Explainable RL Policies by Distilling to Locally-Specialized Linear Policies with Voronoi State Partitioning

Senne Deproost[1,2] [0009−0009−4757−0290],
Denis Steckelmacher[1,2] [0000−0003−1521−8494], and
Ann Nowé[1,2] [0000−0001−6346−4564]

[1] Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
[2] BP&M, Flanders Make@VUB, Pleinlaan 2, 1050 Brussels, Belgium

Abstract. Deep Reinforcement Learning is one of the state-of-the-art methods for producing near-optimal system controllers. However, deep RL algorithms train a deep neural network, that lacks transparency, which poses challenges when the controller has to meet regulations, or foster trust. To alleviate this, one could transfer the learned behaviour into a model that is human-readable by design using knowledge distillation. Often this is done with a single model which mimics the original model on average but could struggle in more dynamic situations. A key challenge is that this simpler model should have the right balance between flexibility and complexity or right balance between balance bias and accuracy. We propose a new model-agnostic method to divide the state space into regions where a simplified, human-understandable model can operate in. In this paper, we use Voronoi partitioning to find regions where linear models can achieve similar performance to the original controller. We evaluate our approach on a gridworld environment and a classic control task. We observe that our proposed distillation to locally-specialized linear models produces policies that are explainable and show that the distillation matches or even slightly outperforms the black-box policy they are distilled from.

Keywords: reinforcement learning, explainable ai

1 Introduction

With the evergrowing demand for complex automation across domains, the need for advanced controllers is outweighing the available engineers that can program them [1]. Optimal control techniques such as Model Predictive Control can optimize a controller given a global model of the system is provided [2]. However, in case no model is available or the system is highly dynamic, Deep Reinforcement Learning (DRL) can learn an optimal controller using repetitive interactions with the system [3]. DRL has shown state-on-the-art performance in many applications and is being increasingly integrated in the controller generation methodology [4, 5, 6]. However, the capabilities and strength of DRL comes at the cost of

users not understanding how the controller works. The artificial neural network at the core of the controller is complex and gives little insight into why and how it behaves. This raises distrust among users who should implement, test, deploy and operate the controller. To alleviate this lack of understanding, the field of *Explainable Reinforcement Learning* (XRL) introduced several solutions [7]. Here, explanation generating techniques aim to answer questions such as *what* is the global control strategy, *why* a controller behaves in a certain way, *why not* another behaviour, *when* to expect this outcome etc. Often this involves the creation of a surrogate model that is by design more interpretable than the original network while retaining most of the performance. Then, using a technique called knowledge distillation, the surrogate learns post-hoc how to mimic the original model [8]. From the myriad of human-readable classes that exist, those based on visual or written decision boundaries are the most informative for controller design. For example, rule-based decision trees construct a hierarchical set of splitting nodes in the state space with a defined behaviour in the leaf nodes. Summarizing the splits along the dimensional axis needed to reach a leaf can be seen as an explanation when that behaviour is performed. This method has shown to produce good tree models that can mimic network behaviour [9, 10]. However, we argue that a direct translation to any explainable surrogate limits the types of users that could interact with it. The product life cycle of a controller involves many stakeholders, each with different technical expertise. The control tester is not required to have the same domain specific language (DSL) as the control implementer. A factory worker only needs to interact via the user interface while the operational technician needs to tweak the controller parameters. If instead we would opt for a common language that is explainable to every user, we would omit the use of nuances specific to the user's DSL. Aside communication, fixing a DSL could limit the performance of the controller if the chosen language is unable to capture complex behaviour. With this shortcoming in mind, we propose an intermediate step between network and explanation. By partitioning the operational state space of the controller into regions with arbitrary simple behaviour, we could provide an interpretable representation that delivers insight for each type of user. For an initial version of this idea, we want to find regions where linear functions can operate in with similar performance to the original DRL network. The parameters, or weights, of these models indicate the importance of each input variable in forming the controller output signal while the bias term is an offset factor to this weighted sum.

Contributions: In this paper, we propose a knowledge distillation algorithm that splits the operational state space of the controller into Voronoi cells. This post-hoc method uses a trained DRL agent to gather experiences in the form of state-action pairs to learn the linear models associated to these cells. The models are continuously optimized and a periodic update is performed to decide their decision boundaries based on their loss in following the original model. To validate our method, we chose a continuous space gridworld and a control task. We observe the capability our algorithm to find these linear subpolicy regions while staying close to the performance of the original DRL policy.

2 Background

2.1 Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm for solving sequential decision problems [11].These are defined as a Markov Decision Process (MDP) and formulated by tuple $(S, A, \mathcal{P}^a_{ss'}, \mathcal{R}^a_{ss'})$ in a control problem scheme [12]. Here, the state space S consists of any state s_t that can be observed at t with $t \in [0, t_{max}]$ while A denotes the action space with a_t the action performed at time t. $\mathcal{P}^a_{ss'}$ is the probability distribution for each state transition $s_t \rightarrow s_{t+1}$ when a_t is performed and $\mathcal{R}^a_{ss'}$ associates a reward signal to each transition. Reward is a value indicating how good or bad the chosen actions are in following a certain control objective. The MDP is subject to the Markovian property $Prob\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\}$, where the state transition is independent from past transitions. A policy π_t is a mapping $s_t \rightarrow a_t$ that decides the chosen behaviour at each timestep. The objective of the policy is to maximize the discounted sum of rewards $R(\tau) = \sum_t \gamma^t r_{t+1}$, whereby $\gamma \in [0, 1[$ as a discount factor.

