On Reducing Undesirable Behavior in Deep-Reinforcement-Learning-Based Software

OPHIR M. CARMEL and GUY KATZ, The Hebrew University of Jerusalem, Israel

Deep reinforcement learning (DRL) has proven extremely useful in a large variety of application domains. However, even successful DRL-based software can exhibit highly undesirable behavior. This is due to DRL training being based on maximizing a reward function, which typically captures general trends but cannot precisely capture, or rule out, certain behaviors of the model. In this paper, we propose a novel framework aimed at drastically reducing the undesirable behavior of DRL-based software, while maintaining its excellent performance. In addition, our framework can assist in providing engineers with a comprehensible characterization of such undesirable behavior. Under the hood, our approach is based on extracting decision tree classifiers from erroneous state-action pairs, and then integrating these trees into the DRL training loop, penalizing the model whenever it performs an error. We provide a proof-of-concept implementation of our approach, and use it to evaluate the technique on three significant case studies. We find that our approach can extend existing frameworks in a straightforward manner, and incurs only a slight overhead in training time. Further, it incurs only a very slight hit to performance, or even in some cases — improves it, while significantly reducing the frequency of undesirable behavior.

CCS Concepts: • **Software and its engineering** → **Software development techniques**; • **Computing methodologies** → **Reinforcement learning**; • **Human-centered computing** → *Visualization techniques.*

Additional Key Words and Phrases: Deep Reinforcement Learning, Safety, Decision Trees, Explainability

ACM Reference Format:
Ophir M. Carmel and Guy Katz. 2024. On Reducing Undesirable Behavior in Deep-Reinforcement-Learning-Based Software. *Proc. ACM Softw. Eng.* 1, FSE, Article 68 (July 2024), 22 pages. https://doi.org/10.1145/3660775

1 INTRODUCTION

Deep reinforcement learning (DRL) is a paradigm that assists engineers in training RL agents, through the use of neural networks. DRL has proven remarkably powerful in settings involving sequential decision making, such as game playing [Lample and Chaplot 2017; Mnih et al. 2015; Ye et al. 2020], Internet congestion control algorithms [Jay et al. 2019], and smart transportation systems [Katz et al. 2017; Palanisamy 2020]. This trend is likely to intensify in coming years, with DRL taking part in an increasing number of mission-critical software systems.

Despite its impressive success, DRL has a significant drawback. As with other deep-learning-based methods, DRL models are *opaque*: it is highly difficult for humans to comprehend their internal decision making, and consequently to guarantee that the resulting software is error-free. This is not merely a theoretical issue: several undesirable behaviors have been observed in modern DRL-based software (e.g., [Eliyahu et al. 2021] and [Kazak et al. 2019]). If these issues are not addressed, they could hamper the deployment of DRL models in various domains of interest.

Authors' address: Ophir M. Carmel, ophir.carmel@mail.huji.ac.il; Guy Katz, g.katz@mail.huji.ac.il, The Hebrew University of Jerusalem, Jerusalem, Israel.

ACM 2994-970X/2024/7-ART68
https://doi.org/10.1145/3660775

It is generally accepted that many bugs, or inaccuracies, in DRL models stem from the *reward function* in use [Amodei et al. 2016]. In DRL, the reward function is the objective that the model is trained to optimize; and so, the resulting model can only be as good as its reward function. In complex systems, there are usually multiple goals to be satisfied simultaneously, leading to complex reward functions. Further, there are often multiple policies that can achieve high rewards, meaning that two models that display similar performance (i.e., achieve similar reward scores) may be quite different from each other, making them difficult to compare. Finally, even "good" models may present highly undesirable behavior — in cases that were not adequately addressed in the reward function. For example, it has been observed that the Aurora congestion control system [Jay et al. 2019] might sometimes choose to increase a sender's sending rate over an already congested network; or choose to decrease it even when the current bandwidth is extremely under-utilized [Eliyahu et al. 2021]. Both of these actions are clearly incorrect, even though the Aurora model is generally very successful. Such inaccuracies in DRL models motivate the main goal of this work: to improve the quality of DRL models, by significantly reducing the frequency in which they present undesirable behavior. Because achieving a high reward value does not guarantee the absence of undesirable behavior, achieving this goal could boost the safety of even state-of-the-art DRL models.

Here, we present a novel engineering methodology for reducing, and sometimes nearly preventing, undesirable behavior in DRL-based software. Our approach uses a *reward reshaping* technique [Wiewiora 2010]: it influences the DRL reward function in subtle ways, in order to eventually produce a model that satisfies the original requirements, but which is also less likely to produce undesirable behavior. A key novelty in our approach is the way in which bad behavior is expressed, which allows us to deal with the significant challenge of reducing undesirable behavior even when it is difficult to formulate this behavior as part of the reward function. Starting with an initial, already-trained model we (i) collect instances where the model presented undesirable behavior; (ii) generate from these state-action pairs a *decision tree* that expresses an infinite set of undesirable behaviors; and finally (iii) *inject* this decision tree back into the training process, generating an augmented model that is far more likely to operate correctly compared to the original.

