals found in the Lunar Lander environment

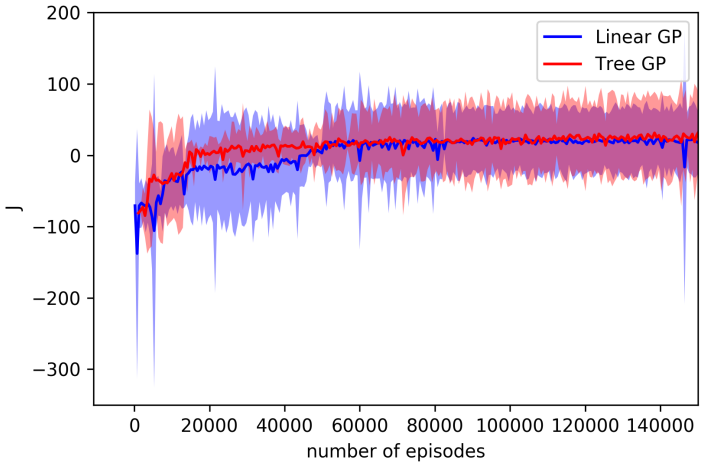


Figure 5.6: Tree GP vs. Genetic GP

# 6 Conclusion and Future Work

Motivated by the black-box nature of neural networks, in this work we presented a new algorithm that combines genetic programming with the gradient-based Deep-Q-Network algorithm. We compare our algorithm to two genetic programming baselines. Unfortunately, our current approach has several drawbacks regarding the hyperparameters tuning. The high number of hyperparameters of both algorithms and the computational cost of the evolution makes the tuning not feasible. The second problem is related to the hyperparameters of Q-Learning, such as the learning rate or the update frequency of the targets. Given that each individual represents a different architecture, a combination that works for one individual does not mean that it works well for the others. Despite this fact, we also see that our approach needs more samples to achieve the performance of the baselines. This is due to the fact that each individual in our algorithm needs to be trained and additionally evaluated. However, our algorithm outperforms the baselines on a more complex environment.

We propose several directions for future work in the context of differentiable trees in genetic programming for reinforcement learning. Given that the DQN algorithm is known to be hyperparameter-sensitive we propose to leave the value-based methods and explore the policy gradient methods. Policy gradient algorithms such as REINFORCE[50] can be easily integrated into our framework. Instead of performing Q-updates we could perform policy-gradient updates. Furthermore, another direction that looks promising is the actor injection algorithm of [22]. However, instead of using neural network actors we can use differentiable trees. The third direction is based on imitation learning. In the context of genetic programming [46] tried to use imitation learning unsuccessfully. Differentiable trees would enable us to use supervised learning to learn to imitate a teacher in the form of a neural network.

# Bibliography


[1] Riad Akrour, Davide Tateo, and Jan Peters. *Continuous Action Reinforcement Learning from a Mixture of Interpretable Experts*. 2020. DOI: 10.48550/ARXIV.2006.05911. URL: https://arxiv.org/abs/2006.05911.

[2] Alnour Alharin, Thanh-Nam Doan, and Mina Sartipi. "Reinforcement Learning Interpretation Methods: A Survey". In: *IEEE Access* 8 (2020), pp. 171058–171077. DOI: 10.1109/ACCESS.2020.3023394.

[3] Alejandro Barredo Arrieta et al. *Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward Responsible AI*. 2019. DOI: 10.48550/ARXIV.1910.10045. URL: https://arxiv.org/abs/1910.10045.

[4] Mayank Bansal, Alex Krizhevsky, and Abhijit Ogale. *ChauffeurNet: Learning to Drive by Imitating the Best and Synthesizing the Worst*. 2018. DOI: 10.48550/ARXIV.1812.03079. URL: https://arxiv.org/abs/1812.03079.

[5] "Basic Concepts of Linear Genetic Programming". In: *Linear Genetic Programming*. Boston, MA: Springer US, 2007, pp. 13–34. ISBN: 978-0-387-31030-5. DOI: 10.1007/978-0-387-31030-5_2. URL: https://doi.org/10.1007/978-0-387-31030-5_2.

[6] Cristian Bodnar, Ben Day, and Pietro Lió. "Proximal Distilled Evolutionary Reinforcement Learning". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.04 (Apr. 2020), pp. 3283–3290. DOI: 10.1609/aaai.v34i04.5728. URL: https://doi.org/10.1609%2Faaai.v34i04.5728.

[7] Greg Brockman et al. *OpenAI Gym*. 2016. DOI: 10.48550/ARXIV.1606.01540. URL: https://arxiv.org/abs/1606.01540.

[8] Diqi Chen, Yizhou Wang, and Wen Gao. “Combining a Gradient-Based Method and an Evolution Strategy for Multi-Objective Reinforcement Learning”. In: *Applied Intelligence* 50.10 (Oct. 2020), pp. 3301–3317. ISSN: 0924-669X. DOI: 10.1007/s10489-020-01702-7. URL: https://doi.org/10.1007/s10489-020-01702-7.

