DEPARTMENT OF DIGITAL SYSTEMS

NCSR DEMOKRITOS
INSTITUTE OF INFORMATICS AND
TELECOMMUNICATIONS

Explainable Reinforcement Learning using Interpretable Models

Interpretability of the TD3 method on continuous action-state spaces

by

Emmanouil Lykos

Submitted

in partial fulfilment of the requirements for the degree of

Master of Artificial Intelligence

at the

UNIVERSITY OF PIRAEUS

February 2024

Author . . . . . Emmanouil Lykos

II-MSc “Artificial Intelligence”
February 20, 2024

Certified by. . . . . .

George Vouros
Professor
Thesis Supervisor

Certified by. . . . . .

Theodoros Giannakopoulos
Researcher
Member of Examination Committee

Certified by. . . . . .

Maria Dagioglou
Researcher
Member of Examination Committee

Explainable Reinforcement Learning using Interpretable Models

Interpretability of the TD3 method on continuous action-state spaces

By

Emmanouil Lykos

Submitted to the II-MSc "Artificial Intelligence" on
February 20, 2024,
in partial fulfillment of the
requirements for the MSc degree

Abstract

Deep Reinforcement Learning methods achieved new milestones in the field of Artificial Intelligence in various domains like gaming and autonomous driving. Those methods incorporate the capabilities of Deep Neural Networks into well known function approximation Reinforcement Learning methods. Although agents' performance is excellent in many cases, their decision-making mechanisms are considered black boxes, there-

fore, there is a need for software engineers, developers, domain experts, operators etc. to interpret in different levels the inner working of these methods to provide explanations.

The contribution of this thesis is a method that inherently generates interpretable models regarding the decision making of Deep Reinforcement Learning agents which are operating in environments with continuous action spaces. Initially, we will specify the problem that we are solving in a formal way and the scope of this thesis along with the current scientific contributions in that direction and what are the contributions of this thesis. Then, we will provide the necessary background knowledge in order for the reader to understand the proposed method, by firstly describing the interpretable models that we are using and then by presenting the Twin Delayed Policy Gradient method, which is the Actor-Critic Deep Reinforcement Learning method that we aim to modify in order to generate interpretable policy models. Afterwards, we specify our method which follows the mimicking paradigm and replaces the target policy neural network model with an interpretable one, along with the various modifications that we can apply. Afterwards, our method gets evaluated in various environments using Gymnasium and gets compared with the primary policy model that was trained from the original Twin-Delayed Policy Gradient method, both in terms of the learning curve and the standalone performance of the generated primary neural network policy model and the interpretable policy model mimicking it, in order to evaluate interpretations' quality. The performance of agents with the interpretable method is shown to be competitive with comparison to the ones generated from the original non-interpretable method, however with limitations. Last but not least, we justify the results, draw our conclusions and provide directions for future work in this field.

Thesis Supervisor: George Vouros
Title: Professor at University of Piraeus

Acknowledgments

Initially, I want to thank professor George Vouros for his continuous support and feedback through the course of this thesis. Furthermore, I want to thank all the professors I met through the courses of the present MSc for their dedication to expand our horizons in the field of Artificial Intelligence in order to learn new things and also enable us learn further. Last but not least, I want to thanks my parents for their continuous support through my whole educational journey.

Contents

List of Figures

List of Figures

List of Tables

List of Tables

1 Introduction

Reinforcement Learning[1] is the field of Machine Learning where the agents determine their optimal policy by interacting directly with their environment with ultimate goal to maximize their expected reward. A high-level illustration of how a reinforcement learning agent works step-wise, can be found in Figure 1 where the Agent performs an action, observes its next state and the feedback(Reward) from the environment, updates its policy accordingly and repeats this process.

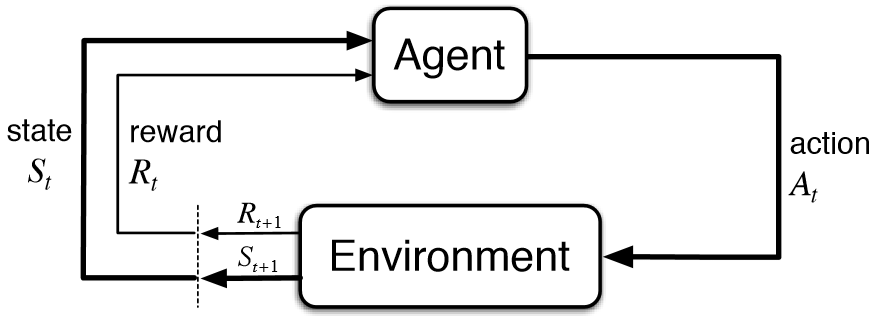


Figure 1: Reinforcement Learning agent's mechanics

Reinforcement Learning approaches, nowadays, have applications in many domains, from simple to more complex, from simple mazes to robotics and gaming, for instance. Initially, the first approaches were made to solve small-scale Markov Decision Processes[2], thus, it served good to solve them via Dynamic Programming using Bellman Equation[3]. Afterwards, in order for agents to be able to handle environments with discrete state and action spaces and increase their reward incrementally per episode or per step, Monte Carlo[4] and Temporal Difference methods[5] were introduced, respectively. Then, because the state-action space might not be discrete or small, the aforementioned methods were tweaked in a way that the agents do not need to keep a map between all the states or state-action combinations and their expected value. Instead, this mapping is performed via a differentiable function that takes as input the

state or both the state and the action, hence, the agent needs to store only the function's parameters. Those methods are called function approximation methods[6] and they were the stepping stone for two things. Firstly, because those methods can use any differentiable function, neural networks can also be used as an approximator, like in [7], where the authors are using Convolutional Neural Networks in order to build an agent that can perform well in Atari games. However, this approach can also be applied using fully connected neural networks for other environments like MuJoCo. Secondly, there are Policy Gradient[8] and Actor-Critic[9] methods, where instead of learning the optimal policy via the state or state-action value functions, the agent (actor) learns directly the policy by learning a probability distribution on actions, given agent's current state, and the critic evaluates the policy providing feedback to the actor about the optimality of the policy. Finally, note that policy gradient methods can use Deep Neural Networks in order to generate the policy distribution, e.g. as done in [10–12].

Although deep neural networks models provide solutions to a variety of problems [13], including functions' approximations or modeling of environmental aspects, in reinforcement learning, their main problem arises when one needs to interpret their decision making. This happens because deep neural network models are considered black boxes due to the way their parameters are tuned and interact, making it very complicated to inspect its internal workings to justify somehow their decisions. Even if it can be done, its impossible to provide a simple and convincing explanation.

Deep Reinforcement Learning, incorporating neural network models is not an exception to the aforementioned problem. In fact, in real-world applications, especially in safety critical ones, it is of paramount importance for researchers to extract explanations of the deep reinforcement learning agents, despite their excellent performance. The reasons for this, are firstly that users and researchers will be more confident if they know the logic that the neural network follows in its decisions, and secondly, it will be easier to identify any small errors that may result to huge consequences if they hap-

pen in a real-world setting. Hence, it is clear that there is a need to design and develop reinforcement learning methods that can generate explainable agents.

Before proceeding to the work that was done in this thesis, it is needed to define the basic terms and concepts around Explainable Reinforcement Learning[14]. Initially, $interpretability$ is defined as the ability of a system to provide explanations from an interpretable model and those explanations are called interpretations. Furthermore, $explainability$ is defined as the ability of the system to provide surface representations from interpretations. Therefore, in order to be able to extract surface representations and make a model explainable, it is needed to have an interpretable model along with an explanation logic that will filter out the interpretations and keep the most useful ones to meet transparency requirements. Last but not least $transparency$ is defined as the ability of the model to provide explanations that are understandable by the people that will see them. In this thesis, we will focus only on making Deep Reinforcement Learning models interpretable.

In order to generate an interpretable model of a Deep Reinforcement Learning model, there are mainly two ways: distillation or mimicking. The distillation process gathers knowledge from the trained Deep Reinforcement Learning agent, exploiting any of the constituent Deep Reinforcement Learning models in a direct way through a process of transforming Deep Reinforcement Learning models' elements into interpretable models. The mimicking process monitors the interaction of the Deep Reinforcement Learning agent with the environment, and gathers interaction samples, recording agent decisions, state transitions, rewards and consulting Deep Reinforcement Learning assessed values in order to train the interpretable model. Overall, the distillation paradigm aims at exploiting the knowledge acquired by the DRL models via model inspection facilities, while the mimicking method produces interpretable models without exploiting the inner working components of DRL agents.

Many methods are developed regarding Interpretable Deep Reinforcement Learning[14] with the purpose to demystify the internal logic of Reinforcement Learning agents. In this thesis, we will focus to interpret the decisions of Deep Reinforcement Learning agents that are trained to perform in environments with continuous action-spaces. These agents are trained in most cases using policy-gradient or actor-critic methods, because they generate a probability distribution on possible actions, or choose directly the action, instead of giving a value to every state-action pair. Such a method is the Twin-Delayed Policy Gradient method[11]. Our approach is based on the work cited in [15] where they replace the target neural network models of the Deep-Q-Networks method with interpretable Boosting Trees, in order to generate interpretations from the neural network value functions. Because, we want to apply this method to agents that interact with continuous action spaces, we adopt a similar approach to interpret a Twin Delayed Policy Gradient[11] agent. For this purpose we are using decision tree-based models, which are interpretable by nature (inherently interpretable): Every internal tree node represents a check about a certain condition and every leaf node represents the decision, therefore, for each decision that the tree makes we can extract the predicate that corresponds to it.

The structure of the present thesis is the following. In the second section, we will provide related work with what we are doing and present the contributions of our work. In the third section we cite the background theory that is necessary to understand the rest of this thesis. This section is split into two parts. The first one presents the various interpretable models that will be used. The second one, presents the Twin-Delayed Policy Gradient(TD3)[11] method, which is an Actor-Critic method to be made interpretable using the mimicking paradigm, in order to produce an interpretable agent that would work in environments that have continuous action spaces. Then, in the fourth and fifth section we will describe our methods and their experimental evaluation along with the justification of the results, respectively. Finally, we draw our conclusions and provide

directions for further work.

2 Related Work

One of the first approaches to incorporate Decision Trees in Reinforcement Learning methods to explain the resulting policy can be seen in [16]. This is done by employing Decision Trees in order to approximate the Q-values. The method runs like the Q-Learning algorithm[17], but the complex part is how the Decision Tree is updated. In the case of the greedy action selection, it calculates the Q-value difference between the current and the previous timestep, ΔQ, using the Bellman equation, and updates the previous timestep's Q-value accordingly. Then, the agent adds the ΔQ value to the previous timestep state's node history. A split check is made, and a node split is performed if the average ΔQ is smaller than two times the standard deviation of ΔQ. Overall, this approach shows that tree-based approaches can work on Reinforcement Learning tasks and especially in Q-Learning. However, the authors do not investigate method's applicability on Deep Reinforcement Learning tasks.