2.2 Black Box Deep Reinforcement Learning

The first RL algorithms represented their policy using tabular values and were only applicable to discrete state and action spaces. Should the policy operate in continuous space, a discretization of the learned values has to be applied with sufficient resolution. How accurate the policy can represent these values is bound by the discretization error, which reduces with table size [13]. The higher the resolution, the more accurate these values are learned. However, control policies for complex systems often require a small resolution, causing an explosion of table size and therefore system memory. A solution is to learn these values using an artificial neural network since they are considered universal function approximators and can generalize well over unforeseen inputs [3, 14]. For visual inputs, the state could be transformed into an embedding using convolutions before using it as input to a regular feed forward neural network. This Deep Reinforcement Learning (DRL) approach has proven itself quite successful in a myriad of applications, especially to control complex dynamic systems. However, transparency of the policy is lost due to the computational complexity of a neural network. The large amount of parameters describing the operations done at inference are only informative on a global level in relation with the parameters of all other layers. In addition, two sets of different parameterized networks can yield similar behaviour rendering the semantic meaning of one parameter useless. Recently, the field of Explainable Reinforcement Learning (XRL) has researchers investigate new methods to represent the policy in a more human-understandable manner [7]. Introduced methods can describe network behaviour on both local and global level. The method we present aims for a global description using simple linear models. However, we do note that the local description of a region is only given when performing inference with a nearest neighbor algorithm.

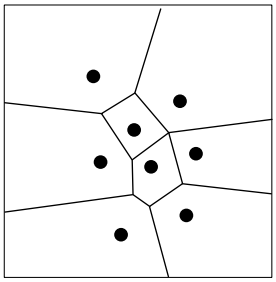


Fig. 1: An example of a Voronoi diagram using 8 codeword points.

For a more globally informed explanation, the region should be described in a meaningful way independent from all other regions.

2.3 Voronoi quantization

Vector quantization is a compression technique that groups similar datapoints into regions described by a single representative codeword point $c \in \Re^n$ [15]. A most common one, nearest neighbor vector quantization, defines c to be the point closest to each other point in the region. This quantizer $\psi \rightarrow \Re^n$, the Voronoi quantizer, maps each n-dimensional vector $x \in \Re^n$ onto a finite set of codewords $C = \{c_1, c_2, ..., c_m\} \subset \Re^n$, both from the set $\Re$ of real numbers. This mapping creates a partitioning of m disjoint Voronoi cells of the vector space following

$$R_k = \{x \in \Re^n : \psi(x) = c_k\} \tag{1}$$

for any $i = 1, 2, .., m$.

The mapping of a given vector x to a codeword c_i is given by

$$R_k = \{x : \|x - c_k\| \leq \|x - c_j\|\}, \forall k \neq j \tag{2}$$

which implies that c_i is the nearest neighboring codeword of x.

The main benefit of this quantization is the simple representation of the regions which is only the list of codewords C. The mapping from an arbitrary point to a region R_k can efficiently be done with a nearest neighbor search over all codewords. Often this is done using a kd-tree [16] which has an average time complexity of $O(\log n)$ to lookup the region-defining codeword. The intuition behind Voronoi quantization is that a region R_k contains all states that are similar to codeword c_k according to a definition of distance. In a later section, we will motivate that the capability of a subpolicy model in following the original global model dictates the selection of these codewords.

3 Related work

There are two approaches commonly used in XRL: those that use an inherently interpretable formalism and those that post-hoc imitate a pre-trained policy using state-action pairs. Our approach belongs to the second category. This latter technique, knowledge distillation [17], has been used to transfer the behaviour from a complex model to a simpler one. Using Soft Decision Trees, decision nodes describe the splits needed to come to a behaviour defined in the leaf nodes [9]. The usage of linear models as an explanation was popularized by Ribeiro et al. using their algorithm LIME [18]. With it, they can generate linear functions for classification tasks to explain the decision of a model at a local level. In Reinforcement Learning, LIME has been used to explain the behaviour [19] as well as the used reward function [20].

Our work extends upon earlier work done by Lee et al. [21]. Their online TD-AVQ algorithm was an attempt to use TD-learning in continuous environments. Instead of improving TD-learning, we mainly want to use this idea to find linear models instead of discrete actions, generalizing the behaviour explanation over a wider region compared to a single discrete action. The partitioning algorithm, based on Voronoi cells, aggregates states based on the amount of reward that could be collected. When a certain threshold is exceeded, a new cell is created with the codeword is introduced and the corresponding action is associated with that region. A minimum distance threshold ensured that newly formed regions where not too small, determining the resolution of the partitioning. Additionally, they included a recurring step where a codeword would be removed if neighboring cells had similar action values and therefore can be merged. Their algorithm has been shown good performance in learning a policy for a navigation gridworld as well as a simple mechanical gripper set up.

	TD-AVQ	Ours
Distillation	No	Yes
Focus on explainability	No	Yes
Using modern deep learning	No	Yes
Splitting criteria	Accumulated reward	Prediction loss

Table 1: The key differences between TD-AVQ and our method. Whereas TD-AVQ is an improvement on traditional TD-learning, we want to distill interpretable linear policies.

4 Partitioning the state space

The main idea behind our approach is to find regions in state space S where a model $\tilde{\pi}$ of arbitrary complexity can produce behaviour that is close to the

original policy π. In this paper, we limit ourselves to linear models but note that this method is model-agnostic.