[9] Carlo D’Eramo et al. “MushroomRL: Simplifying Reinforcement Learning Research”. In: *Journal of Machine Learning Research* 22.131 (2021), pp. 1–5. URL: http://jmlr.org/papers/v22/18-056.html.

[10] K. Deb et al. “A fast and elitist multiobjective genetic algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017.

[11] Keith L. Downing. “Adaptive Genetic Programs via Reinforcement Learning”. In: *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*. GECCO’01. San Francisco, California: Morgan Kaufmann Publishers Inc., 2001, pp. 19–26. ISBN: 1558607749.

[12] Félix-Antoine Fortin et al. “DEAP: Evolutionary Algorithms Made Easy”. In: *Journal of Machine Learning Research* 13 (July 2012), pp. 2171–2175.

[13] Scott Fujimoto, Herke van Hoof, and David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. 2018. DOI: 10.48550/ARXIV.1802.09477. URL: https://arxiv.org/abs/1802.09477.

[14] Tanmay Gangwani and Jian Peng. *Policy Optimization by Genetic Distillation*. 2018. arXiv: 1711.01012 [stat.ML].

[15] Matthieu Geist and Olivier Pietquin. “A Brief Survey of Parametric Value Function Approximation A Brief Survey of Parametric Value Function Approximation”. In: 2010.

[16] Claire Glanois et al. *A Survey on Interpretable Reinforcement Learning*. 2021. DOI: 10.48550/ARXIV.2112.13112. URL: https://arxiv.org/abs/2112.13112.

[17] Daniel Hein, Steffen Udluft, and Thomas A. Runkler. *Interpretable Policies for Reinforcement Learning by Genetic Programming*. 2017. DOI: 10.48550/ARXIV.1712.04170. URL: https://arxiv.org/abs/1712.04170.

[18] Daniel Hein et al. “Particle swarm optimization for generating interpretable fuzzy reinforcement learning policies”. In: *Engineering Applications of Artificial Intelligence* 65 (Oct. 2017), pp. 87–98. DOI: 10.1016/j.engappai.2017.07.005. URL: https://doi.org/10.1016%2Fj.engappai.2017.07.005.

[19] Zhengyao Jiang and Shan Luo. *Neural Logic Reinforcement Learning*. 2019. DOI: 10.48550/ARXIV.1904.10729. URL: https://arxiv.org/abs/1904.10729.

[20] L. P. Kaelbling, M. L. Littman, and A. W. Moore. "Reinforcement Learning: A Survey". In: (1996). DOI: 10.48550/ARXIV.CS/9605103. URL: https://arxiv.org/abs/cs/9605103.

[21] Dmitry Kalashnikov et al. *QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation*. 2018. DOI: 10.48550/ARXIV.1806.10293. URL: https://arxiv.org/abs/1806.10293.

[22] Shauharda Khadka and Kagan Tumer. *Evolution-Guided Policy Gradient in Reinforcement Learning*. 2018. DOI: 10.48550/ARXIV.1805.07917. URL: https://arxiv.org/abs/1805.07917.

[23] Shauharda Khadka et al. "Collaborative Evolutionary Reinforcement Learning". In: (2019). DOI: 10.48550/ARXIV.1905.00976. URL: https://arxiv.org/abs/1905.00976.

[24] Jiri Kubalik et al. "Symbolic Regression Methods for Reinforcement Learning". In: *IEEE Access* 9 (2021), pp. 139697–139711. DOI: 10.1109/access.2021.3119000. URL: https://doi.org/10.1109%2Faccess.2021.3119000.

[25] William B. Langdon et al. "Genetic Programming: An Introduction and Tutorial, with a Survey of Techniques and Applications". In: *Computational Intelligence: A Compendium*. Ed. by John Fulcher and L. C. Jain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 927–1028. ISBN: 978-3-540-78293-3. DOI: 10.1007/978-3-540-78293-3_22. URL: https://doi.org/10.1007/978-3-540-78293-3_22.

[26] Abe Leite, Madhavun Candadai Vasu, and Eduardo Izquierdo. "Reinforcement learning beyond the Bellman equation: Exploring critic objectives using evolution". In: Jan. 2020, pp. 441–449. DOI: 10.1162/isal_a_00338.

[27] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2015. DOI: 10.48550/ARXIV.1509.02971. URL: https://arxiv.org/abs/1509.02971.

[28] Zachary C. Lipton. *The Mythos of Model Interpretability*. 2016. DOI: 10.48550/ARXIV.1606.03490. URL: https://arxiv.org/abs/1606.03490.

[29] Francis Maes et al. “Policy Search in a Space of Simple Closed-form Formulas: Towards Interpretability of Reinforcement Learning”. In: *Discovery Science*. Ed. by Jean-Gabriel Ganascia, Philippe Lenca, and Jean-Marc Petit. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 37–51. ISBN: 978-3-642-33492-4.