Another approach is the Conservative Q-Improvement algorithm[18]. This algorithm utilizes a decision tree to estimate the Q-value function by keeping tree sizes small, hence, easier interpretable and transparent. Agent's policy is represented by a single decision tree and its whole training procedure is virtually the same as in Stochastic Gradient Trees(SGTs)[19], that will be presented later, where in each timestep it is determined whether we should split, update or do nothing to the invoked leaf node, with two main differences. The first is that CQI has one tree that predicts all the available actions while SGTs need to have one tree for each action. The second one is the criterion regarding whether a leaf node is eligible for splitting. In SGTs, student t-test is applied on whether the loss per update is equal to zero or not. On the other hand, in CQI trees, a split is performed if the expected discounted future reward for the new policy after the split increases above a specific dynamic threshold, and that's why this method is called conservative, because it wants to keep the tree as small as possible, assuming

that a smaller tree is more interpretable than a bigger one. The problem with those approaches is that they work well for problems with discrete-action spaces, but they are not designed for problems with continuous-action spaces.

Programmatically Interpretable Reinforcement Learning(PIRL)[20] is a Reinforcement Learning framework that explains Deep Reinforcement Learning agents' policies by translating them to a high-level, domain-specific programming language in order to be in a form that is readable by humans. Also, the resulting program might not be as effective as the neural policy, but at least it can be configured easily by experts. More formally, having a Partially Observable Markov Decision Process(POMDP)[21], a functional language and given a policy sketch that syntactically defines a set of programmatic policies, the main goal is to find a program that maximizes the long-term reward. In order to avoid brute-force search, A. Verma et al.[20] proposes Neurally Directed Program Search(NDPS) which performs local search on available policies, guided by an oracle policy, which is extracted from a fully trained Deep Reinforcement Learning method, like DDPG[10]. NDPS evaluates the candidate programmatic policies by measuring their distance with the output of the oracle policy in a set of interesting histories which are enriched with additional histories generated by the programmatic policy. Evaluation of NDPS policies showed that they can have smoother trajectories, and better generalization abilities compared to the policies discovered by DRL. However, no evaluation of the explainability or interpretability of PIRL is provided, with or without human subjects. Although this method can work for environments with continuous action spaces, it cannot work in an online manner because the policy oracle needs to be defined beforehand.

Another mimicking work can be seen in the work of Y. Coppens et al.[22] where they utilize Soft-Decision Trees(SDTs)[23] in order to explain the policy of a fully trained model. SDTs are a classification model which is a combination of binary Decision Trees and perceptrons. More specifically, each node contains internally a perceptron model

that gives the probability on going to the right or left node for internal nodes, while leaf nodes' learn softmax distributions over possible classes. In [22] the authors use a pre-trained Reinforcement Learning model and try to make the SDT to mimick its policy by fitting it with a number of state-action pairs extracted from the model. The evaluation of this approach was done on Mario AI Benchmark[24] using a pre-trained actor-critic model with Proximal Policy Optimization(PPO)[25]. It was determined that SDTs could explain the extracted policy, but it could not perform standalone in the same level as the PPO agent. Moreover, the interpretability of SDTs is questionable because they can explain the features' weights on internal node's decision, but leaf nodes' decision is opaque because it is determined by a softmax distribution. Also, because SDTs want to have good generalization properties, for some inputs the SDT output differs significantly with the one of the PPO policy, thus, the fidelity of SDT decreases (i.e. it results to decisions that are different to those of teh original model). Finally, although this method can provide a good explanation in problems that have spatial input it is not known whether they can produce good interpretation in a non-spatial setting.

The work that is more closely to the approach taken in this thesis, can be seen in the paper of A. Kontogiannis et al.[15]. In this approach the authors created an interpretable DQN agent by replacing the target Q-network of the DQN[7] and Double DQN[26] algorithms with an interpretable model and specifically Gradient Boosting Regressor(GBR) trees[27], which are also presented and utilized in this thesis. Hence, this approach is also considered inherently interpretable because it replaces a neural network model from the original algorithm with an interpretable one. The difference with PIRL[20] and Y. Coppens et al.[22] papers, is that in this work the neural network policy model is not used as a pre-trained oracle to which the interpretable model fits its decisions given the states. The interpretable model is learnt in an online manner together with the neural network model, thus, the authors are able to also monitor agent's training. This approach was evaluated in the Air Traffic Management Challenge and showed ini-

tially that the neural network model has competitive performance even though it used the interpretable model as the target Q-network. Furthermore, it was shown that the interpretable model has high fidelity in terms of the decision that it makes in comparison to that of the neural network model. However, due to overfitting, high fidelity does not always ensures that the interpretable model will perform well in the given environment, thus, its way of making decisions will not give any useful decisions and interpretations overall. This aspect is not investigated in this work. Furthermore, this approach is not designed for problems with continuous action-spaces as DQN and Double DQN algorithms are designed for problems with discrete action-spaces.

3 Background Knowledge

In this section, we will describe the theory behind the components that the proposed method uses in order to understand the technical aspects of this thesis. Initially, we present the theory behind the Stochastic Gradient Trees and Gradient Boosting Regression Trees which are the interpretable models that will be used to make Twin-Delayed Policy Gradient algorithm interpretable. Afterwards, we will provide information about Reinforcement Learning Actor-Critic methods in general and specifically about the Twin-Delayed Policy Gradient algorithm.

3.1 Interpretable Models

3.1.1 Stochastic Gradient Trees

Stochastic Gradient Trees[19] are state-of-the-art incremental models that can be used for both regression and classification and their updates are based on Stochastic Gradient Descent[28] which is widely used on Neural Networks. Its main contributions, in comparison with similar models like Hoeffding Trees[29], is that those models are general and scalable because they can optimize to arbitrary differentiable loss functions and because they do not need a number of trees to construct the final model in order to predict a single value, but only one, i.e. they are not ensemble models, respectively.

The training method of the tree consists of two components. The first is a method to evaluate a possible split and compute leaf node predictions using only gradient information. The second, is an approach to determine whether there is enough information to split a leaf node where in particular, it is based on t-tests. In Stochastic Gradient Trees one-sample t-tests are used rather than hypothesis tests based on Hoeffding inequality, because it is needed for the SGT to use loss functions with unbounded gradients, in order to determine whether enough evidence has been observed to justify splitting a

node.

In order to see everything in a more formal way, let's assume that the tree has a loss function $l(y, \hat{y})$ where y is the ground truth label and $\hat{y}$ is tree's prediction which is equal to $y = \sigma(f(\mathbf{x}))$ where σ is a user-defined activation function, f represents the SGT, and $\mathbf{x}$ the given instance. Because Stochastic Gradient Trees are updated incrementally, for each update timestamp t we want to minimize the expected loss from last update timestamp r until current timestamp t:

$$\mathbb{E}[l(y, \hat{y})] \approx \frac{1}{t-r} \sum_{i=r+1}^{t} l(y_i, \hat{y}_i)$$

Because Stochastic Gradient Trees are trained incrementally, they change their structure when each sample arrives, hence, we need to find a modification of the tree $u : \mathcal{X} \rightarrow \mathbb{R}$ that minimizes the loss function. The modification can be splitting a leaf node, updating its prediction value or doing nothing due to insufficient evidence to determine the best split. More formally we have the following regarding the updates at a given timestep:

$$f_{t+1} = f_t + argmin_u[\mathcal{L}_t(u) + \Omega(u)],$$

where f_t is the state of the tree at timestep t,

$$\mathcal{L}_t(u) = \sum_{i=r+1}^{t} l(y_i, f_t(\mathbf{x}_i + u(\mathbf{x}_i))$$

and

$$\Omega(u) = \gamma|Q_u| + \frac{\lambda}{2} \sum_{j \in Q_u} v_u^2(j)$$

The Ω term works as a regularizer of a possible modification, $Q_u \subset \mathbb{N}$ is the set of unique identifiers for the new leaf nodes that participate in modification u, and $v_u : \mathcal{N} \to \mathbb{R}$ is the difference in the predictions between the given node and its parent node. In Ω, the first term penalizes the creation of a new node and the second one penalizes big changes in the prediction value of new nodes in relation with their parent nodes. Thus, we want our tree to have a minimal number of nodes and keep its prediction range low.

In order to keep consistency between the splitting criterion and leaf prediction values with the loss function, the loss function $\mathcal{L}_t(u)$ is approximated using a Taylor expansion:

$$\mathcal{L}_t(u) \approx \sum_{i=r+1}^{t} [l(y_i, f_t(\mathbf{x}_i)) + g_i u(\mathbf{x}_i) + \frac{1}{2} h_i u^2(\mathbf{x}_i)]$$

where g_i and h_i are the first and second derivatives, respectively, of l with respect to $f_t(\mathbf{x}_i)$. Because the first term is a constant, we can simplify the expansion to:

$$\begin{aligned} \Delta\mathcal{L}_t(u) &= \sum_{i=r+1}^{t} [g_i u(\mathbf{x}_i) + \frac{1}{2} h_i u^2(\mathbf{x}_i)] \\ &= \sum_{i=r+1}^{t} \Delta l_i(u) \end{aligned}$$

which describes the loss change due to modification u. In order to find the best split, this function is calculated for every possible split in order to find the one that gives the maximum reduction of the loss function. Also, for each potential split we need to find the prediction values of the new leaf nodes. Note that at time t where an instance $\mathbf{x}_t$ arrives, the modification(if any) will be applied only to the leaf node that the particular instance falls to. In order to find the optimal values $v_u(j)$, let's define firstly a potential

split:

$$u(\mathbf{x}) = \begin{cases} v_u(q_u(\mathbf{x})), & \text{if } \mathbf{x} \in \text{Domain}(q_u) \\ 0, & \text{otherwise} \end{cases}$$

where q_u maps an instance in the given leaf node to an identifier for a leaf node that would be created if the split were performed and $\mathbf{x}$ will be falling to. Also, we denote the set Q_u, which contains the identifiers for leaf nodes that will be created after the split u. Then, we define the set I_u^j which is the set of instances that would reach the new leaf node identified by j. Thus, loss difference can be written as:

$$\begin{aligned} \Delta\mathcal{L}_t(u) &= \sum_{j \in Q_u} \sum_{i \in I_u^j} [g_i v_u(j) + \frac{1}{2} h_i v_u^2(j)] && \Leftrightarrow \\ &= \sum_{j \in Q_u} [(\sum_{i \in I_u^j} g_i) v_u(j) + \frac{1}{2} ((\sum_{i \in I_u^j} h_i) v_u^2(j)] \end{aligned}$$

Now, the optimal $v_u(j)$ can be found by adding to the above the corresponding term from Ω,

$$\sum_{j \in Q_u} [(\sum_{i \in I_u^j} g_i) v_u(j) + \frac{1}{2} ((\sum_{i \in I_u^j} h_i) v_u^2(j)] + \frac{\lambda}{2} v_u^2(j)$$

and setting its derivative with respect to v_u equal to zero and solving by v_u,

$$\begin{aligned} 0 &= (\sum_{i \in I_u^j} g_i) + (\lambda + \sum_{i \in I_u^j} h_i) v_u(j) && \Leftrightarrow \\ v_u^*(j) &= -\frac{\sum_{i \in I_u^j} g_i}{\lambda + \sum_{i \in I_u^j} h_i} \end{aligned}$$

The above equation estimates the quality of a possible split but it does not provide any indication on whether we should perform the split. Because Stochastic Gradient Trees

do not want to bound their gradients in order to keep their generality, they are using Student's t-test to determine whether a split should be made. The t statistic is computed by

$$t = \frac{\overline{L} - \mathbb{E}[\overline{L}]}{s/\sqrt{n}}$$

where $\overline{L}$ is the mean change in loss if the split were applied, s is the sample standard deviation of L_i and, under null hypothesis, $\mathbb{E}[\overline{L}]$ is assumed to be zero, i.e. it is assumed that the split does not result in loss change. A p value can be computed using the inverse cumulative distribution function of the t-distribution and, if p is less than δ, the split can be applied.