In algorithm 1 we describe the main loop that in each iteration both trains the subpolicies and manages the partitions of the state space. The loop runs for $n_{\texttt{epochs}}$ and performs operations on the partitions at different iteration frequencies. Every $n_{\texttt{split}}$ iterations, the algorithm finds regions that are eligible to be split due to their performance being insufficient on a given trajectory. Every $n_{\texttt{merge}}$ iterations, each region is compared to its neighbors to check whether or not their learned subpolicy is similar and if one of them can be removed. The last $n_{\texttt{freeze}}$ iterations do not alter the partitioning but instead only optimize the subpolicies.

The list $\tilde{\pi}$ is initially comprised of only one subpolicy $\tilde{\pi}_0$ with its parameters set arbitrarily. The list of codewords C has only c_0, which is the first state s_0 that will be observed in the first iteration. To map an observed state s_t to a subpolicy $\tilde{\pi}_i$, a kd-tree [16] is constructed using the list of all codewords C. With the state, inference $\texttt{kd-tree}(C, s_t)$ is performed to find the single nearest neighbor codeword based on Manhattan distance. The returned index i corresponds with the index of the policy in the subpolicies list $\tilde{\pi}$.

Algorithm 1 Voronoi State Partitioning

Require: Trained policy π, environment `env`, empty buffer B
Initialize list of subpolicies $\tilde{\pi}$ with policy $\tilde{\pi}_0$ and s_0 as codeword $c_0 \in C$
for $n = 0$ to $n_{\texttt{epochs}}$ **do**
  for $t = 0$ to t_{max} **do** ▷ Collect experiences
    $a_t \leftarrow \pi(s_t)$
    Perform a_t in `env`, observe s_{t+1}
    $i = \texttt{kd-tree}(C, s_t)$ ▷ Find nearest codeword
    Add s_t and a_t to buffers $\tilde{\pi}_i$
    Add $s_t$to B
    if terminal(s_t) **then**
      End episode
    end if
  end for
  for each $\tilde{\pi}_i \in \tilde{\pi}$ **do** ▷ Train subpolicies with their buffers
    $\texttt{train}(\tilde{\pi}_i)$
  end for
  if $n < n_{\texttt{freeze}}$ **then**
    for each $n_{\texttt{split}}$ iteration **do** ▷ Split regions
      $\texttt{split\_regions}(C, \tilde{\pi})$
    end for
    for each $n_{\texttt{merge}}$ iteration **do** ▷ Merge regions
      $\texttt{merge\_regions}(C, \tilde{\pi})$
    end for
  end if
  $\texttt{reset}(B)$
end for

4.1 Learning subpolicies

Each iteration n, a full episode in the environment is performed with RL policy π to gather both state transitions $s_t \leftarrow s_{t+1}$ and selected actions a_t. Each state s_t is orderly stored in buffer B for 1 iteration to be used as trajectory for the splitting of regions. At each step, a lookup is performed to find the closest code word with the `kd-tree` and both state and action are added to the buffer of the corresponding subpolicy. After every episode, each subpolicy is trained using a MSE loss with mini batches of their corresponding buffers. We note that we deliberately want to overfit the models since newly seen data should be captured by new subpolicies if needed. The last $n_{\texttt{split}}$ iterations are used to only train the subpolicies and not to alter the partitioning. This is done to avoid lower performance of newly introduced subpolicies right before the final iteration is reached.

4.2 Splitting regions

Splitting one region c_i into two is based on the performance of subpolicy $\tilde{\pi}_i$ when mimicking π in that region (Alg. 2). Every $n_{\texttt{split}}$ episodes, the gathered episode trajectory of π is traversed a second time using actions from $\tilde{\pi}$. For each encountered state s_t that is part of a new region c_i, the corresponding subpolicy with index $i = \texttt{kd-tree}(C, s_t)$ is retrieved and a list of regional losses $\texttt{loss}_{\tilde{\pi}_i}$ is initialized empty.

With every state that is contained within the region, inference with both $\pi(s_t)$ and $\tilde{\pi}_i(s_t)$ is performed. Loss is calculated as the mean squared error (MSE) between action a_t^{π} and $a_t^{\tilde{\pi}_i}$ and added to $\texttt{loss}_{\tilde{\pi}_i}$.

If a state is part of a different region, $\texttt{loss}_{\tilde{\pi}_i}$ is reinitialized to 0. When the mean of gathered regional losses exceeds threshold value `max_loss` at s_t and the distance between that state and regional codeword c_i exceeds `min_pol_distance`, a split is performed. A new subpolicy is initialized randomly and added to $\tilde{\pi}$ and s_t becomes a new codeword c_j added to C. Since this split effects all bordering regions, the neighbours of the old region are found using *Delaunay Triangulation* [22] and their buffers are reset to avoid experiences that would be under control of the newly formed policy.

The intuition behind this approach is that a trained subpolicy is only capable of performing behaviour of a certain complexity in a region of the state space. If the policy performs good enough, it can handle these states. If the loss starts increasing, it has difficulties to perform the behaviour at this moment in its training. If however too much loss is accumulated, we could see that moment as the first encountered most difficult region. Adding another subpolicy at that point gives the other subpolicy a better demarcated region of state space where it has proven itself before. The newly introduced subpolicy starts with empty buffers and initial learning conditions.