Our approach includes several novel aspects that differentiate it from existing techniques. Most notably, numerous existing techniques for *safe reinforcement learning* (e.g. [Dalal et al. 2018; Tessler et al. 2018]) rely on the assumption that the undesirable behaviors are known, and are straightforward to manually specify. We argue that this is often not the case — and that in fact it may be quite difficult for engineers and stakeholders to pinpoint the root case of the undesirable behavior. Our method addresses this difficulty by placing only a minimal burden on the humans in the loop: they are only required to flag *bad* state-action pairs. Such a flagging is, by definition, one of the simplest cognitive tasks required as part of removing undesirable behavior; and it is usually both feasible [Karger et al. 2013; Sheng and Zhang 2019], and also easier than tracking down and analyzing the root cause of the undesirable behavior [Johnson et al. 2020; Leszak et al. 2000], or adjusting the reward function to attempt and rule it out — which may lead to additional unexpected results [Li 2017; Wiewiora 2010]. Another advantage of the flagging is that it only requires a basic, user-level understanding of the system at hand, this effort can be crowd-sourced, either through paid-for services or through bug reports provided by users. To the best of our knowledge, ours is the first approach that applies reward reshaping without a specification that is provided upfront.

Another key aspect of our approach is that once the labeling of undesirable state-action pairs is provided, it is used in training a decision tree that can, in fact, be used by engineers to *explain*, or better understand, the root cause behind the undesirable behavior. Such an understanding can assist the engineers in fine-tuning the reward function, if such a chance is deemed necessary, or at

least in better understanding the system's limitations. This part of our approach also entails human input, in the form of *grammars*, as we discuss later.

We further point out that our approach does not attempt to make the resulting DRL satisfy some global, hard constraints, which may be infeasible [Cai et al. 2023]; instead, the approach offer users a trade-off between accuracy and safety, and allow them to fine-tune it according to their specific needs. Finally, our approach is fairly straightforward to implement as an extension to a variety of existing frameworks, as we later demonstrate.

For evaluation purposes, we apply our approach to three diverse case studies: *Aurora*, *Traffic Control* and *Snake*. In two of these case studies, our approach succeeds in decreasing the frequency of undesirable behavior significantly, while maintaining (and even improving) performance, and without substantially increasing the required training time. In the third case study, we also succeed in decreasing the frequency of undesirable behavior significantly, albeit while slightly degrading performance. Our success in tackling these three, very different models, showcases the wide applicability of our approach. Our code and benchmarks are available online [Carmel and Katz 2024].

It is worth mentioning that, although our case studies centered on DRL models, our approach is actually more general — and can be applied to other kinds of RL models [Sutton and Barto 1998]. We chose to focus here on DRL because it is generally considered the state of the art in RL techniques [Jay et al. 2019; Lample and Chaplot 2017; Mnih et al. 2015; Palanisamy 2020; Silver et al. 2016; Ye et al. 2020], achieving superior scalability and successfully solving more complex tasks than its counterparts.

The rest of this paper is organized as follows. In Sec. 2 we present the necessary background on DRL and decision trees, and in Sec. 3 we describe the different steps of our approach. In Sec. 4 we describe our three case studies, and then describe how our approach was applied to them in Sec. 5. Related work is discussed in Sec. 6, and we conclude with Sec. 7.

2 BACKGROUND

2.1 Supervised Learning and Decision Trees

Supervised learning [Liu and Wu 2012] is a machine-learning paradigm, in which labeled samples are generalized into a function that maps previously unseen inputs into a set of outputs. In classification tasks, the set of outputs is finite. Decision trees [Kingsford and Salzberg 2008,?; Safavian and Landgrebe 1991] are a particular kind of classifiers, which resemble binary trees. Each tree node represents a query about the value of some data feature; and it splits the data into two sets, depending on whether the query evaluates to true or false (see Fig. 1). In a "good" decision tree, each node will split the data into two sets that are of similar cardinality, so that the tree does not become too deep. The splitting process continues with each internal node of the tree, until reaching a leaf, which corresponds to one of the possible output labels.

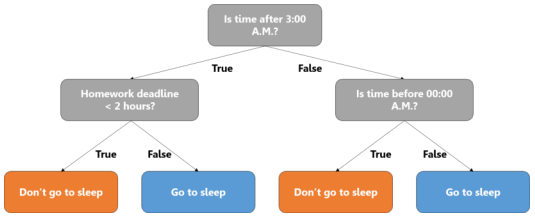


Fig. 1. An toy example of a decision tree classifier, with 2 labels: "Go to Sleep" and "Don't go to sleep".

Of the many existing forms of classifiers (e.g., deep neural networks), decision trees are considered to be fairly interpretable, because they consist of a sequence of queries pertaining directly to input features [Kingsford and Salzberg 2008; Safavian and Landgrebe 1991].

2.2 Deep Reinforcement Learning

Deep learning [LeCun et al. 2015] is a machine learning approach for training *deep neural networks* (*DNNs*). In a DNN, a complex input is processed in an iterative fashion, with each layer of the DNN computing a set of latent features, using both linear and non-linear transformations. Deep learning has had amazing success in recent years, due to its uncanny ability to generalize and learn complex structures.