The way that the split happens differs whether we are splitting a feature with discrete or continuous value range. In the former case, we just create a branch of the split node to each possible value, hence, if a feature of a leaf node has three distinct values and is about to be split, we create 3 branches, one for each possible value. In the latter case, given the upper and lower bounds of the feature's values, which the user gave or they were determined, we discretize it by performing equal width binning and then handle the discrete values as an ordinal because the algorithm finds the best point to perform a binary split.

For batch learning problems, epochs are introduced in the method to update the Stochastic Gradient Tree using multiple passes over the training data, similar to the way those are used in the Neural Networks training procedure.

3.1.2 Gradient Boosting Regression Trees

What boosting algorithms are doing is that they use a weak learning procedure-that performs a bit better than random guessing-a number of times in order for their combination to provide a strong learning procedure. There are several approaches for that

like Random Forests[30], XGBoost[31], AdaBoost[32] etc. However, Gradient Boosting algorithms construct additive models by sequentially fitting a simple parameterized function(base learner) to current pseudo-residuals by least-squares at each iteration. The pseudo-residuals are the gradient of the loss function being minimized, with respect to the model values at each training data point, evaluated at current step.

In every Machine Learning regression problem, what we want to do, is given some label variables y_i and some feature vectors $\mathbf{x} = \{x_1, x_2, ..., x_n\}$, in a training sample denoted as $\{y_i, \mathbf{x}_i\}_i^N$, to find a general function F that fits to the given training sample, even if the instances are derived from various distributions. This is done by firstly defining a loss function $\Psi(y, F(\mathbf{x}))$ that we want our model F to minimize given the training sample. This loss function can be any differentiable function such as mean-squared error or cross-entropy loss. In case of boosting, $F(\mathbf{x})$ is expanded as follows:

$$F(\mathbf{x}) = \sum_{m=0}^{M} \beta_m h(\mathbf{x}; \mathbf{a}_m)$$

where $h(\mathbf{x}; \mathbf{a}_m)$ is the "weak" model of the ensemble model, $\mathbf{a_m}$ are its parameters and β_m is the weight of this model, with these parameters, to the final decision. Thus, these ensemble models train each base learner h sequentially by having an initial guess $F_0(\mathbf{x})$, and then for $m = 1, 2, ..., M$

$$(\beta_m, \mathbf{a}_m) = argmin_{\beta, \mathbf{a}} \sum_{i=1}^{N} \Psi(y_i, F_{m-1}(\mathbf{x}_i) + \beta h(x_i; \mathbf{a})) \tag{1}$$

and

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}_i) + \beta_m h(x_i; \mathbf{a}_m) \tag{2}$$

The above is easier to be solved in two steps by firstly finding base-learner's parameters $\mathbf{a}_m$ by least-squares:

$$\mathbf{a}_m = argmin_{\mathbf{a},\rho} \sum_{i=1}^{N} [\tilde{y_{im}} - \rho h(x_i; \mathbf{a})] \tag{3}$$

to the current pseudo residuals

$$\tilde{y_{im}} = -[\frac{\theta \Psi(y_i, F(x_i))}{\theta F(x_i)}]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})} \tag{4}$$

and secondly we find β_m by solving a single parameter optimization problem as follows:

$$\beta_m = argmin_{\beta} \sum_{i=1}^{N} \Psi(y_i, F_{m-1}(\mathbf{x}_i) + \beta_m h(x_i; \mathbf{a}_m)) \tag{5}$$

Specifically, in Gradient Boosting Regression Tree[27], a base learner is in fact a regression tree with L leaves. The parameters of this learner are the split nodes along with their decision makers, which in the end, split the space into L disjoint regions $\{R_{lm}\}_{l=1}^{L}$ at iteration m. At each iteration we create a new L-terminal node tree. Each terminal node keeps as a prediction a constant value which is equal to $\tilde{y}_{lm} = mean_{\mathbf{x}_i \in R_{lm}}(\tilde{y}_{im})$ where $\tilde{y}_{im}$ is equal to the pseudo residual shown before. Now, as first step we need to find the base learner's hyperparameters where in our case each partition is determined in a top down "best first" manner using a least squares splitting criterion. Then, we can solve $\beta_m = argmin_{\beta} \sum_{i=1}^{N} \Psi(y_i, F_{m-1}(\mathbf{x}_i) + \beta_m h(x_i; \mathbf{a}_m))$ separately for each region R_{lm} and because those nodes predict a constant value within each region we can just change the problem and solve the following simple problem:

$$\gamma_{lm} = argmin_{\gamma} \sum_{\mathbf{x}_i \in R_{lm}} \Psi(y_i, F_{m-1}(\mathbf{x}_i) + \gamma) \tag{6}$$

Thus, the model F_{m-1} is then updated in each corresponding region as follows because every leaf node predicts a fixed value:

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \nu * \gamma_{lm} 1(\mathbf{x} \in R_{lm}) \tag{7}$$

where ν is the learning rate of the procedure. Thus the algorithm is the following, where various loss metrics can be applied with mean squared error be the most common one:

Algorithm 1 Gradient Boosting Regressor

1: $F_0(\mathbf{x}) = argmin_\gamma \sum_{i=1}^{N} \Psi(y_i, \gamma)$
2: **for** $m = 1$ to M **do**
3: $\quad \tilde{y}_{im} = -[\frac{\partial \Psi(y_i, F(\mathbf{x}_i))}{\partial F(\mathbf{x}_i)}]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}$, $i = 1, ..., N$
4: $\quad \{R_{lm}\}_1^L = L -$ terminal node tree$(\{\tilde{y}_{im}, \mathbf{x}_i\}_1^N)$
5: $\quad \gamma_{lm} = argmin_\gamma \sum_{\mathbf{x}_i \in R_{lm}} \Psi(y_i, F_{m-1}(\mathbf{x}_i) + \gamma)$
6: $\quad F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \nu \cdot \gamma_{lm} 1(\mathbf{x} \in R_{lm})$
7: **end for**

One extension of the above algorithm, in order to improve performance, is the induction of randomness by training the m^{th} tree with a subset of the training instances instead of the entirety of them. The algorithm is adjusted as follows(we just take a subsample to fit base learner and compute the model update for current iteration):

Algorithm 2 Stochastic Gradient Boosting Regressor

1: $F_0(\mathbf{x}) = argmin_\gamma \sum_{i=1}^{N} \Psi(y_i, \gamma)$

2: **for** $m = 1$ to M **do**

3: $\quad \{\pi(i)\}_1^N = \text{rand\_perm}\{i\}_1^N$

4: $\quad \tilde{y}_{\pi(i)m} = -[\frac{\partial \Psi(y_{\pi(i)}, F(\mathbf{x}_{\pi(i)}))}{\partial F(\mathbf{x}_{\pi(i)})}]_{F(\mathbf{x})=F_{m-1}(\mathbf{x})}, i = 1, ..., \tilde{N}$

5: $\quad \{R_{lm}\}_1^L = L - \text{terminal node tree}(\{\tilde{y}_{\pi(i)m}, \mathbf{x}_{\pi(i)}\}_1^N)$

6: $\quad \gamma_{lm} = argmin_\gamma \sum_{\mathbf{x}_{\pi(i)} \in R_{lm}} \Psi(y_{\pi(i)}, F_{m-1}(\mathbf{x}_{\pi(i)}) + \gamma)$

7: $\quad F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \nu \cdot \gamma_{lm} 1(\mathbf{x} \in R_{lm})$

8: **end for**

3.2 Actor-Critic Methods and Twin-Delayed Policy Gradient Algorithm

Actor-Critic methods[9] are a subclass of Policy Gradient methods[8] that their goal is to directly determine the optimal policy(the one that maximizes the expected reward) using the one-step expected return in addition to the policy model itself. Therefore, Actor-Critic methods have two components, the “actor” which references to the learned policy, and the ”critic” which references to the learned state value function that gets trained usually using TD-learning[5, 6]. Those methods help us address problems that have continuous action space because they generate a distribution in the action space rather than a score for each state-action pair. A graphical representation on how Actor-Critic methods work can be shown in Figure 2 where the actor, through its policy model, decides which action should be taken and the critic who through its value function informs the actor how good was the action taken and how the actor should adjust.

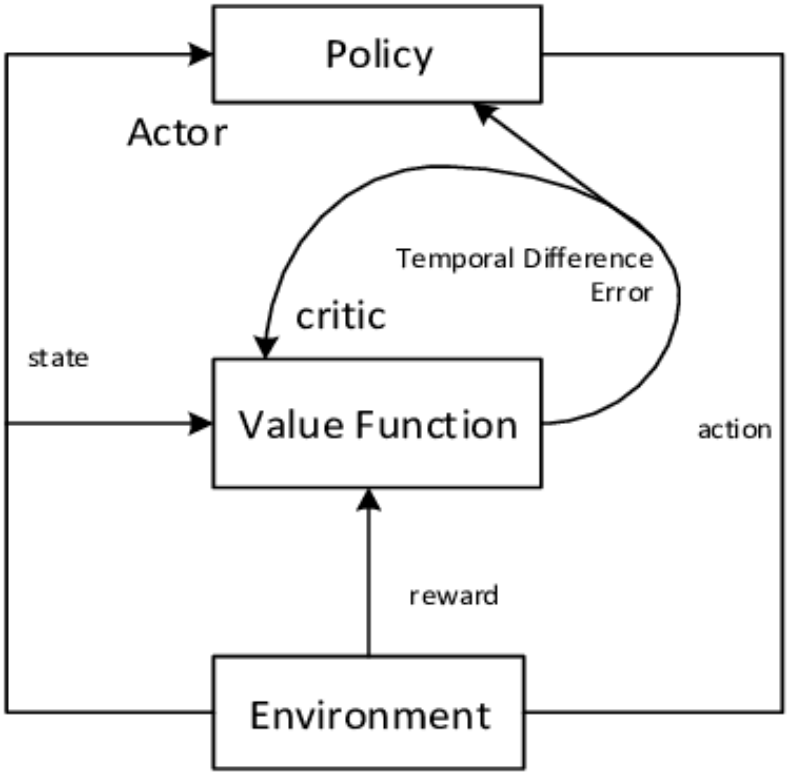


Figure 2: General framework of Actor-Critic methods

Twin-Delayed Policy Gradient(TD3)[11] is an actor-critic method that builds on top of Deep Deterministic Policy Gradient[10] which is a Policy Gradient version of Deep Q-Learning[7] method. In addition, what TD3 wants to address, that are reasons also to convergence to suboptimal policies, are overestimations in value estimates, high variance and overfitting to value function's peaks. Those problems are relevant to the critic of the algorithm. About overestimation bias, approaches for Double Deep-Q-Learning[26] and Double Q-Learning[33] were employed initially, where in the former, an additional target network is used to determine the value of the state-action function and gets updated periodically based on the primary critic network. In the latter approach, two state-action value functions are used where at each step one is picked to select the next action and the one is updated based on the other. However, none of the aforementioned approached gave the desired results. Therefore, clipped Double Q-

Learning is used where it assumes that a value function suffering from overestimation bias can work as an upper bound to the true value function in order to favor underestimations which are not persistent because agents do not want to stay at a state that give small estimations. This is achieved by updating the critic networks by determining the target value using the following formula:

$$y = r + \gamma min_{i=1,2} Q'_{\theta_i}(s', \pi_\phi(s'))$$

where θ_i, $i = 1, 2$ are the weights of critic networks and ϕ are the weights of the policy network and it is clear from the formula that the target is determined using the most pessimistic prediction between two critics in order to prevent overestimation bias.