Algorithm 2 `split_regions`

$i_{\texttt{prev}} = \texttt{kd-tree}(C, s_0)$
$\texttt{loss}_{\tilde{\pi}_i} = 0$
for $s_t \in B$ **do**
  $i = \texttt{kd-tree}(C, s_t)$
  if $i \neq i_{\texttt{prev}}$ **then**
    $i_{\texttt{prev}} = i$
    $\texttt{loss}_{\tilde{\pi}_i} = 0$
  end if
  $\texttt{loss}_{\tilde{\pi}_i} \leftarrow \texttt{loss}_{\tilde{\pi}_i} \cup \text{MSE}(\pi(s_t), \tilde{\pi}_i(s_t))$
  if $\text{mean}(\texttt{loss}_{\tilde{\pi}_i}) > \texttt{pol\_loss}_{\texttt{max}}$ and $\|s_t - c_i\| > \texttt{min\_pol\_disance}$ **then**
    Add s_t to C
    Add new $\tilde{\pi}_i$ to $\tilde{\pi}$
    $M \leftarrow \text{neighbours}(\tilde{\pi}_i)$
    $\texttt{reset\_buffers}(M)$
  end if
end for

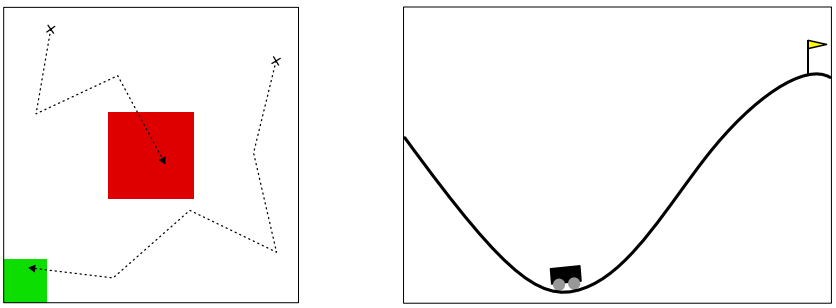
4.3 Merging regions

When performing splits, the newly chosen codewords could be sub-optimal since we could further train the affected subpolicies. If a split occurs in regions where the behaviour of the original policy differs little, neighbouring subpolicies could emerge with similar behaviour and parameters. To avoid this, every $n_{\texttt{merge}}$ iterations a pairwise comparison between each subpolicy and its neighbors M is performed. As a measurement of similarity, we use the $L\infty$ norm of both sets of subpolicy parameters. If this norm is above a certain threshold, the subpolicies are different enough to be kept. If this value is below this threshold, the `merge_regions` procedure merges the regions (Alg. 3). This removes subpolicy $\tilde{\pi}_j$ and associated codeword c_j from the known regions. The buffer of the remaining subpolicy $\tilde{\pi}_i$ is reset together with all neighboring subpolicies in M since their decision boundaries are impacted as well.

Algorithm 3 `merge_regions`

for each $\tilde{\pi}_i \in \tilde{\pi}$ **do**
  $M \leftarrow \text{neighbours}(\tilde{\pi}_i)$
  for each $j \in M$ **do**
    if $\|\tilde{\pi}_i.parameters_n - \tilde{\pi}_j.parameters_n\|_\infty < \texttt{min\_param\_distance}$ **then**
      Remove $\tilde{\pi}_j$ from $\tilde{\pi}$
      Remove c_j from C
      $\texttt{reset\_buffers}(M)$
    end if
  end for
end for

5 Experimental Validation



(a) SimpleGoal (b) MountainCarContinuous

Fig. 2: The two environments used to validate our method.

To validate our approach, we used both a continuous space gridworld, called `SimpleGoal`, as a navigation and `MountainCarContinuous` from the Gymnasium library as a classic control problem [23, 24]. As the original policy, we used a standard version of TD3, a well-known DRL algorithm [25]. For `SimpleGoal` we trained a TD3 agent for 500.000 steps using the standard parameters while for `MountainCar` we used a pretrained one from the Stable-Baselines 3 / RL Zoo Hugging Face repository [26, 27].

Evaluation is done by analyzing the spread of the gathered episodic returns. To account for the stochasticity of our distillation approach we use a DRL policy to obtain 85 distilled policies policies for `SimpleGoal` (i.e we apply algorithm 1, 85 times using the same DRL policy) and 40 for `MountainCarContinuous`. We evaluate each of the distilled policies for 1000 runs using random start states and observe the episodic returns. This yields a spread of 85000 and 40000 individual runs that we use to compare the DRL's 1000 evaluations with. Outliers are not shown on the boxplot but are analyzed afterwards. Since both environments have an action space of size 2, we can visualize the produced policies for an improved understanding how the algorithm performs the partitioning.

Training the DRL policy and subpolicies were done on a M2 MacBook Pro with 16GB of RAM. The used deep learning library is PyTorch v2.3.1 in a Python 3.11.9 environment using Adam as the parameter optimizer [28, 29].