Reinforcement learning (RL) [Sutton and Barto 2018] is a machine learning paradigm, in which an agent learns by iteratively interacting with its environment, while trying to maximize a *reward function*. In time step t of the execution, the environment is in some state s_t, and the agent selects some action a_t. The environment then transitions to state s_{t+1}, and provides the agent with a *reward* value, r_t, indicating how well the agent has performed. The agent's goal is to maximize the cumulative discounted reward; that is, to choose an action that maximizes the current reward, and also the next rewards. The agent does this by learning a policy π, which maps each state to the best possible action in that state. The RL training loop is illustrated in Fig. 2.

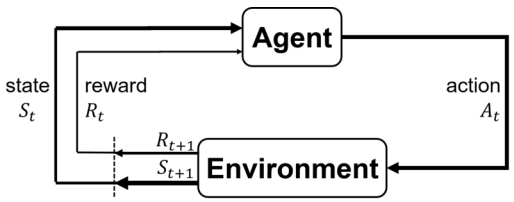


Fig. 2. (Borrowed from [Bhatt 2019]) The main loop of reinforcement learning.

One issue with RL is its limited scalability: learning an optimal, or even an approximately optimal policy has been observed to be computationally difficult in complex systems [Arulkumaran et al. 2017]. To overcome this limitation, engineers now apply *deep reinforcement learning* (DRL) [Arulkumaran et al. 2017; Li 2017; Mousavi et al. 2018; Wang et al. 2020], in which the policy being learned is expressed as a DNN. This allows tackling much more complex problems, which, without the DNN, would require significant manual effort and computing power. DRLs are implemented by various methods, including Deep Q-Learning, Actor-Critic, and Policy Optimization methods. DRL has repeatedly been demonstrated to be both scalable and powerful [Arulkumaran et al. 2017; Wang et al. 2020].

2.3 Reward Reshaping

Reward reshaping is an RL technique, in which external knowledge from domain experts is utilized to adjust the rewards provided to the agent in training, in order to improve the learned policy [Hu et al. 2020; Wiewiora 2010]. This technique can also expedite training, and is particularly useful in settings where reward accumulation is slow during early training [Wiewiora 2010]. When reward reshaping is applied, the reward r_t is replaced with some modified reward $r_t + f_t$, where f_t is decided by the external expert, as opposed to the environment. Common reward reshaping techniques seek to improve the performance of the resulting agent [Ng et al. 1999] and to reduce its training time [Wiewiora et al. 2003].

3 APPROACH

We propose a novel reward-reshaping based approach, aimed at allowing engineers to benefit from the advantages of DRL while reducing or avoiding the undesirable behaviors it usually entails. Our approach assumes that an initial DRL model has already been trained and deployed, and that it occasionally exhibits undesirable behavior — that the engineers would like to remove, or at least reduce. Our technique then allows the engineers to retrain a superior model.

The high-level steps of our approach are:

(1) Obtain traces of the initial DRL model, and label state-action pairs in those traces as desirable or undesirable. Then, generalize these labeled pairs into a decision tree.
(2) Manually inspect the decision tree to gain insight into the root cause of the problem; adjust the tree if needed.
(3) Leverage the decision tree to re-train the DRL model, with a reshaped reward function aimed at reducing the frequency of undesirable behavior.
(4) Analyze the results, possibly fine-tuning hyper-parameters if needed.

We now proceed to explain each of these steps in greater detail.

3.1 Detecting Undesirable Behavior

The first step of our approach entails characterizing the undesirable behavior of the model. Ideally, we would describe this behavior as a logical formula, but this is known to be difficult in practice [Clarke et al. 2018]. To circumvent this difficulty, we use only a simple form of specification: ⟨state, action⟩ pairs, labeled by a human expert to indicate whether selecting this action in this state is *desirable* or *undesirable*. We stress that the human in the loop needs not have any knowledge or understanding of the system's internal states; instead, she is only required to raise a flag when an undesirable action occurs. Once such a flagging occurs, our framework automatically records the ⟨state, action⟩ pair in question.

As a running example, consider the Aurora congestion control system [Jay et al. 2019]. Aurora's goal is to maximize the throughput of a computer network, by penalizing the agent when latency is observed or when packets are lost (additional details appear in Sec. 4). Consider a situation in which the agent observes perfect network conditions, i.e. low latency and no packet loss; and yet chooses to decrease the packet sending rate, which will likely decrease the network's throughput. Clearly, this is undesirable behavior. Here, we propose to rely on a human expert to mark such state-actions pairs as undesirable, even if that expert is unable, or unwilling, to write a logical formula that precisely captures these cases.

Once we have a set of state-action pairs marked as desirable or undesirable, our next step is to generalize them, through supervised learning, so that we are able to classify additional, previously unseen state-action pairs. We choose here to use decision trees, in order to benefit from their relative interpretability [Kingsford and Salzberg 2008; Safavian and Landgrebe 1991]. We next discuss how these trees are trained in our setting.