Variance Reduction is ensured with target networks and delayed policy updates. Variance leads to a noisy gradient that accumulates errors which not only affects performance but also results in slower convergence. One way to mitigate it, is to update target network less frequently than every step which is done in Deep Q-Network for example in order to have more stable learning. Another way to reduce variance in an actor-critic method is to update the policy(actor) less frequent than the state-action value function(critics). This helps preventing policy divergence because when updating the actor less frequently than the critics, the critics have reduced their error as much as possible. So, in summary we want at each policy model update to have the best critics that we can and then update the policy utilizing them. Also, that way it is ensured that policy updates are done when the outputs of critic networks are changed significantly.

In order to prevent agent's policy network's overfit to value estimates peaks and also promote exploration, stochasticity should be added to the agent. Also, the critic is susceptible to errors if it is updated by a deterministic policy. For that, a SARSA-like regularization strategy[1] is used where its basic principle is that similar actions, should have similar value. In order to do that, the value function gets fit in a small area around

target action, so the fitted function will be more smooth. In practice this is done by adding bounded random noise to target policy's selected action. More formally, the target update is modified as:

$$y = r + \gamma Q_{\theta'}(s', \pi_{\phi'}(s') + \epsilon),$$
$$\epsilon \sim \text{clip}(\mathcal{N}(0, \sigma), -c, c)$$

Considering and combining the aforementioned tweaks, the TD3 algorithm can be seen in **Algorithm 3**. More specifically, in lines 1-3 the initialization of the agent happens by initializing the invoked neural networks, their target ones and the replay buffer. Then, for each timestep the agent in line 5 selects an action using the main policy network with additional noise ϵ in order to promote exploration, performs it and stores the transition to replay buffer as seen in line 6. Afterwards, from lines 7 to 10, the agent takes a sample from the replay buffer, finds the next action a' from the sampled transitions and applies clipped Double Q-Learning to find the target value in order to calculate critics' loss and update them accordingly. Then, from lines 11 to 13, the actor policy network gets updated according to the predefined frequency d, using the chain rule in the weights of first critic network θ_1. Last but not least, from lines 14 to 16, the target networks are getting updated in order to keep the errors small.

Algorithm 3 Neural Networks TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2} and actor network π_ϕ with random parameters θ_1, θ_2, ϕ
Initialize target networks for the critics $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$, $\phi' \leftarrow \phi$
Initialize replay buffer $\mathcal{B}$
for $t = 1$ **to** T **do**
    Select action with exploration noise $\alpha \sim \pi_\phi(s) + \epsilon$, $\epsilon \sim N(0, \sigma)$ and observe reward r and new state s'
    Store transition tuple (s, α, r, s') in $\mathcal{B}$
    Sample mini-batch of N transitions (s, α, r, s') from $\mathcal{B}$
    $\widetilde{\alpha} \leftarrow \pi_{\phi'}(s') + \epsilon$, $\epsilon \sim \text{clip}(N(0, \sigma), -c, c)$
    $y \leftarrow r + \gamma min_{i=1,2} Q_{\theta_i'}(s', \widetilde{\alpha})$
    Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, \alpha))^2$
    if t mod d == 0 **then**
        Update ϕ by the deterministic policy gradient:
        $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_\alpha Q_{\theta_1}(s, \alpha)|_{\alpha=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
        Update target networks:
        $\theta_i' \leftarrow \tau\theta_i + (1 - \tau)\theta_i'$
        $\phi' \leftarrow \tau\phi + (1 - \tau)\phi'$
    end if
end for

4 Interpretable Twin-Delayed Policy Gradient

In order to make our TD3 agent interpretable, one of the two following approaches could be followed. In the first, both the primary and target policy networks will be replaced with an interpretable incremental model, like SGT, according to the distillation paradigm. These models will get updated in the same manner as neural networks by configuring the loss function accordingly. In the second approach, we can interpret directly the primary policy model, the target policy model, the primary critics or the target critics using the mimicking paradigm. In this thesis, the target policy network is handled as a mimic of the policy network because we want to extract the policy itself, thus, there was no point at extracting critics' interpretations. Additionally, in the mimicking approach, the primary policy network could not be replaced by an interpretable model because this model gets updated incrementally. This method was introduced for interpretable Deep Q-Networks version[15], where in order to update the target model, the agent samples experience from replay buffer and fit the target from that. It is done in that manner because the knowledge gathered in the parameters of one or more neural networks cannot be transferred directly to an interpretable tree model. In this thesis, the second method will be presented and investigated because it is a straightforward method to make the original TD3 method interpretable and also we are able to use non-incremental interpretable models because the target captures only the current instance of the primary model. Furthermore, we can create different snapshots of the interpretable model during training, so as to be able to examine the training progress of the method.

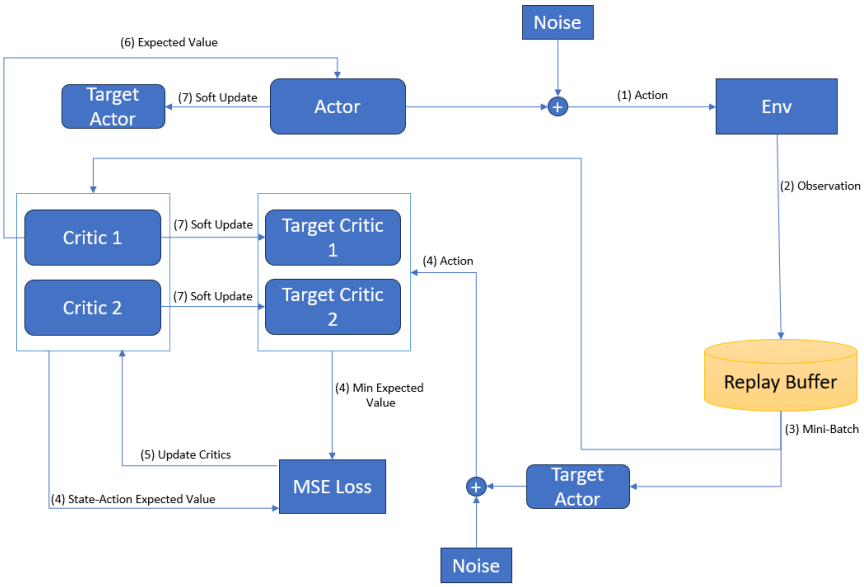
The Interpretable Twin-Delayed Policy Gradient method using mimicking, can be seen in **Algorithm 4**. Additionally, an illustration of its differences with the original TD3 algorithm can be seen in Figure 3. In this algorithm, it is necessary to explain three modifications. Firstly, from lines 13 to 22, it is seen that the primary policy neural network

model is updated with different frequency than the target policy interpretable model and also with different batch size(line 20). This is done for the same reason as the delayed policy updates because we do not do soft-update like Original TD3 algorithm(line 22), and instead we do something similar as copying the weights of the neural policy model to the interpretable one, thus, when target policy model gets updated less frequently the learning is more stable. We are also using different batch size to train the interpretable models because interpretable models need more data to be good mimics of the primary neural network policy model, i.e. to minimize validation loss. Secondly, in line 22, unlike the original TD3 algorithm where the target policy network is changed as $\tau\phi + (1-\tau)\phi'$, the target policy model is fitted with the actions extracted from current primary policy network. This is done because we want to extract the logic behind the primary policy network. Last but not least, in line 20, the sampling of the N' transitions in order to update the interpretable policy model model must be made more specific. This step is done with either of the following ways:

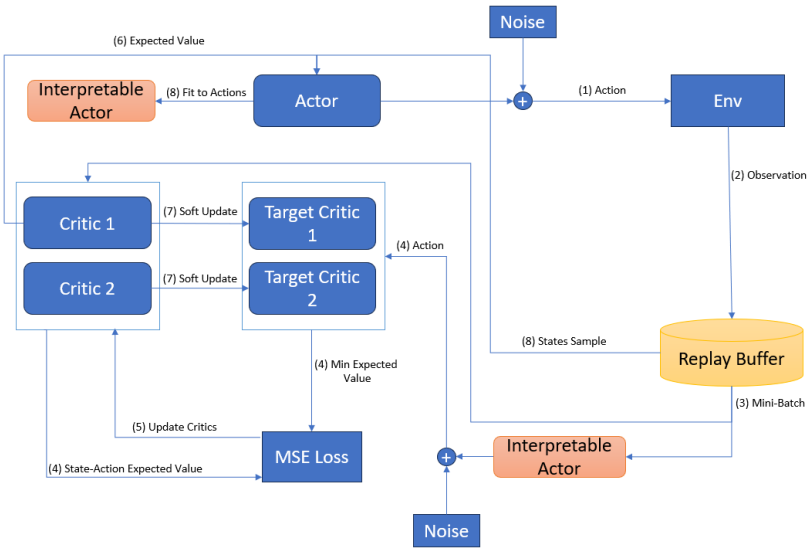
- **All** which means that the **entirety** of states from the replay buffer are taken.
- **Recent** which means that the N' most recent states in terms of insertion are taken from the replay buffer.
- **Recent Window** which means that a random sample of N' states is taken from the M' most recent ones in terms of insertion in the replay buffer, where $N' < M'$.

Algorithm 4 Interpretable TD3

Initialize critic networks Q_{θ_1}, Q_{θ_2} and actor network π_ϕ with random parameters θ_1, θ_2, ϕ
Initialize target networks for the critics $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$
Initialize randomly target actor interpretable model with parameters ϕ'.
Initialize replay buffer $\mathcal{B}$
for $t = 1$ **to** T **do**
    Select action with exploration noise $\alpha \sim \pi_\phi(s) + \epsilon$, $\epsilon \sim N(0, \sigma)$ and observe reward r and new state s'
    Store transition tuple (s, α, r, s') in $\mathcal{B}$
    Sample mini-batch of N transitions (s, α, r, s') from $\mathcal{B}$
    $\widetilde{\alpha} \leftarrow \pi_{\phi'}(s') + \epsilon$, $\epsilon \sim \text{clip}(N(0, \sigma), -c, c)$
    $y \leftarrow r + \gamma min_{i=1,2} Q_{\theta_i'}(s', \widetilde{\alpha})$
    Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, \alpha))^2$
    if t mod d == 0 **then**
        Update ϕ by the deterministic policy gradient:
        $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_\alpha Q_{\theta_1}(s, \alpha)|_{\alpha=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$
        Update target networks:
        $\theta_i' \leftarrow \tau\theta_i + (1 - \tau)\theta_i'$
    end if
    if t mod d' == 0 **then**
        Sample mini-batch of N' transitions (s, α, r, s') from $\mathcal{B}$ using the selected sampling strategy for interpretable actor model.
        Initialize interpretable model's parameters ϕ'
        Update ϕ' fitting interpretable model using states s as instances and $\pi_\phi(s)$ as labels.
    end if
end for



(a) Original TD3



(b) Interpretable TD3

Figure 3: Illustration of the inner workings of Twin-Delayed Policy Gradient algorithm in both original and interpretable versions

An extra sampling method, is the **Experience Gain** one where the basic idea is to

sample N' transitions from the experience of the last K actor's instances. However, in order to do that, the primary actor network needs to run for N' steps, without updating its weights, in order to gain the required experience. To make this approach more comprehensive we describe it using the state diagram in 4 and break down how each state is operating.