5.1 Navigation task

Description In `SimpleGoal` (Fig. 2a) has to navigate in continuous space towards a goal region while avoiding a pitfall in the middle. The task is performed in a bounded space 1.0×1.0 with the goal being located at $x < 0.1, y < 0.1$ and the pitfall at $0.4 < x < 0.6, 0.4 < y < 0.6$. The observation space is the current

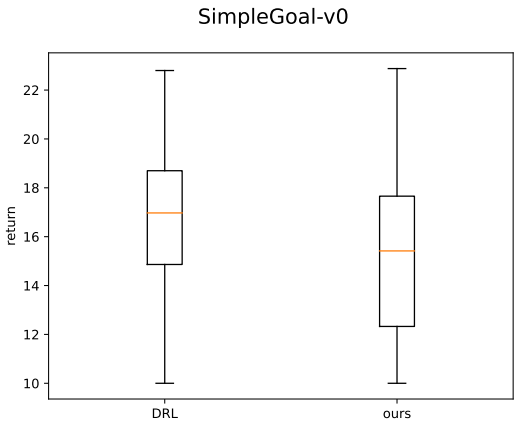
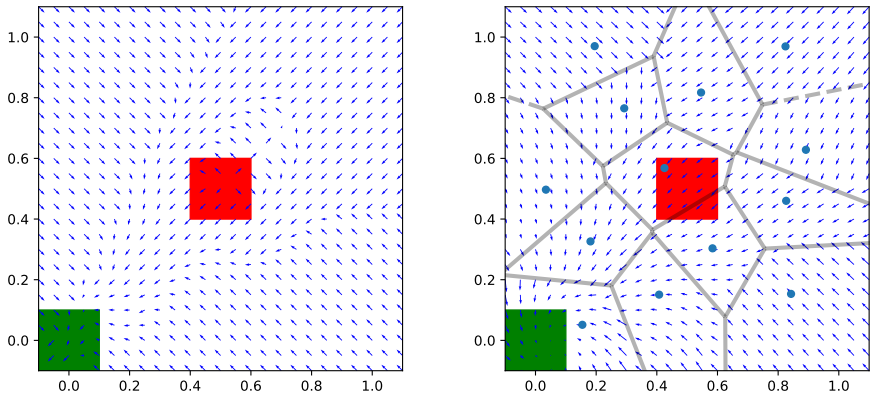


Fig. 3: Spread of performance in terms of achieved return (higher is better). The DRL agent is evaluated on 1000 evaluations while 85 instances of our method were evaluated on the same 1000 episodes for a total of 85000 runs. We observe similar spread and performance in both settings.

x and y coordinate of the agent. The action, with space bound by $[-1, 1]$, is the change in x and y for the next step and is calculated by $dx = 0.1a_0$ and $dy = 0.1a_1$. Each timestep, a reward of $r_t = 10*(\texttt{old\_distance}-\texttt{new\_distance})$ is returned based on the progress the agent makes in approaching the goal. An additional reward of 10 is given if the goal is reached within the truncation time of 50 steps. When the agent enters the pitfall, a reward of -10 is given and the episode is terminated.

Performance The performance of our algorithm for 40×1000 runs is reported in figure 3 (right part) and compared to the 1000 runs of the DRL agent (left part). The DRL agent has returns in a range of $[-10.0, 22.667]$ with mean return 16.599 standard deviation 3.117. 24 outliers have been observed, with a collective mean of 8.369 and standard deviation 5.539. They are spread over range $[-10.0, 10.364]$. For performance on all runs, we observe an average return of 12.657 with standard deviation 8.436. with values ranging $[-11.790, 22.798]$. The outliers, in total 11067, have a mean of -8.042 and a standard deviation of 1.830. They spread over a range of $[-11.790, 1.192]$.

If we summarize the overall performance of the method, we see it can learn the `SimpleGoal` environment with a slight decrease in performance. The median over all training sessions is lower but its spread is similar to the DRL agent. The observed outliers over all runs, often in the negative range, indicate that the number of linear policies in decisive regions such as around the pitfall should



(a) TD3 (b) Distilled 13 locally-linear policies.

Fig. 4: On SimpleGoal: original black-box policy learned with TD3, and the result of its distillation to explainable locally-linear policies.

be increased or the behaviour of existing subpolicies should be more complex to avoid negative reward being encountered.

Visualization We visualize the behaviour of each policy using a quiver plot where each arrow originates from a state and points into the direction of movement indicated by the policy (Fig. 4). The linear functions that makes up this partitioning can be found in appendix B.1.

We notice that, globally, the plot of the subpolicies tends to follow the original DRL one. The 13 partitions of the state space are distributed seemingly uniformly with key regions at the goal state and around the pitfall. The borders between cells indicate a sudden difference in behaviour, something we could observe in the DRL one as well. The general trend is a movement towards the goal with behaviour to go around the pitfall region. However, we notice that in several important states around the pitfall the agent fails to avoid it.

5.2 Control task

Description `MountainCarContinuous` is a classic control problem where a toy car has to climb out of a sinusoidal valley towards the top of a hill. The observation space is the position of the car $x \in [-1.2, 0.6]$ along the x-axis and the velocity $v \in [-0.07, 0.07]$. The action space is continuous and its value, applied force F, is bound by $F \in [-1, 1]$. The goal state is located at $x = 0.45$ on top of the hill. The truncation length of an episode is 1000 steps. For each step, a reward of $-0.1 * F^2$ is returned, which penalizes the agent applying performing large forces on the car. When the goal is reached, a reward of 100 is given and the episode is terminated with success.