Grammars. Our choice of decision trees is geared towards improved explainability; but in order to fully tap their potential, we need their internal nodes to represent meaningful queries on the data, which could then assist humans in interpreting them. Naturally, different choices of features may be adequate for different problem domains. Thus, we parameterize our approach with a *grammar*, which defines the set of features that a tree may contain. We define what a grammar is in our context, and propose here some grammars that are adequate for common problem domains; and these may be fine-tuned to support additional domains, as needed.

Definition 3.1 (Grammars). *Observe a DRL agent, whose environment states $s_1, s_2, \ldots$ are comprised of a set of features of interest. In our context, A grammar rule is a function that can be applied to these features (individual features, or sets thereof). A* grammar *is a set of grammar rules.*

The idea behind a grammar is to allow the user to provide functions and predicates relevant to the problem at hand; and then use these to construct formulas that will populate the nodes of the decision tree. We regard the grammar rules as templates, and the features as instances of those templates. Clearly, different grammars may result in different decision trees. Even if these different trees, when integrated into the DRL training, result in similar reward values, they can afford highly varying degrees of interpretability: a "good" grammar, which uses predicates that are relevant to the system at hand, will result in a decision tree that is more straightforward for humans to comprehend (e.g., is smaller), and will thus better contribute to the system's explainability.

As a toy example, consider a DRL agent whose input (the model state) is a vector of real-valued numbers. A reasonable grammar to attempt in this case might consist of rules (functions and predicates) such as $\geq$, $>$ and $=$, *average* or *max*. Some of these rules might prove less appropriate if the model states are Boolean vectors.

As another illustrative example, suppose we wish to train a DRL controller for a navigating robot (a task in which DRL has been shown to be highly effective [Marchesini and Farinelli 2020]). The robot is equipped with LIDAR sensors, each measuring the distance to the nearest obstacle in the direction at which the sensor is directed; and in each time step, the robot moves in one of the four cardinal directions. Thus, the DRL controller receives the LIDAR readings as input, and outputs the selected action (direction). The robot's goal is to navigate through a maze towards a target location, without bumping into walls or obstacles.

In constructing a meaningful grammar for this example, we would use our knowledge of the task at hand. For instance, if the robot should only move in certain direction if the closest obstacle in that direction is at least 1 meter away, we would add the predicate $x \geq 1$, where x can be instantiated to each of the LIDAR readings. We might add functions that compute the minimal or maximal LIDAR readings, if it is meaningful that the robot stays close to walls (e.g., to apply the *left-hand rule*), or stay clear of them. Finally, if we know that it is important that the robot identifies entrances to corridors in the maze, we might add a predicate that receives LIDAR readings from three adjacent sensors, and returns true if they represent such an entrance (i.e., the two exterior readings are similar, but the middle one is significantly greater, marking an opening). In this manner, we can construct a meaningful grammar, which would then result in a decision tree with interpretable features.

These simple examples demonstrate the basic guidelines for grammar construction — using logic and domain knowledge to add functions and predicates that express meaningful connections between parts of the input. Naturally, a grammar can be iteratively refined by the user if the need arises.

Training. Once we select a grammar, the next step is to use it to generate a decision tree for predicting whether a state-action pair represents undesirable behavior. Formulas produced by the grammar serve as the features for this tree; and the actual training can be carried out using any standard framework (e.g., *sklearn* [Pedregosa et al. 2011]).

3.2 Inspecting Decision Trees

After creating the decision tree, we propose to manually inspect it in order to try and determine the root cause of the undesirable behavior. This may allow a human expert to adjust and fine-tune the tree, in case the original labeling was inaccurate, or if the state-action pairs did not properly

cover all relevant cases. In addition, this may assist the human expert to better characterize the undesirable behavior, and thus render the system more explainable.

In our use cases, we found it useful to inspect paths of the tree that lead to the same result (desirable or undesirable), and deduce the common behavior described by them. This was easier when the tree was not overly deep (about depth 5), and so the number of paths was not prohibitive. This fact is part of our motivation for selecting adequate grammars, as these allow for more shallow trees without sacrificing precision. We elaborate more on this topic in Sec. 5.

3.3 Reducing Undesirable Behavior

Using the decision tree, we now train the model again, but with the following modification. In each iteration of the training procedure, after obtaining the current state, the selected action and the reward, we evaluate the decision tree on the current state and action. If the tree determines that the ⟨state, action⟩ pair constitutes undesirable behavior, we modify the reward value as described next, and the continue with the training as usual; otherwise, the reward value is unchanged. Fig. 3 depicts the overall flow.

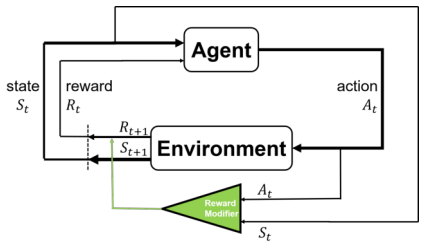


Fig. 3. The modified reinforcement-learning loop.