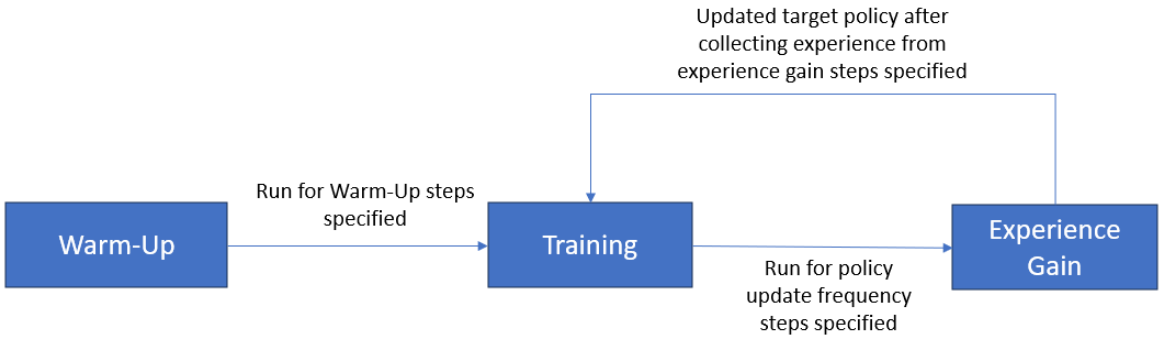


Figure 4: State diagram of an Interpretable TD3 Agent using Experience Gain sampling method

As we can see from the diagram 4, the agent starts from the warm-up state, just like the original TD3 algorithm in order to prevent any local optima due to the initial parameters of the policy, and then the agent alternates between the Training State and the Experience Gain State. Before describing each state in detail, we will first show the agent's initialization:

Algorithm 5 Agent's Initialization

1: Initialize critic networks Q_{θ_1}, Q_{θ_2} and actor network π_ϕ with random parameters θ_1, θ_2, ϕ
2: Initialize target networks for the critics $\theta_1' \leftarrow \theta_1$, $\theta_2' \leftarrow \theta_2$
3: Initialize randomly interpretable target actor model with parameters ϕ'.
4: Initialize Replay buffer $\mathcal{B}$
5: Initialize Experience Gain Replay Buffer $\mathcal{B}'$

As we can see in the above algorithm, we initialize the networks and the interpretable target actor model. Also, we create two replay buffers, where one is the same as the one in the original TD3 algorithm and the other is used to store experience from the experience gain state and eventually sample from that in order to train the target actor model. Then, the pseudocode for each of those states can be seen in **Algorithm 6**, **Algorithm 7** and **Algorithm 8**.

Comparing **Algorithm 6** and **Algorithm 7**, the only differences between Train and Warm-up state is that in the former, the action is selected in a greedy way, while in the latter, the action that the agent will take is selected randomly and also after a number of steps the agent switches to Train state without returning again to warm-up state. The Experience Gain state that is shown on **Algorithm 8** involves the current actor snapshot, running in the environment in a greedy manner with some exploration noise(line 3), without updating its weights, and storing the transitions into the second replay buffer B', as shown in line 4. When we have stored $ExperienceGainSteps$ transitions, if the Experience Gain Replay Buffer is full, then the target actor model is trained as shown from lines 9 to 11. Replay Buffer B' has size equal to $numActors \cdot ExperienceGainSteps$, thus, this means that B' is full when at least $numActors$ actor's instances have passed through Experience Gain state.

Last but not least, one thing to take care for, is that the interpretable models that are currently used, do not have any mechanism for multi-output regression. In order to handle it, the agent can keep one instance for each action coordinate and use only the given state as input to each instance. Another way, in order to introduce dependence among action's coordinates, can be that for each action coordinate, the agent makes its prediction based on the state and the prediction that he did on previous action coordinates, if applicable.

Algorithm 6 Warm-up State

Total warm-up steps, $WarmupSteps > 0$

$CurrStep \leftarrow 0$

Select random action action $\alpha \sim \epsilon$, $\epsilon \sim N(0, \sigma)$ and observe reward r and new state s'

Store transition tuple (s, α, r, s') in $\mathcal{B}$

$CurrStep \leftarrow CurrStep + 1$

Sample mini-batch of N transitions (s, α, r, s') from $\mathcal{B}$

$\widetilde{\alpha} \leftarrow \pi_{\phi'}(s') + \epsilon$, $\epsilon \sim \text{clip}(N(0, \sigma), -c, c)$

$y \leftarrow r + \gamma min_{i=1,2} Q_{\theta_i'}(s', \widetilde{\alpha})$

Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, \alpha))^2$

if t mod d == 0 **then**

    Update ϕ by the deterministic policy gradient:

    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_\alpha Q_{\theta_1}(s, \alpha)|_{\alpha=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$

    Update target networks:

    $\theta_i' \leftarrow \tau\theta_i + (1 - \tau)\theta_i'$

end if

if $CurrStep == WarmUpSteps$ **then**

    Move to **Train State**

end if

if t mod d' == 0 **then**

    Move to **Experience Gain State**

end if

Algorithm 7 Train State

Select action with exploration noise $\alpha \sim \pi_\phi(s) + \epsilon$, $\epsilon \sim N(0, \sigma)$ and observe reward r and new state s'

Store transition tuple (s, α, r, s') in $\mathcal{B}$

Sample mini-batch of N transitions (s, α, r, s') from $\mathcal{B}$

$\widetilde{\alpha} \leftarrow \pi_{\phi'}(s') + \epsilon$, $\epsilon \sim \text{clip}(N(0, \sigma), -c, c)$

$y \leftarrow r + \gamma min_{i=1,2} Q_{\theta_i'}(s', \widetilde{\alpha})$

Update critics $\theta_i \leftarrow \text{argmin}_{\theta_i} N^{-1} \sum (y - Q_{\theta_i}(s, \alpha))^2$

if t mod d **then**

    Update ϕ by the deterministic policy gradient:

    $\nabla_\phi J(\phi) = N^{-1} \sum \nabla_\alpha Q_{\theta_1}(s, \alpha)|_{\alpha=\pi_\phi(s)} \nabla_\phi \pi_\phi(s)$

    Update target networks:

    $\theta_i' \leftarrow \tau\theta_i + (1 - \tau)\theta_i'$

end if

if t mod d' == 0 **then**

    Move to **Experience Gain State**

end if

Algorithm 8 Experience Gain State

Total experience gain steps, $ExperienceGainSteps > 0$
$CurrStep \leftarrow 0$
Select action with exploration noise $\alpha \sim \pi_\phi(s) + \epsilon$, $\epsilon \sim N(0, \sigma)$ and observe reward r and new state s'
Store transition tuple (s, α, r, s') in $\mathcal{B}'$
$CurrStep \leftarrow CurrStep + 1$

if $CurrStep == ExperienceGainSteps$ **then**
    if Experience Gain Buffer B' is full **then**
        Sample random mini-batch of N' transitions (s, α, r, s') from $\mathcal{B}'$.
        Initialize target actor model's parameters ϕ'
        Update ϕ' using states s as instances and $\pi_\phi(s)$ as labels.
    end if
    Move to **Train State**
end if

5 Experimental Evaluation

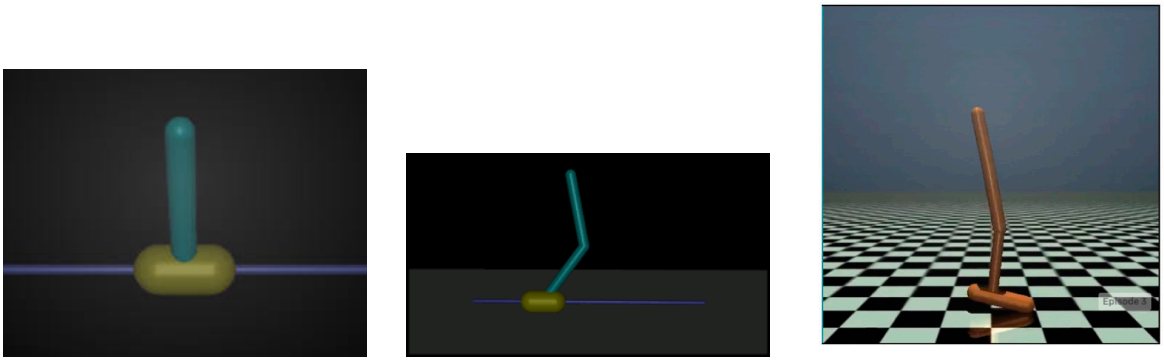
5.1 Evaluation tasks

In the previous section we presented an interpretable version of Twin Delayed Policy Gradient algorithm that helps us build interpretable agents that can perform on environments with continuous actions. Our approach's performance will be evaluated on the suite of MuJoCo continuous control tasks[34], interfaced through Gymnasium[35] which is a maintaining clone of OpenAI Gym[36], without any modifications to the environment or the reward in order for our experiments to be easily reproducible. The environments that the experiments will be conducted are the following:

- **InvertedPendulum** and **InvertedDoublePendulum**[37] where the agent is actually a cart that can move left or right applying a specific amount of force with ultimate goal to balance the one or two piece pole which is attached to the cart, respectively. Those environments are the simpler ones because the agent has only one action dimension, thus, it does not need to coordinate more than one joints. Also their observation space is small because their dimensionality is equal to 4.
- **Hopper**[38] is an environment where the agent has 3 joints that he can control, the thigh, the leg and the foot. Therefore, its 3-dimensional action space represents the torque that can be applied to its three joints. On the other hand, its observation space consists of 11 dimensions. The goal of the agent is to keep jumping forward as much as it can.

Note, that in the last environment the agent is required to jump forward with controlled and not abrupt movements in order to achieve maximum rewards. Consequently, we can see that the evaluation tasks cover various cases of state-action complexity and also showcase increased difficulty because not only they have high dimensional observation space, but the presented models should also map those observations to high dimen-

sional action spaces. This mapping is also challenging because each action dimension needs an instance of the interpretable model used. An illustration of the environments can be seen in Figure 5. More details about the environments can be found in Gymnasium's documentation.



(a) InvertedPendulum-v4

(b) InvertedDoublePendulum-v4

(c) Hopper-v4

Figure 5: Illustration of Gymnasium environments that will be used for our method's evaluation.

5.2 Implementation details and experimental setup

In order to compare our approach in a fair manner with the original TD3 algorithm, the hyperparameters of the interpretable method are made equal to the hyperparameters of the original method, and will be equal to those stated in [11], thus, the presented approach will differ only in the new hyperparameters. Specifically, both the actor and critic networks are 2-layer fully connected feed-forward neural networks where the first layer contains 400 neurons and the second one 300 neurons that are using rectified linear units[39] as their activation function. Actor network's output layer will use $tanh$ multiplied by maximum possible action in order to produce decisions within the agent's

bounds. Both network parameters are updated using Adam [40] with a learning rate of 10^{-3}. As you can see in **Algorithm 3** the networks are trained from a mini batch of collected experience, saved in replay buffer. This mini batch has size $N = 100$ and the replay buffer $\mathcal{B}$ has size equal to 1 million transitions.