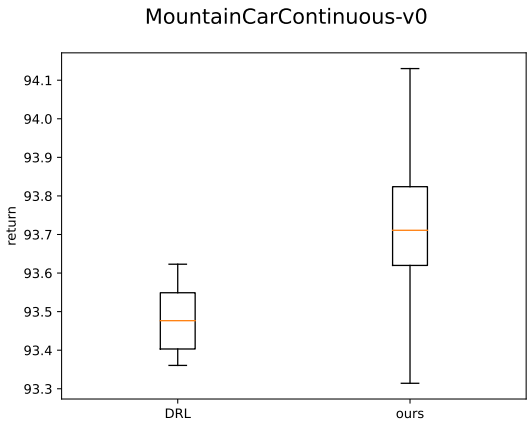
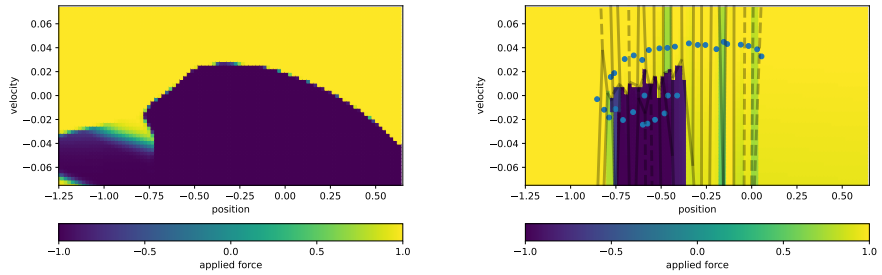


Fig. 5: Spread of performance in terms of achieved return (higher is better). The DRL agent is evaluated on 1000 evaluations while 40 instances of our method were evaluated on the same 1000 episodes for a total of 40000 runs. We observe a larger spread in our method and a higher median performance compared to the DRL agent.

Performance The 85×1000 runs of our algorithm are reported in figure 5 (right part) and compared to the 1000 runs of the DRL agent (left part). TD3 performs over all episodes with a mean return of 93.481 and a standard deviation of 0.075 within a range of $[93.360, 93.622]$. No outliers were observed. For all runs with our method, we observe a mean of 93.534 with standard deviation of 1.345 and a range of $[71.050, 94.680]$. 3130 outliers are spread with 73.19% of values below the spread lowerbound of 93.313 and 26.81% above upperbound of 94.130. Respective means are 90.116 and 94.221 with standard deviation 4.338 and 0.110.

Over all runs, we observe a higher median compared to DRL. When looking at the mean per run of the algorithm, we observe that the partitioning consistently produces policies with higher returns. We do note that the difference is quite small relative to the return gained by DRL. The outliers indicate training where the algorithm did not cover the space well enough but still manages to fulfill the task (a positive return always indicates success).

Visualization Both the DRL policy and best subpolicies are visualized using a hotmap over their state space (Fig. 6). The x-axis indicates the position of the car while velocity is given on the y axis. At each point, the color indicates the amount of force applied on the car. The 32 regions of the partitioned space have their defining codeword in a spiral-like shape that closely follows the hill of the DRL policy at higher velocities. The region with negative force (colored blue to purple) is much smaller with the linear functions and has a slightly higher value



(a) TD3 (b) Ours

Fig. 6: On MountainCar: original black-box policy learned with TD3, and the result of its distillation to explainable locally-linear policies.

(Appendix B.2). We also notice that the decision boundary between negative and positive force is more rigid compared to DRL.

6 Discussion

In this paper, we presented a new representation of a Deep Reinforcement Learning using Voronoi State Partitioning. By searching regions where a simple linear model can perform well enough to closely follow a DRL policy, we are able to have global insight in the behaviour of the policy. We validated the approach on both a navigation task and a control task and observed promising, even improved, results compared to the original policy.

Both validation environments have an observation space dimensionality of 2. However, for partitioning a state space with high dimensionality, the kd-tree lookup becomes exponentially less efficient with the number of dimensions [16]. This *curse of dimensionality* also affects lookup accuracy since with high dimensions, an observation that is equal on all dimensions but one could be considered similar. The definition of nearest points in space doesn't hold anymore.

The chosen hyperparameters were retrieved ad-hoc. A more rigid search for a better learning setting would yield an improved coverage of subpolicies over the state space.

Interpretability, and eventually explainability, of a DRL policy is our main motivation for designing this algorithm. However, due to the way regions are constructed by codeword points and their position in state space relative to other points, it becomes difficult to interpret the demarcation of a region. The method allows for the interpretation of the policy only if inference is performed on the Voronoi cells using a kd-tree. A future version should instead define regions along the axis of the state space, something rule-based and tree ensembles are capable off.

At last, we emphasize the model-agnostic nature of the algorithm. Every class of model with learnable parameters could be used to stand in for regional

behaviour. In a later iteration of this approach, we would investigate more controller like schemes such as PID controllers.

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

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

Reproduction Code is made available for reproduction at `https://gitlab.ai.vub.ac.be/sdeproost/IPR.git` under MIT license.

References

[1] European Labour Authority et al. *Report on Labour Shortages and Surpluses – 2024*. Publications Office of the European Union, 2025.

[2] James Blake Rawlings, David Q. Mayne, and Moritz Diehl. *Model Predictive Control: Theory, Computation, and Design*. 2nd edition. Madison, Wisconsin: Nob Hill Publishing, 2017. 623 pp. ISBN: 978-0-9759377-3-0.

[3] Volodymyr Mnih et al. "Human-Level Control through Deep Reinforcement Learning". In: *Nature* 518.7540 (Feb. 26, 2015), pp. 529–533. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14236. URL: https://www.nature.com/articles/nature14236.

[4] Marc G. Bellemare et al. "Autonomous Navigation of Stratospheric Balloons Using Reinforcement Learning". In: *Nature* 588.7836 (Dec. 2020), pp. 77–82. ISSN: 1476-4687. DOI: 10.1038/s41586-020-2939-8. URL: https://www.nature.com/articles/s41586-020-2939-8.

[5] Jonas Degrave et al. "Magnetic Control of Tokamak Plasmas through Deep Reinforcement Learning". In: *Nature* 602.7897 (Feb. 17, 2022), pp. 414–419. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/s41586-021-04301-9. URL: https://www.nature.com/articles/s41586-021-04301-9.