Reward Modification. When undesirable behavior is detected, we penalize the agent by multiplying its reward r by a *reward-modifier* α, so that $r * \alpha \leq r$. For a positive initial reward, we set $\alpha \in [0, 1]$; and for a negative reward, $\alpha \in [1, \infty]$. Listing 1 depicts the pseudocode for the *get-reward-modifier* function.

```
def get_reward_modifier(tree, state, action):
    outcome = tree.classify(state, action)
    return ( outcome == UNDESIRABLE_BEHAVIOR ) ? REWARD_MODIFIER : 1
```

Listing 1. The *get reward modifier* function.

Penalizing the reward is then performed as shown in Listing. 2.

```
def modify_reward(reward, state, action):
    modifier = get_reward_modifier(state, action)

    if reward >= 0:
        reward = reward * modifier
    else:
        reward = reward * (1/modifier)

    return reward
```

Listing 2. The *modify reward* function

Finally, the modified training loop appears in Listing 3.

```
while step < max_steps:
    # get state
    current_state = get_state()

    # get action
    action = get_action(current_state)

    # get reward
    reward = get_reward(old_state, old_action, current_state)

    # modify the reward
    reward = modify_reward(reward, state, action)

    # update environment
    environment.step(action)

    old_state = current_state
    old_action = action
    step += 1
```

Listing 3. The modified training loop.

3.4 Result Analysis and Hyper-Parameter Adjustment

After re-training the DRL model using the decision tree, we propose to analyze the performance of the new model, and adjust the reward modifier value if needed. Decreasing the reward modifier generally decreases the frequency of undesirable behavior occurring, (as we discuss in Sec. 5); however, decreasing it too much may interfere with the training and hurt performance. Therefore, we propose to train multiple models, using varied reward modifiers, and then choose the one that achieves the best reward values. This process can also be automated in a straightforward way.

4 CASE STUDIES

For evaluation purposes, we applied our approach to three case studies, each time following the steps described in Sec. 3: analyzing state-action pairs, and labeling them as desirable or undesirable behavior; writing an appropriate grammar for the system at hand; using the labeled state-action pairs and grammar to train a decision tree; and then using this tree to retrain the model. We then compared the original and re-trained models, in order to assess the effectiveness of our method. We also examined the decision trees to better understand and characterize the undesirable behavior of each system. Our code and benchmarks are available online [Carmel and Katz 2024]. Through our experiments, we attempted to answer the following research questions (RQs):

(1) **RQ1:** can our approach reduce the agent's undesirable behavior, without significantly harming its performance?
(2) **RQ2:** can our approach be used to better explain the undesirable behavior of the DRL-trained agent to a human?

4.1 Case Study 1: Aurora

Congestion control is the task of balancing the sending rate of packets into a computer network, in order to minimize packet loss and maximize throughout. A main issue in congestion control is that bandwidth is constantly changing, and the sending rate must change accordingly [Eliyahu et al. 2021]. Aurora [Jay et al. 2019] is a congestion controller that uses a DRL agent to govern sending

rates. The Aurora agent is trained with the Proximal Policy Optimization (PPO) algorithm [Schulman et al. 2017], on a fully connected network with two hidden layers. In each time step, it takes three input vectors, which information about the t most recent time steps in the network (in our case, $t = 10$) — the *latency gradient*, indicating an increase or decrease in latency; the *latency ratio*, indicating the ratio between current latency and minimum latency; and the *sending ratio*, indicating the ratio between sent packets and received packets. Consequently, two consecutive Aurora model states $s_1 \rightarrow s_2$ share the same input vectors, except for the oldest entry (in each vector) in s_1, which is dropped and replaced with a fresh entry in s_2. We refer to this relation between consecutive states, which is quite common in computer network systems [Eliyahu et al. 2021], as a *sliding window*. The output of the model is a single value, indicating whether the sending rate should increase (positive value) or decrease (negative value), and by how much.

Grammar. We specify the following, "sliding window" grammar rules, for some Aurora state $s = [s_i, s_{i+1}, ..., s_{i+k}]$:

- **Value:** returns the value Value_i, defined as entry s_i.
- **Diff:** returns the value $\text{Diff}_{i,j}$, defined as $s_i - s_j$.
- **Sign:** returns the value $\text{Sign}_{i,j}$, defined as $Sign(s_i - s_j)$: the sign of the difference between the i'th and j'th values.
- **Average:** returns the value $\text{Average}_{i_1,...,i_m}$, defined as $(\sum_{n \in \{i_1,...,i_m\}} s_n)/m$, i.e. the average of the values with indices $\{i_k | k \in \{0, \ldots, m\}\}$.
- **Action:** returns the value Action, which is the output selected by the agent.