Target policy smoothing is implemented by adding $\epsilon \sim \mathcal{N}(0, 0.2)$ to the actions chosen by the target actor network, clipped to $(-0.5, 0.5)$. Delayed policy updates consist of updating the actor and target critic network every d iterations, with $d = 2$. While a larger d would result in a larger benefit with respect to accumulating errors, for fair comparison, the critics are only trained once per time step, and training the actor for too few iterations would cripple learning. Both policy and critic target networks are updated with τ equal to 0.005, which in practice means that their weights are 99.5 percent the same with a 0.5 percent change in the direction of the weights of the corresponding primary model. Furthermore, the number of warm-up steps, where the agent is choosing actions randomly, is equal to 1000 for all environments.

In terms of interpretable TD3's hyperparameters, as it was mentioned before, they are the same as the original TD3 algorithm, where applicable, however, there are some which are introduced in the presented approach. Firstly, the update frequency of the target actor interpretable model d' is equal to 200, because if it gets set to a smaller value the learning becomes slower without giving better results. Secondly, because the interpretable target actor interpretable model needs more data in order to be able to replicate the primary actor network, the number of sampled instances N' is equal to 10000. Furthermore, the sampling method of N' transitions that seem to work best was the Experience Gain one where those transitions are getting sampled from the most recent policy network (actor) instance. This approach in comparison with the others in terms of the learning curve of the primary policy model did not show any major difference. However, when testing the performance of the interpretable models, the one trained with Experience Gain performed much better, which means that it fits better the policy

of the neural network actor. This happens because the interpretable model fits from instances extracted from a single distribution, i.e. in our case, a single actor policy network. Finally, as stated in the previous section, the interpretable models used are not designed for multi-output regression and two ways were provided. The approach that is assuming complete independence among action coordinates performed the same level as the one that is introducing a dependence of an action coordinate with its previous one. Also, the former approach can be parallelized, thus, the predictions can be done faster. Thus, in the experiments run, we have one interpretable model's instance per action coordinate that takes as input only the given states.

Last but not least, we need to provide information about the hyperparameters of the interpretable models that get used in the method. As mentioned in previous section, the interpretable model can be a number of Stochastic Gradient Trees or Gradient Boosting Regression Trees. For the former, the SGTs are trained for 10 epochs with learning rate of 0.5, the numerical attributes are split into 8 equal width bins and in order to perform a split we should see at least 16 samples. Also, the bounds for each attribute x are equal to the 95% confidence interval derived from the given batch of the N' transitions. For the Gradient Boosting Regression Trees (GBR), we did a grid-search in order to find the best GBR hyperparameters for each of the experimental tasks, given that GBR is faster than SGTs, thus, it was feasible to test it in all cases. The hyperparameters that we tuned in GBR are the following:

1. The number of estimators, i.e. trees, that the model will have.
2. The maximum depth of those estimators.
3. Method's Learning Rate.
4. The percentage of the number of the training samples that each tree will fit on.
5. The minimum number of training samples that should fall into a leaf node.

The custom hyperparameters can be seen in a more organized manner in Tables 1, 2, and 3.

Table 1: Interpretable TD3 and TD3(where applicable) Hyperparameters

Hyperparameter	**Value**
Neural Network Actor's Learning Rate	10^{-3}
Critic's Learning Rate	10^{-3}
τ	$5x10^{-3}$
Neural Network's Batch Size	**100**
Discount Factor	0.99
Actor's Update frequency in terms of Agent's Steps	2
Warm-up Steps	10^3 or 10^4
Replay Buffer Size	10^6
Number of neurons of the first fully connected layer of the networks	400
Number of neurons of the second fully connected layer of the networks	300
Sampling strategy for training (interpretable) target actor	*Experience Gain*
Experience Gain Actor Samples	10^5
Experience Gain Actor Instances	1
Target Actor's Update frequency in terms of Agent's Steps	200
Noise to action	10^{-1}

Table 2: Stochastic Gradient Trees Hyperparameters per environment

Hyperparameter	**Value**
SGT Train Epochs	10
SGT Bins for Numerical Values	8
SGT Learning Rate	0.5
SGT Batch Size(# samples to see in order to split)	16

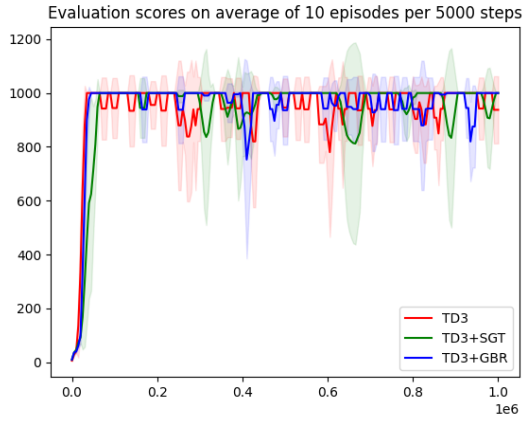
Table 3: Gradient Boosting Regression Trees Hyperparameters per environment

	InvertedPendulum-v4	**InvertedDoublePendulum-v4**	**Hopper-v4**
# Estimators	500	500	100
Estimator's Max Depth	5	5	20
Learning Rate	0.5	0.5	0.1
Estimator's subsample	1.0	1.0	1.0
# Minimum Samples	100	100	250

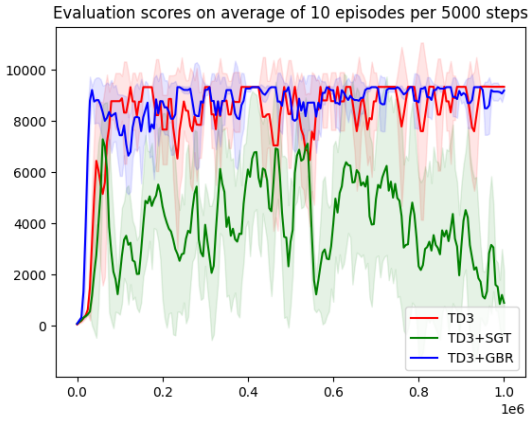
The evaluation of the approaches and their comparison with the Original TD3 algorithm, is done in two phases where in the former their learning curve and fidelity of the interpretations is evaluated, and in the latter the standalone performance of the resulted interpretable model is evaluated. In the first phase, the agent runs for 1 million training steps and every 5000 training steps the agent stops training and runs for 10 episodes where the agent chooses actions without any exploration noise in order to determine the performance given its current parameters. The average evaluation results from 5 experiment runs along with their standard deviation are shown in Figure 6. Note that, in all environments the agent runs for at most 1000 steps per episode.

The implementation of the presented approaches was done in Python, using PyTorch with GPU optimization for the neural networks, SciKit-Learn[41] for the Gradient Boosting Regressor and pySGT was used for the Stochastic Gradient Trees. The implementation of the presented approach can be found in this GitHub repository.

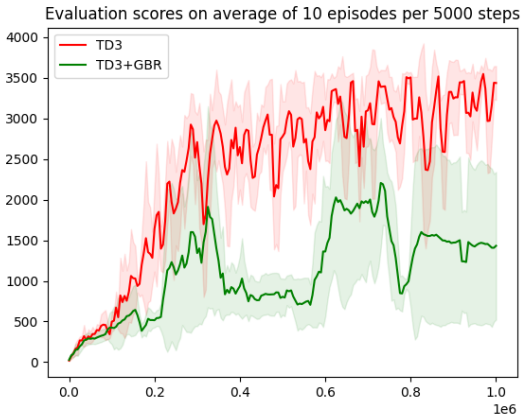
5.3 Experimental results



(a) InvertedPendulum-v4



(b) InvertedDoublePendulum-v4

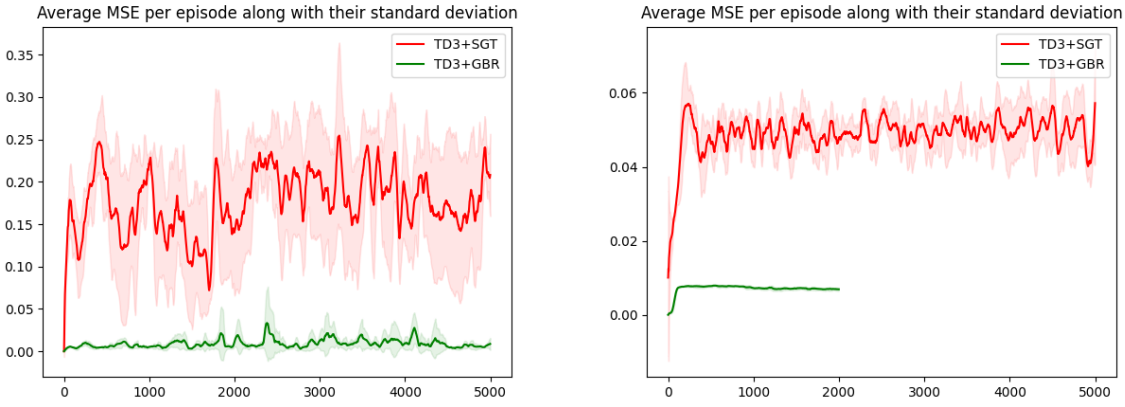


(c) Hopper-v4

Figure 6: Learning curves of various models in the respective environments.

Figure 6 shows per environment and per method, the average reward and its standard deviation that the neural network policy model yields across 5 runs, when ran in evaluation mode i.e. without adding any noise to its chosen actions, for 10 episodes. The applied methods switch to evaluation mode per 5000 training steps and in Figure's 6 results, their agents were trained for 1 million steps. From this Figure, the following can

be induced. Initially, it can be seen in Figure 6a that in the easier problem which is the InvertedPendulum, TD3 using SGTs performs in the same level as the original TD3 algorithm. However, when the problem becomes a little bit harder like the InvertedDoublePendulum, as can be seen in Figure 6b, TD3 with SGT has clearly worse performance than the original TD3 and TD3 using GBRs agents, where those two still have similar performance. Investigating why this happens, two things were investigated. The first one was how good is the fit that the interpretable model does on the given data. This gets investigated by determining the training Mean Squared Error(MSE) per episode. MSE per episode is the average training MSE along the interpretable model updates that happened in a single episode. The results can be seen in Figure 7. Note that, it is normal sometimes for the resulting curves to have variable sizes because a better performing agent covers 1 million timesteps in less episodes than the one who performs worse, thus, each episode runs for less steps.



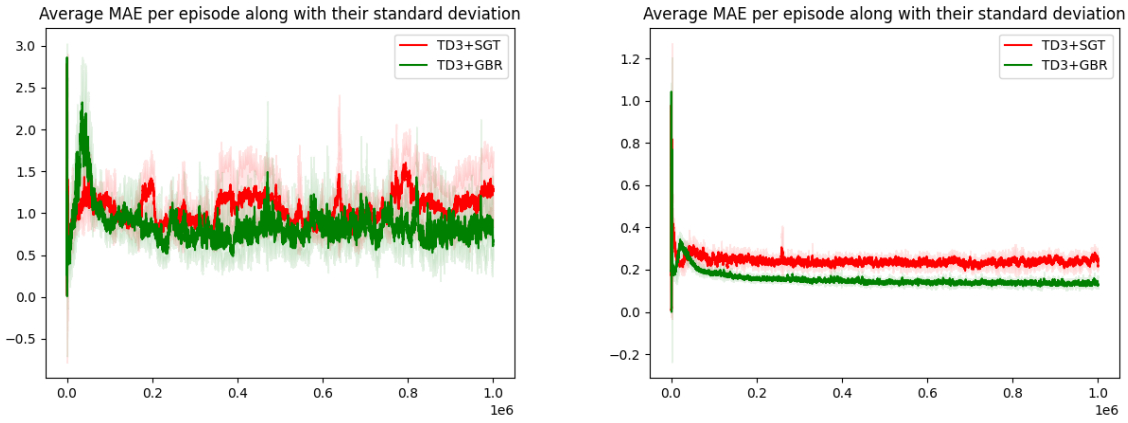
(a) InvertedPendulum-v4 (b) InvertedDoublePendulum-v4

Figure 7: Training MSE of interpretable models in the respective environments.