[6] Chengxi Li. "Deep Reinforcement Learning in Smart Manufacturing_ A Review and Prospects". In: *CIRP Journal of Manufacturing Science and Technology* (2023).

[7] Yanzhe Bekkemoen. "Explainable Reinforcement Learning (XRL): A Systematic Literature Review and Taxonomy". In: *Mach Learn* (Nov. 29, 2023). ISSN: 0885-6125, 1573-0565. DOI: 10.1007/s10994-023-06479-7. URL: https://link.springer.com/10.1007/s10994-023-06479-7.

[8] Guangda Ji and Zhanxing Zhu. "Knowledge Distillation in Wide Neural Networks: Risk Bound, Data Efficiency and Imperfect Teacher". In: *Proceedings of the 34th International Conference on Neural Information Processing Systems*. Nips '20. Vancouver, BC, Canada and Red Hook, NY, USA: Curran Associates Inc., 2020. ISBN: 978-1-7138-2954-6.

[9] Youri Coppens et al. "Distilling Deep Reinforcement Learning Policies in Soft Decision Trees". In: *International Joint Conference on Artificial Intelligence*. 2019. URL: https://api.semanticscholar.org/CorpusID:201700841.

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

[11] Richard S. Sutton and Andrew Barto. *Reinforcement Learning: An Introduction*. Nachdruck. Adaptive Computation and Machine Learning. Cambridge, Massachusetts: The MIT Press, 2014. 322 pp. ISBN: 978-0-262-19398-6.

[12] Richard Bellman. "A Markovian Decision Process". In: *Indiana University Mathematics Journal* 6 (1957), pp. 679–684. URL: https://api.semanticscholar.org/CorpusID:123329493.

[13] Lucian Busoniu et al. *Reinforcement Learning and Dynamic Programming Using Function Approximators*. 1st ed. USA: CRC Press, Inc., 2010. ISBN: 1-4398-2108-9.

[14] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer Feedforward Networks Are Universal Approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[15] R. Gray. "Vector Quantization". In: *IEEE ASSP Magazine* 1.2 (Apr. 1984), pp. 4–29. ISSN: 1558-1284. DOI: 10.1109/MASSP.1984.1162229.

[16] Songrit Maneewongvatana and David M. Mount. "On the Efficiency of Nearest Neighbor Searching with Data Clustered in Lower Dimensions". In: *Computational Science — ICCS 2001*. Ed. by Vassil N. Alexandrov et al. Red. by G. Goos, J. Hartmanis, and J. Van Leeuwen. Vol. 2073. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 842–851. ISBN: 978-3-540-42232-7 978-3-540-45545-5. DOI: 10.1007/3-540-45545-0_96. URL: http://link.springer.com/10.1007/3-540-45545-0_96.

[17] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. Mar. 9, 2015. arXiv: 1503.02531 [cs, stat]. URL: http://arxiv.org/abs/1503.02531. Pre-published.

[18] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Kdd '16. San Francisco, California, USA and New York, NY, USA: Association for Computing Machinery, 2016, pp. 1135–1144. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939778. URL: https://doi.org/10.1145/2939672.2939778.

[19] Ziyang Lu et al. *Explainable AI for Radar Resource Management: Modified LIME in Deep Reinforcement Learning*. June 26, 2025. DOI: 10.48550/arXiv.2506.20916. arXiv: 2506.20916 [cs]. URL: http://arxiv.org/abs/2506.20916. Pre-published.

[20] Jacob Russell and Eugene Santos Jr. "Explaining Reward Functions in Markov Decision Processes". In: The Thirty-Second International Florida Artificial Intelligence Research Society Conference. Sarasota, May 2019.

[21] Ivan S.K. Lee and Henry Y.K. Lau. "Adaptive State Space Partitioning for Reinforcement Learning". In: *Engineering Applications of Artificial Intelligence* 17.6 (Sept. 2004), pp. 577–588. ISSN: 09521976. DOI: 10.1016/j.engappai.2004.08.005. URL: https://linkinghub.elsevier.com/retrieve/pii/S0952197604000879.

[22] B. Delaunay. "Sur la sphère vide". In: *Bull. Acad. Sci. URSS* 1934.6 (1934), pp. 793–800.

[23] Andrew William Moore. *Efficient Memory-Based Learning for Robot Control*. University of Cambridge, 1990.

[24] Mark Towers et al. *Gymnasium: A Standard Interface for Reinforcement Learning Environments*. Nov. 8, 2024. DOI: 10.48550/arXiv.2407.17032.

arXiv: 2407.17032 [cs]. URL: http://arxiv.org/abs/2407.17032. Pre-published.

[25] Scott Fujimoto, Herke van Hoof, and David Meger. *Addressing Function Approximation Error in Actor-Critic Methods*. Oct. 22, 2018. arXiv: 1802.09477 [cs, stat]. URL: http://arxiv.org/abs/1802.09477. Pre-published.

[26] Antonin Raffin. *RL Baselines3 Zoo*. GitHub, 2020. URL: https://github.com/DLR-RM/rl-baselines3-zoo.