The motivation is that for sliding window inputs, it is useful to inspect how inputs change over time; and that comparing directly adjacent temporal readings is more useful than comparing those that are far apart. We selected the following instantiations of these rules, instantiated on the *sending ratio* part of the model state, to serve as the features for the decision tree:

$$\{\textbf{Value}_i | i \in \{0, \ldots, 9\}\} \cup \{\textbf{Diff}_{i,i+1} | i \in \{0, \ldots, 8\}\} \cup \{\textbf{Diff}_{i,i+2} | i \in \{0, \ldots, 7\}\} \cup$$
$$\{\textbf{Sign}_{i,i+1} | i \in \{0, \ldots, 8\}\} \cup \{\textbf{Sign}_{i,i+2} | i \in \{0, \ldots, 7\}\} \cup \{\textbf{Average}_{i,i+1} | i \in \{0, \ldots, 8\}\} \cup$$
$$\{\textbf{Average}_{i,i+2} | i \in \{0, \ldots, 7\}\} \cup \{\textbf{Average}_{i,i+1,i+2} | i \in \{0, \ldots, 7\}\} \cup \{\textbf{Action}\}$$

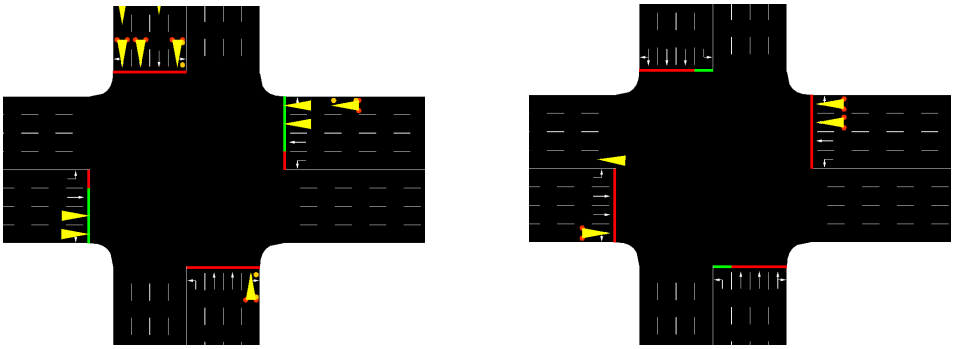
Undesirable Behavior. The undesirable behavior that we target is cases in which network conditions are nearly perfect (low latency, and almost no packet loss), but in which the agent decides to decrease the sending rate. Such actions are clearly not optimal with respect to the goal of maximizing throughput [Eliyahu et al. 2021]. In order to detect instances of this behavior and label it, we looked for traces with very low latency and close-to-optimal sending ratios (less than 1.2 sending ratio in all 10 entries).

4.2 Case Study 2: Traffic Control

In *Traffic Control* [Vidali 2019], a DRL agent manages a road intersection. The agent is a Deep Q-Learning agent [Van Hasselt et al. 2016], trained using a fully connected network with five hidden layers. At each time step, the agent needs to determine what the traffic lights at the intersection should show, in order to maximize the intersection's throughput.

The intersection has eight lanes: (i) West to East (W2E); (ii) West to Turn Left (W2TL); (iii) East to West (E2W); (iv) East to Turn Left (E2TL); (v) North to South (N2S); (vi) North to Turn Left (N2TL); (vii) South to North (S2W); and (viii) South to Turn Left (S2TL). Each lane is divided into 10 segments in different distances from the intersection. The agent's input is 80 Boolean flags, one per segment, indicating whether or not cars are currently in that segment. The agent's output action determines which of the lanes get a green light for this time step, and which do not, according

to one of 4 possible configurations: (i) North and South (NS), which opens up the N2S and S2N lanes; (ii) North and South Turn Left (NSL), which opens up the N2TL and S2TL lanes; (iii) East and West (EW), which opens up the E2W and W2E lanes; or (iv) East and West Turn Left (EWL), which opens up the E2TL and W2TL lanes. An illustration appears in Fig. 4.



(a) An EW traffic light configuration. (b) An NSL traffic light configuration.

Fig. 4. The intersection for the *Traffic Control* system.

Grammar. In this case study, it is convenient to consider each of the traffic lanes separately. Doing that, we observe that each lane has the "sliding window" structure, discussed previously; i.e., in each lane, each car iteratively moves forward towards the intersection (or away from it), and two consecutive observations of the lane are tightly linked. However, unlike in Aurora, here each car can traverse several segments in a single time step, and so the sliding window can shift by multiple entries (as opposed to a single entry). To account for this, we adjust our sliding window grammar, and compute averages and differences on the individual sliding windows (corresponding to individual lanes). Further, to simplify the decision tree, we remove some of the features that are less likely to be meaningful — namely, those that concern lane segments that are very far away from the intersection, and those that concern sets of segments that are far apart from each other.

The resulting grammar that we use is thus:

- **Value:** returns the value $\text{Value}_{lane,i}$, defined as $s_{lane,i}$ in a specific lane.
- **Diff:** returns the value $\text{Diff}_{lane,i,j}$, defined as $s_{lane,i} - s_{lane,j}$ in a specific lane.
- **Average:** returns the value $\text{Average}_{lane,i_1,\ldots,i_m}$, defined as $(\sum_{n\in\{i_1,\ldots,i_m\}} s_{lane,n})/m$, i.e. the average of the values in the set of indices $\{i_k | k \in \{0,\ldots,m\}\}$ in a specific lane.
- **IsAction:** returns the value IsAction_d, defined as $action == d$.