From Figure 7, it is clear that training MSE is higher for the SGT case than the GBR case in both environments. However, the paradox in this is that SGT's MSE in InvertedPen-

dulum is higher than MSE in the InvertedDoublePendulum case, but InvertedPendulum SGT agent performs in the same level as the GBR variation and the original TD3 algorithm. This might happen because InvertedPendulum is an environment which is easier to solve, i.e. fit to the optimal policy, than the InvertedDoublePendulum one, even if it has greater MSE, or it might be less sensitive to mistakes, thus, even if in some action makes a bad decision, the error can be fixed at a next step. Furthermore, comparing the MSE per episode between Interpretable TD3 using SGT with the one using GBR per environment, in both Figures 6a and 6b we can see that GBRs fit much better than SGTs, having training loss close to zero.

The second thing that we investigated is the fidelity of the interpretable models. Fidelity is defined as the difference between the decision made by the interpretable model with the decision that the target neural network policy model would have made if it got updated as the original TD3 algorithm. Fidelity is measured by the Mean Absolute Error per each training step between the decisions of those two models. The fidelity along with its standard deviation per step across 5 runs of 1 million steps each, can be seen in Figure 8. We can see that the fidelity of the interpretable policy model is higher in the InvertedDoublePendulum case(Figure 8b) than in the InvertedPendulum one(Figure 8a). Both interpretable models in InvertedPendulum, as seen in Figure 8a present the same fidelity, converging nearly to 1, which is half of the maximum possible MAE. However, in both versions of interpretable TD3, the neural network policy model yields the maximum possible episodic reward when it gets to evaluation phase, as seen in Figure 6a, thus, the training reaches the optimum.

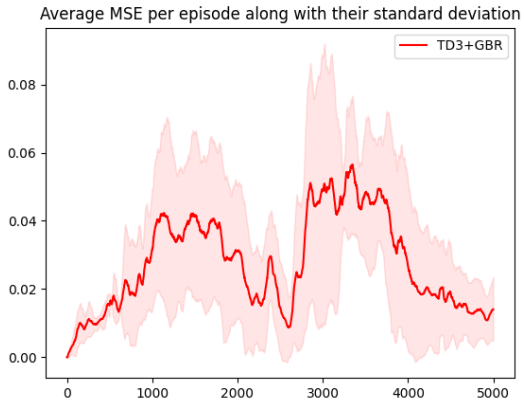


(a) InvertedPendulum-v4 (b) InvertedDoublePendulum-v4

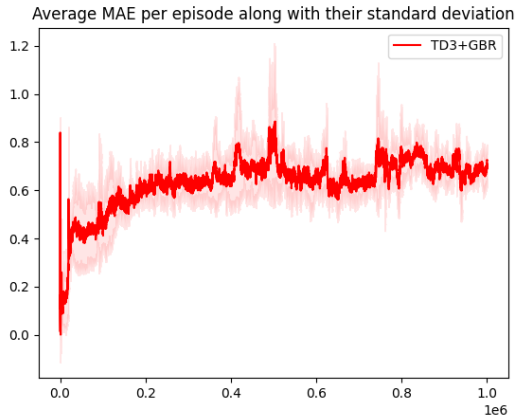
Figure 8: MAE of interpretable models in the respective environments.

However, because TD3 with SGTs performs worse even in simple problems, we do not evaluate it in the rest of the environments.

As Figure 6c shows, for the Hopper environment, the interpretable TD3 using GBR performs worse that original TD3 algorithm. In order to explain why this happens, we will check initially, the average MSE per episode and interpretable agent's fidelity. From Figure 9a, we can see that MSE is low throughout training, as it was for InvertedDoublePendulum. This means that GBR does not have a problem to fit to neural network's policy. From Figure 9b we can see that the fidelity decreases as the training progresses and it is virtually at the same level with InvertedPendulum as seen in Figure 8a. However, the main particularity that Hopper environment introduces, is that this environment's action space is multidimensional, therefore, it is more difficult for the interpretable models to be trained effectively well, assuming that there is a separate model per action dimension.



(a) Hopper's MSE per episode



(b) Hopper's MAE per step

Figure 9: Hopper's average MSE and MAE per episode.

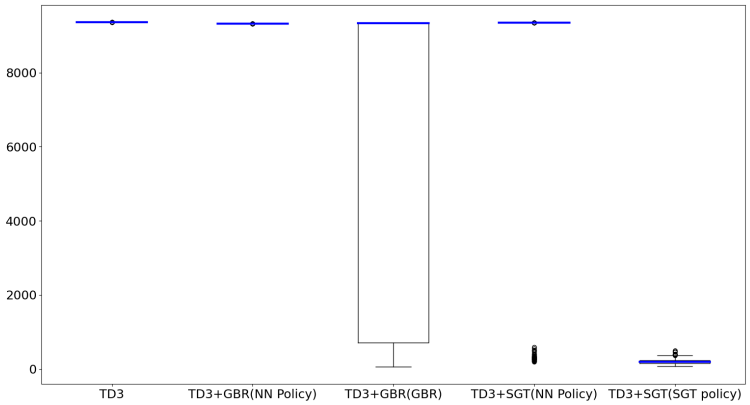
In the second phase, we will run the best, according to the evaluation results, agent's policy neural network and interpretable target policy model that was extracted from the previous phase, for 500 episodes without updating any of their parameters, i.e. they run in evaluation mode. This is useful firstly in order to see what the neural network policy model learned but also how the interpretable model can perform by itself on the environment. If the interpretable model performs closely to the primary policy network, then we can assume that its interpretations are reliable because they represent a policy of similar quality with the one of the neural network policy model, thus, it has a good basis to provide explanations regarding agent's policy. The results can be seen in Table 4 where we report the average episodic reward of each policy model, along with its standard deviation, if run standalone in each environment for 500 episodes. In this table, TD3 represents the Original TD3 method, where NN Policy represents the primary neural network policy model. Furthermore, TD3+GBR and TD3+SGT denote the Interpretable TD3 method using GBR and SGT as their target interpretable policy model, respectively. In those methods both the primary neural network policy model(NN Pol-

icy) and the target interpretable model(GBR or SGT) are evaluated in terms of their standalone performance, i.e the reward.

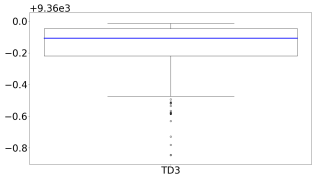
Table 4: Best agent's primary policy network and target policy model average reward with standard deviation, extracted from 500 evaluation episodes.

	TD3	TD3+GBR		TD3+SGT	
	NN Policy	NN Policy	GBR	NN Policy	SGT
InvertedPendulum-v4	1000 ± 0.0	1000 ± 0.0	1000 ± 0.0	1000 ± 0.0	51.83 ± 12.07
InvertedDoublePendulum-v4	9359.85 ± 0.14	9324.27 ± 0.15	6145.95 ± 4046.92	8790.02 ± 2174.30	201.70 ± 64.96
Hopper-v4	3531.76 ± 4.72	3343.69 ± 517.79	2409.32 ± 916.76	N/A	N/A

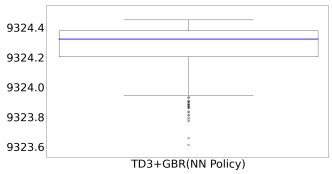
Initially, from Table 4, we can see that TD3+SGT method's interpretable SGT policy model does not generalize at all in every environment that it was tested because the average episodic reward is much lower than the ones that the other policy models yield. However, we can see that TD3+SGT method's NN Policy, has perfect performance in InvertedPendulum environment while in InvertedDoublePendulum it has non-optimal performance but the average reward is not much lower than the optimal one. However, it reports an extremely high standard deviation. This is expected because, as seen in Figure 6b, the NN Policy model learnt from TD3 using GBRs, does not have the same performance as the neural network policy model learnt from the original TD3. However, as shown in Table 4 the average episodic reward is lower with comparison to TD3's NN Policy's average reward and their standard deviation high. Checking Figure 10a, this happens mostly due to few under-performing episodes that yield reward around 500, while from Figure 10d it can be seen that in the the episodes that their total reward are not outliers, which comprise the majority of them, TD3+SGT's NN Policy yields very high rewards, i.e., around 9349 units.



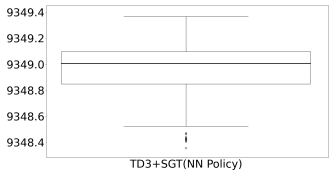
(a) All models boxplots except TD3+SGT(SGT)



(b) TD3 zoomed-in boxplot



(c) TD3+GBR(NN Policy) zoomed-in boxplot



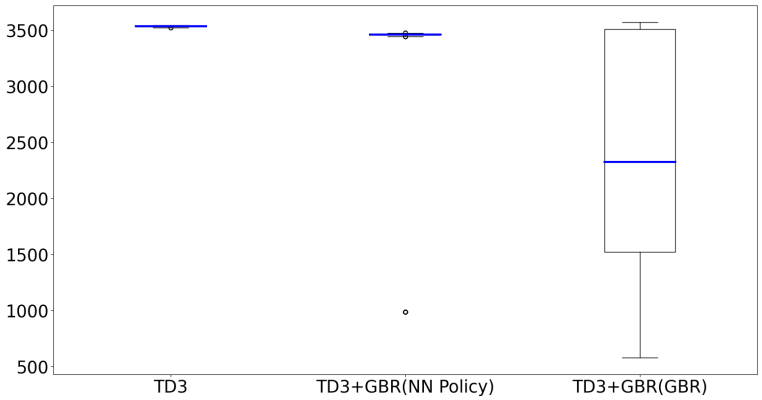
(d) TD3+SGT(NN Policy) zoomed-in boxplot

Figure 10: Boxplots of evaluation scores in InvertedDoublePendulum-v4 per policy model. Note that each subfigure has different scaling.

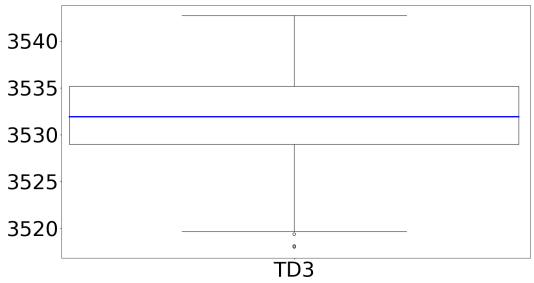
Regarding results for the TD3+GBR, in comparison to the TD3, reported in Table 4, we can deduce the following. Initially, with regard to TD3+GBR method's NN Policy performance, we can see that in all environments it has almost similar performance with the TD3 NN Policy, except for Hopper where it reports a larger deviation. As seen from the corresponding boxplot in Figure 11a, this happens due to a small number of under-performing episodes with reward close to 1000 units, while in the rest of the episodes, as seen from Figure 11c, the policy models reports very high episodic reward, around 3460 units, which is close to the average reward of TD3. Last but not least, from TD3+GBR method's GBR model's results, we can see that although in Inverted-

Pendulum the policy has optimal performance, in InvertedDoublePendulum and Hopper, GBR does not seem to generalize well. This might happen due to a small number of under-performing episodes or this might be due to poor generalization which would result in many under-performing episodes. Regarding TD3+GBR GBR policy model's performance in InvertedDoublePendulum environment, from Figure 10a we can see that even though there is a high median value, lower rewards are not considered outliers. Therefore, we can extract the conclusion that in half of the episodes the GBR policy model performs optimally, while in the others it yields suboptimal rewards. Regarding TD3+GBR GBR policy model's performance in Hopper environment, from Figure 11a we can see that its episodic rewards span through the whole range, with no outliers. Consequently, we can see that even though the GBR policy models have good fidelity with respect to neural network policies, they do not have the same performance.