[30] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), pp. 529–533. DOI: 10.1038/nature14236. URL: https://doi.org/10.1038/nature14236.

[31] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. DOI: 10.48550/ARXIV.1312.5602. URL: https://arxiv.org/abs/1312.5602.

[32] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. DOI: 10.48550/ARXIV.1912.01703. URL: https://arxiv.org/abs/1912.01703.

[33] Aloïs Pourchot and Olivier Sigaud. *CEM-RL: Combining evolutionary and gradient-based methods for policy search*. 2018. DOI: 10.48550/ARXIV.1810.01222. URL: https://arxiv.org/abs/1810.01222.

[34] Aaron M. Roth et al. *Conservative Q-Improvement: Reinforcement Learning for an Interpretable Decision-Tree Policy*. 2019. DOI: 10.48550/ARXIV.1907.01180. URL: https://arxiv.org/abs/1907.01180.

[35] Reuven Rubinstein. “The Cross-Entropy Method for Combinatorial and Continuous Optimization”. In: *Methodology And Computing In Applied Probability* 1.2 (1999), pp. 127–190. DOI: 10.1023/A:1010091220143. URL: https://doi.org/10.1023/A:1010091220143.

[36] Olivier Sigaud. *Combining Evolution and Deep Reinforcement Learning for Policy Search: a Survey*. 2022. DOI: 10.48550/ARXIV.2203.14009. URL: https://arxiv.org/abs/2203.14009.

[37] Andrew Silva et al. “Optimization Methods for Interpretable Differentiable Decision Trees in Reinforcement Learning”. In: (2019). DOI: 10.48550/ARXIV.1903.09338. URL: https://arxiv.org/abs/1903.09338.

[38] Kenneth O. Stanley and Risto Miikkulainen. “Evolving Neural Networks through Augmenting Topologies”. In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. DOI: 10.1162/106365602320169811.

[39] Felipe Petroski Such et al. *Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*. 2017. DOI: 10.48550/ARXIV.1712.06567. URL: https://arxiv.org/abs/1712.06567.

[40] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[41] Yunhao Tang. "Guiding Evolutionary Strategies with Off-Policy Actor-Critic". In: *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*. AAMAS '21. Virtual Event, United Kingdom: International Foundation for Autonomous Agents and Multiagent Systems, 2021, pp. 1317–1325. ISBN: 9781450383073.

[42] Alexander Topchy and W. F. Punch. "Faster Genetic Programming Based on Local Gradient Search of Numeric Leaf Values". In: *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*. GECCO'01. San Francisco, California: Morgan Kaufmann Publishers Inc., 2001, pp. 155–162. ISBN: 1558607749.

[43] Nicholay Topin et al. *Iterative Bounding MDPs: Learning Interpretable Policies via Non-Interpretable Methods*. 2021. DOI: 10.48550/ARXIV.2102.13045. URL: https://arxiv.org/abs/2102.13045.

[44] Leonardo Vanneschi and Riccardo Poli. "Genetic Programming — Introduction, Applications, Theory and Open Issues". In: *Handbook of Natural Computing*. Ed. by Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 709–739. ISBN: 978-3-540-92910-9. DOI: 10.1007/978-3-540-92910-9_24. URL: https://doi.org/10.1007/978-3-540-92910-9_24.

[45] Abhinav Verma et al. "Programmatically Interpretable Reinforcement Learning". In: (2018). DOI: 10.48550/ARXIV.1804.02477. URL: https://arxiv.org/abs/1804.02477.

[46] Mathurin Videau et al. "Multi-objective Genetic Programming forÂ Explainable Reinforcement Learning". In: *Genetic Programming*. Ed. by Eric Medvet, Gisele Pappa, and Bing Xue. Cham: Springer International Publishing, 2022, pp. 278–293. ISBN: 978-3-031-02056-8.

[47] Yuxing Wang et al. *A Surrogate-Assisted Controller for Expensive Evolutionary Reinforcement Learning*. 2022. DOI: 10.48550/ARXIV.2201.00129. URL: https://arxiv.org/abs/2201.00129.

[48] Ziyu Wang et al. *Sample Efficient Actor-Critic with Experience Replay*. 2016. DOI: 10.48550/ARXIV.1611.01224. URL: https://arxiv.org/abs/1611.01224.

[49] Shimon Whiteson and Peter Stone. "Evolutionary Function Approximation for Reinforcement Learning". In: *J. Mach. Learn. Res.* 7 (Dec. 2006), pp. 877–917. ISSN: 1532-4435.

[50] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Mach. Learn.* 8.3–4 (May 1992), pp. 229–256. ISSN: 0885-6125. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.

[51] Dennis G Wilson et al. *Evolving simple programs for playing Atari games*. 2018. DOI: 10.48550/ARXIV.1806.05695. URL: https://arxiv.org/abs/1806.05695.