We use the Diff rule only for segments that are adjacent, and include the Average rule only for consecutive segments adjacent to the intersection. The instantiations of the grammar rules in this case are thus:

$$\{\textbf{Value}_{lane,i} \mid i \in \{0,\ldots,9\}, lane \in \text{LANES}\} \cup \{\textbf{Diff}_{lane,i,i+1} \mid i \in \{0,\ldots,9\}, lane \in \text{LANES}\} \cup$$
$$\{\textbf{Average}_{lane,0,\ldots,m} \mid m \in \{1,\ldots,7\}, lane \in \text{LANES}\} \cup \{\textbf{Action}_d \mid d \in \text{ACTIONS}\} \cup$$

where $\text{LANES} = \{N2S, S2N, E2W, W2E, N2TL, S2TL, E2TL, W2TL\}$ and $\text{ACTIONS} = \{NS, EW, NSL, EWL\}$.

Undesirable Behavior. We target cases where some lanes are empty, but others are "jammed" — that is, the two segments closest to the intersection are filled with cars — but in which the agent

does assigns the green light to empty lanes instead of the crowded ones. Clearly, this choice of action is not optimal.

4.3 Case Study 3: Snake

In this case study, we study a DRL agent trained to play the game *Snake*. The agent is again a Deep Q-Learning agent [Van Hasselt et al. 2016], trained using a fully connected network with two hidden layers. In this game, the agent controls a snake that is slithering around on a board. In each time step, the snake's head can move in each cardinal direction (except the one which is the opposite of which it is currently facing); and the snake's body always follows its head, along the exact same trajectory that the head took. Each time, an apple will appear on the board, and the snake's goal is to eat as many apples as possible — without colliding with its own tail or with a wall. Each apple collected increases the snake's length by one, making it more challenging to maneuver without colliding. As part of our case study, we built on top of a publicly available implementation of the game [Harder 2022].

The input to the Snake agent is a binary array with 12 entries, with the values depicted in Fig. 5. These entries describe the location of the apple with respect to the snake; the location of obstacles in the snake's vicinity; and the snake's direction. The agent then chooses one from four possible actions: move UP, move RIGHT, move DOWN or move LEFT. Finally, the reward is computed as described in Fig. 5.

State	
Apple is above the snake	0 or 1
Apple is on the right of the snake	0 or 1
Apple is below the snake	0 or 1
Apple is on the left of the snake	0 or 1
Obstacle directly above the snake	0 or 1
Obstacle directly on the right	0 or 1
Obstacle directly below the snake	0 or 1
Obstacle directly on the left	0 or 1
Snake direction == up	0 or 1
Snake direction == right	0 or 1
Snake direction == down	0 or 1
Snake direction == left	0 or 1

Rewards	
Snake eats an apple	10
Snake comes closer to the apple	1
Snake goes away from the apple	-1
Snake dies (hits his body or the wall)	-100

Fig. 5. On the left: the state of Snake. On the right: the rewards at each time step of Snake.

Grammar. Unlike in the previous use cases, the input to the Snake agent does not include a sliding window component. Instead, it includes three sections, which are conceptually separate: the "APL" part, describing the apple's location; the "DIR" part, indicating the snake's direction; and the "OBS" part, indicating any nearby obstacles. We add another section, which is the "ACT" part, describing the action selected by the agent. Each of these parts is binary array of size 4, with each entry corresponding to a direction: up, right, down or left. For example, state $s = [1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1]$ is interpreted as

$$APL = [1, 0, 0, 1], \quad DIR = [0, 1, 0, 0], \quad OBS = [1, 1, 0, 0], \quad ACT = [0, 0, 0, 1]$$

which means that the apple is in the direction "UP-LEFT", the snake is facing "DOWN", there is an obstacle in the direction "UP-RIGHT", and the action selected was "LEFT". We refer to this kind of input as an *equal-size partitions* input, and specify the following grammar rules, for some Snake state $s = [APL, DIR, OBS, ACT]$:

- **Value:** returns the value $\text{Value}_{part,i}$, defined as $s_{part,i}$.
- **IsAction:** returns the value IsAction_d, defined as $s_{ACT,d} == 1$

- **IsEqual:** returns the value $\text{IsEqual}_{part_1,part_2,i}$, defined as $s_{part_1,i} == s_{part_2,i}$.

where $part, part_1, part_2 \in \{APL, DIR, OBS, ACT\}$. The instantiations of the grammar rules in this case are:

- $\{\textbf{Value}_{part,i} \mid i \in \{0,1,2,3\}, part \in \text{PARTS}\}$
- $\{\textbf{IsAction}_d \mid d \in \text{DIRECTIONS}\}$
- $\{\textbf{IsEqual}_{part1,part2,i} \mid i \in \{0,1,2,3\}, part1, part2 \in \text{PARTS}\}$

where PARTS = {APL, OBS, DIR, ACT} and DIRECTIONS = {UP, DOWN, LEFT, RIGHT}. For example, $IsEqual_{APL,OBS,2}$ evaluates to true when the apple and an obstacle are to the snake's right, whereas $IsEqual_{OBS,DIR,3}$ evaluates to true when there is an obstacle directly below the snake, and its direction is also down.