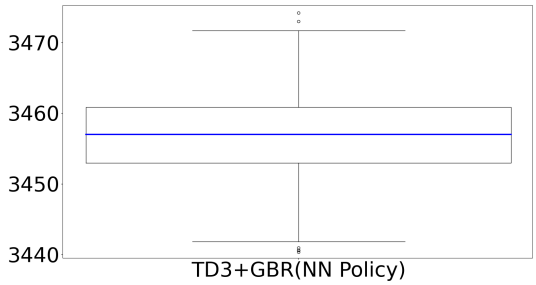
Summing up, from the presented results, we can see that the interpretable models have high fidelity with their respective neural network policy model, but have poor generalization properties which was shown when having them run standalone on the environment. Therefore, the resulting interpretable policy models can provide accurate interpretations regarding the inner workings of the neural network policy model, but, they are not in a level that they can run standalone in the given environment.



(a) All models boxplots



(b) TD3 zoomed-in boxplot



(c) TD3+GBR(NN Policy) zoomed-in boxplot

Figure 11: Boxplots of evaluation scores in Hopper-v4 per policy model. Note that each subfigure has different scaling.

6 Conclusions & Further Work

In this thesis, we presented a method that creates interpretable agents that perform on environments with continuous action spaces and investigated their performance on those environments. Specifically, our method modifies the Twin-Delayed Policy Gradient method by replacing the target neural network model with an interpretable model that is used as a mimic of the primary neural network policy model. Finally, we evaluated our method with each interpretable model in two parts. In the first part, we investigated the learning curve of the primary neural network policy model and whether it converges to a good cumulative reward. We show how well the interpretable policy model fits the given neural network policy and we also show its fidelity in terms of how similar were the interpretable model's decisions in comparison to the ones that the target neural network policy model of the original TD3 method makes. In the second part, we evaluated the performance of the best models that were learnt from the training of the neural network policy model and of the interpretable model, in order to determine their generalization abilities. Overall, regarding the first part we show that TD3 using GBRs can reach the same performance, i.e. yield rewards of the same level, as the original TD3 algorithm. TD3 using SGTs has weaker performance in harder environments. Regarding the second part, the neural network policy models using GBR have similar performance than the one of the original TD3 algorithm, but the GBRs perform worse due to learning capacity. Still in some cases they can have similar performance.

There are several directions that we can take for future work. Initially, this approach does not yield good performance in some environments, like Hopper, even with TD3 using GBRs, thus, we should investigate further why it does not work. An initial thought is that this happens because each action coordinate uses a separate model, thus, we assume independence among the coordinates, resulting in actions' dimensions miscoordination. Therefore, we need to determine how we can modify our approach in order

to perform well, considering correlations among actions' dimensions. Firstly, we can investigate the dependence among action coordinates values and try again to introduce dependence among action coordinates in a "chain" manner, but with a specific order. Secondly, we can work on using interpretable models that are designed for multi-output regression, like XGBoost[31]. Thirdly, because Machine Learning models, unlike Deep Neural Networks, cannot have the same predictive performance with raw data, it is worth trying to apply feature engineering in the sampled states before feeding them to the interpretable models. Furthermore, we should investigate whether we can yield better interpretations using an interpretable model that can perform standalone in the environments yielding rewards close to those of the neural network models. Moreover, it would be useful to also make TD3 interpretable by using distillation instead of mimicking in order to exploit the inner working components of Deep Reinforcement Learning agents and get directly the interpretable policy model. Last but not least, we should try and make interpretable versions of other actor-critic methods, like SAC[12].

References

1. Sutton, R. S. & Barto, A. G. *Reinforcement Learning: An Introduction* Second. http://incompleteideas.net/book/the-book-2nd.html (The MIT Press, 2018).
2. Bellman, R. A Markovian Decision Process. *Indiana Univ. Math. J.* **6,** 679–684. ISSN: 0022-2518 (4 1957).
3. Bertsekas, D. Multiagent Value Iteration Algorithms in Dynamic Programming and Reinforcement Learning. arXiv: 2005.01627 [math.OC] (2020).
4. Barto, A. & Duff, M. Monte Carlo Matrix Inversion and Reinforcement Learning. *Advances in Neural Information Processing Systems* **6** (Feb. 1995).
5. Bradtke, S. Incremental Dynamic Programming for On-Line Adaptive Optimal Control (Dec. 1994).
6. Bradtke, S. & Barto, A. Linear Least-Squares Algorithms for Temporal Difference Learning. *Machine Learning* **22,** 33–57 (Mar. 1996).
7. Mnih, V. *et al.* Playing Atari with Deep Reinforcement Learning. arXiv: 1312.5602 [cs.LG] (2013).
8. Agarwal, A., Kakade, S. M., Lee, J. D. & Mahajan, G. On the Theory of Policy Gradient Methods: Optimality, Approximation, and Distribution Shift. arXiv: 1908.00261 [cs.LG] (2020).
9. Grondman, I., Busoniu, L., Lopes, G. A. D. & Babuska, R. A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* **42,** 1291–1307 (2012).
10. Lillicrap, T. P. *et al.* Continuous control with deep reinforcement learning. arXiv: 1509.02971 [cs.LG] (2019).

11. Fujimoto, S., van Hoof, H. & Meger, D. Addressing Function Approximation Error in Actor-Critic Methods. arXiv: 1802.09477 [cs.AI] (2018).

12. Haarnoja, T., Zhou, A., Abbeel, P. & Levine, S. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. arXiv: 1801.01290 [cs.LG] (2018).

13. Dong, S., Wang, P. & Abbas, K. A survey on deep learning and its applications. *Computer Science Review* **40,** 100379. ISSN: 1574-0137. https://www.sciencedirect.com/science/article/pii/S1574013721000198 (2021).

14. Vouros, G. A. Explainable Deep Reinforcement Learning: State of the Art and Challenges. *ACM Computing Surveys* **55,** 1–39. https://doi.org/10.1145%2F3527448 (Dec. 2022).

15. Kontogiannis, A. & Vouros, G. in, 160–179 (Sept. 2023). ISBN: 978-3-031-40877-9.

16. Pyeatt, L. & Howe, A. Decision Tree Function Approximation in Reinforcement Learning (July 2001).

17. Watkins, C. J. C. H. & Dayan, P. Q-learning. *Machine Learning* **8,** 279–292. ISSN: 1573-0565. https://doi.org/10.1007/BF00992698 (May 1992).

18. Roth, A. M., Topin, N., Jamshidi, P. & Veloso, M. Conservative q-improvement: Reinforcement learning for an interpretable decision-tree policy. *arXiv preprint arXiv:1907.01180* (2019).

19. Gouk, H., Pfahringer, B. & Frank, E. Stochastic Gradient Trees. arXiv: 1901.07777 [stat.ML] (2019).

20. Verma, A., Murali, V., Singh, R., Kohli, P. & Chaudhuri, S. *Programmatically Interpretable Reinforcement Learning* 2019. arXiv: 1804.02477 [cs.LG].

21. Åström, Karl Johan. Optimal Control of Markov Processes with Incomplete State Information I. eng. *Journal of Mathematical Analysis and Applications* **10,** 174–205. ISSN: 0022-247X. %7Bhttps://lup.lub.lu.se/search/files/5323668/8867085.pdf%7D (1965).

22. Coppens, Y., Efthymiadis, K., Lenaerts, T. & Nowe, A. *Distilling Deep Reinforcement Learning Policies in Soft Decision Trees* English. in *Proceedings of the IJCAI 2019 Workshop on Explainable Artificial Intelligence* (eds Miller, T., Weber, R. & Magazzeni, D.) IJCAI 2019 Workshop on Explainable Artificial Intelligence, XAI19 ; Conference date: 11-08-2019 (Aug. 2019), 1–6. https://sites.google.com/view/xai2019/home.

23. Frosst, N. & Hinton, G. *Distilling a Neural Network Into a Soft Decision Tree* 2017. arXiv: 1711.09784 [cs.LG].

24. Togelius, J., Karakovskiy, S. & Baumgarten, R. *The 2009 Mario AI Competition* in *IEEE Congress on Evolutionary Computation* (2010), 1–8.

25. Schulman, J., Wolski, F., Dhariwal, P., Radford, A. & Klimov, O. *Proximal Policy Optimization Algorithms* 2017. arXiv: 1707.06347 [cs.LG].

26. Van Hasselt, H., Guez, A. & Silver, D. Deep Reinforcement Learning with Double Q-learning. arXiv: 1509.06461 [cs.LG] (2015).

27. Friedman, J. Stochastic Gradient Boosting. *Computational Statistics & Data Analysis* **38,** 367–378 (Feb. 2002).

28. Kiefer, J. & Wolfowitz, J. Stochastic Estimation of the Maximum of a Regression Function. *Annals of Mathematical Statistics* **23,** 462–466. https://api.semanticscholar.org/CorpusID:122078986 (1952).

29. Domingos, P. & Hulten, G. *Mining High-Speed Data Streams* in *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Association for Computing Machinery, Boston, Massachusetts,

USA, 2000), 71–80. ISBN: 1581132336. https://doi.org/10.1145/347090.347107.

30. Loh, W.-Y. Classification and Regression Trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* **1,** 14–23 (Jan. 2011).

31. Chen, T. & Guestrin, C. *XGBoost* in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (ACM, Aug. 2016). https://doi.org/10.1145%2F2939672.2939785.

32. Freund, Y. & Schapire, R. E. A desicion-theoretic generalization of on-line learning and an application to boosting (ed Vitányi, P.) 23–37 (1995).

33. Van Hasselt, H. Double Q-learning. 2613–2621 (Jan. 2010).

34. Todorov, E., Erez, T. & Tassa, Y. *MuJoCo: A physics engine for model-based control* in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2012), 5026–5033.

35. Towers, M. *et al. Gymnasium* Mar. 2023. https://zenodo.org/record/8127025 (2023).

36. Brockman, G. *et al.* OpenAI Gym. arXiv: 1606.01540 [cs.LG] (2016).

37. Barto, A. G., Sutton, R. S. & Anderson, C. W. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics* **SMC-13,** 834–846 (1983).

38. Durrant-Whyte, H., Roy, N. & Abbeel, P. in *Robotics: Science and Systems VII* 73–80 (2012).

39. Agarap, A. F. Deep Learning using Rectified Linear Units (ReLU). arXiv: 1803.08375 [cs.NE] (2019).

40. Kingma, D. P. & Ba, J. *Adam: A Method for Stochastic Optimization* 2017. arXiv: 1412.6980 [cs.LG].

41. Pedregosa, F. *et al. Scikit-learn: Machine Learning in Python* 2018. arXiv: 1201.0490 [cs.LG].