[27] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[28] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[29] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

A Hyperparameters

Hyperparameter	SimpleGoal-v0	MountainCarContinuous-v0
$n_{\texttt{epochs}}$	5000	2000
$n_{\texttt{split}}$	20	50
$n_{\texttt{merge}}$	100	100
$n_{\texttt{freeze}}$	1000	400
$n_{\texttt{reset}}$	500	500
`min_param_distance`	0.5	0.3
`min_pol_distance`	0.3	0.04
`max_pol_loss`	0.0001	0.00001
`one_split`	False	False

Table 2: Used hyperparameters for each environment

B Best linear policies

B.1 SimpleGoal

Codeword	Δx	Δy
`[0.891, 0.628]`	$-0.148x - 0.021y - 0.055$	$-0.420x + 0.231y - 1.095$
`[0.826, 0.460]`	$-0.347x + 0.305y - 0.212$	$-1.087x - 1.370y - 0.319$
`[0.407, 0.150]`	$3.175y - 1.000$	$-4.127y - 0.710$
`[0.181, 0.326]`	$-4.588x + 0.045y - 0.620$	$3.134x - 0.082y - 0.978$
`[0.425, 0.568]`	$-1.267x + 0.966y - 0.056$	$-0.281x - 0.967y - 0.702$
`[0.292, 0.765]`	$-1.657x - 0.271y + 0.147$	$0.433x - 0.602y - 0.484$
`[0.154, 0.051]`	$-5.328x + 5.256y + 0.191$	$-2.583x - 4.501y - 0.461$
`[0.545, 0.817]`	$-0.124x - 0.035y - 0.964$	$0.082x - 1.027y + 0.226$
`[0.842, 0.153]`	$-0.628x + 0.406y - 0.385$	$-0.536x - 0.010y + 0.663$
`[0.034, 0.496]`	$-0.649x - 0.652y - 0.076$	$0.706x - 0.6946y - 0.463$
`[0.583, 0.303]`	$-0.290x - 0.470y - 0.855$	$0.103x - 3.580y + 0.991$
`[0.824, 0.969]`	$0.266x - 0.424y - 0.684$	$-0.598x - 0.256y - 0.420$
`[0.195, 0.970]`	$0.327x - 0.246y + 0.349$	$0.620x - 0.077y - 0.882$

Table 3: All codewords and subpolicies, rounded to 4 digits.

B.2 MountainCar

Codeword	F
[-0.592, 0.000]	$-0.375x + 3.004v - 1.205$
[-0.463, 0.000]	$1.664x + 2.371v - 0.211$
[-0.510, 0.040]	$-1.117x + 1.050v + 0.340$
[-0.568, 0.038]	$-0.707x + 0.454v + 0.585$
[-0.575, -0.024]	$0.8357x + 0.7049v - 0.514$
[-0.154, 0.045]	$-0.952x + 1.840v + 0.470$
[-0.655, -0.014]	$-0.007x + 0.840v - 0.992$
[-0.424, 0.041]	$-1.270x + 1.317v + 0.417$
[-0.298, 0.042]	$-0.840x + 1.083v + 0.701$
[-0.256, 0.042]	$-0.216x + 0.673v + 0.916$
[-0.135, 0.043]	$-0.97x - 0.264v + 0.888$
[-0.018, 0.041]	$-0.435x + 0.738v + 0.965$
[-0.711, -0.020]	$0.867x + 0.415v - 0.374$
[-0.481, -0.015]	$1.125x + 0.788v - 0.440$
[-0.854, -0.003]	$-1.006x + 0.301v + 0.150$
[-0.060, 0.043]	$-1.115x + 1.334v + 0.867$
[-0.788, -0.018]	$-0.416x - 0.778v + 0.390$
[-0.815, -0.012]	$-0.569x + 0.090v + 0.535$
[-0.778, 0.016]	$-0.774x + 1.08v + 0.376$
[-0.347, 0.044]	$-1.147x + 0.772v + 0.521$
[0.030, 0.039]	$1.199x + 0.872v + 0.734$
[-0.752, -0.011]	$0.871x - 0.737v + 0.018$
[-0.604, 0.030]	$-0.281x + 0.511v + 0.811$
[-0.194, 0.038]	$-1.299x + 0.470v + 0.705$
[0.054, 0.033]	$0.007x + 0.701v + 0.974$
[-0.412, 0.000]	$1.0258x + 0.073v - 0.464$
[-0.601, -0.024]	$0.7532x + 1.528v - 0.514$
[-0.701, 0.030]	$-0.801x + 0.917v + 0.412$
[-0.651, 0.033]	$-0.350x - 0.09v + 0.7741$
[-0.759, 0.018]	$-0.799x + 0.284v + 0.395$
[-0.534, -0.020]	$0.668x + 0.130v - 0.641$
[-0.465, 0.039]	$-0.155x + 0.835v + 0.602$

Table 4: All codewords and subpolicies, rounded to 4 digits. The colors of the functions represents the ones in figure 6b

C Validation spread metrics

C.1 SimpleGoal

Metric	DRL	Ours
Min	-10.0	-11.790
Max	22.667	22.798
Mean	16.599	12.657
Std	3.117	8.436
Q1	14.542	12.298
Median/Q2	16.956	15.382
Q3	18.687	17.646
IQR	4.145	5.348
Coverage	97.6%	86.17%

Table 5: Spread of returns for experiments on SimpleGoal.

C.2 MountainCar

Metric	DRL	Ours
Min	93.360	71.050
Max	93.622	94.680
Mean	93.481	93.534
Std	0.075	1.345
Q1	93.403	93.403
Median/Q2	93.476	93.476
Q3	93.548	93.548
IQR	0.145	0.145
Coverage	100%	92.18%

Table 6: Spread of returns for experiments on MountainCar.