The motivation for these rules is that in an "equal-size partitions" system, the inputs in the same part are less likely to have meaningful connections, whereas inputs in different parts but which share the same index, may present meaningful connections. Conversely, the sliding window rules used in Aurora and Traffic Control are mostly irrelevant in this case. For example, the $Average_{i,j}$ rule would compute an average over Boolean values that are most likely independent. Instead, the proposed grammar leverages the symmetry between the four parts of the input state, and allows inspecting the relative directions of the snake, apple and obstacles.

Undesirable Behavior. When examining state-action pairs, we noticed cases where the snake would already be moving towards the apple, but the agent would suddenly switch the snake to move in another direction — even though there were no obstacles nearby. This behavior is clearly undesirable.

5 EXPERIMENTS AND RESULTS

5.1 Experiment 1: Reduction of Undesirable Behavior

In this experiment, we set out to answer RQ 1. To do so, we apply our methodology to each of the case studies: we label state-action pairs as desirable or undesirable (according to the criteria in Sec. 4); train the resulting decision trees; and then use these trees to retrain the agents. We train models with different reward modifier values (a model trained with a reward modifier 1.0 is just the original model), and compare their performance. Our hypothesis is that even reward modifiers close to 1 should be sufficient to produce a significant reduction in the frequency of undesirable behavior selected by the agent. To measure this, we compute the reward obtained by the modified model, and compare it to that of the original. Further, we compute the ratio of undesirable behaviors among all behaviors, $\frac{\text{UNDESIRABLE}}{\text{DESIRABLE+UNDESIRABLE}}$. Additionally, we measure the time overhead caused by our method as part of the DRL training procedure (assuming the modified model is trained to achieve a reward similar to that of the original).

Aurora. The results of applying our method to the Aurora use case are depicted in Fig. 6. We trained 5 models for each of the reward modifier values $\{1, 0.8, 0.6, 0.5, 0.4, 0.2, 0.1, 0.05, 0.01\}$. The plot on the left shows the average running-average reward values obtained by the models, for each reward modifier value; and the plot of the right shows the average running-average undesirable ratio. For training, we used approximately 1000 state-action pairs of desirable behavior and undesirable behavior, recorded during the original agent's training. We observe that the various agents tend to converge to a steady running-average reward level after approximately 500 test steps. Once a model has converged, we calculated the average reward over the next 1500 test steps, and these make up the plot on the left.

We observe that, generally, training with small reward modifiers tends to make agents achieve higher rewards. Also, if we examine the training rewards, we see that the models tend to converge

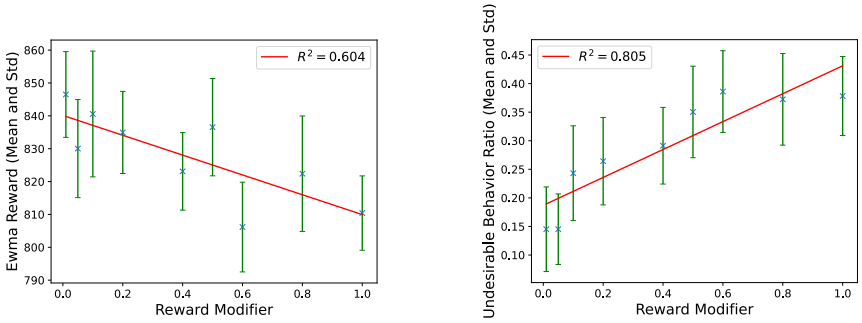


Fig. 6. Aurora: on the left-hand side, the average reward values per reward modifiers value. On the right-hand side, the average undesirable behavior ratio per reward modifier value.

after a similar number of steps. However, our approach does increase training time by as much as 38%, as each training iteration takes longer to carry out. As part of our future work we plan to reduce this overhead, by optimizing our implementation.

The results in Fig. 6 indicate a direct correlation between lower reward modifier values and the scarcity of undesirable behavior. This is expected, because lower modifier values imply a harsher penalty to the agent for undesirable behavior. We were able to decrease the undesirable behavior ratio by around 60%, which is significant; although we were not able to completely remove the undesirable behavior, presumably because it was quite common for the original model.

Overall, we conclude that our framework was able to achieve its objectives for the Aurora case study: the undesirable behavior was significantly reduced, without degrading performance — and even improving it in some cases.

Traffic Control. Next, we performed a similar experiment for the Traffic Control use case. We used 5 different reward modifier values, $\{1, 0.75, 0.5, 0.25, 0.1\}$, and trained 3 different models for each of them (with 100 training and testing iterations per model). The left-hand side of Fig. 7 depicts the running-average rewards we obtained. For training the tree, we used approximately 74,000 state-action pairs. This time, the training time overhead was around 11% for a single episode. We also observed a difference in the number of iterations required for convergence: it took approximately 30-40 additional iterations for models with low reward-modifier values to converge compared to those with higher reward-modifier values. We observe that in this case, applying our approach also resulted in increasing the agent's overall reward. This is unsurprising, as the behavior we labeled as undesirable was indeed counter productive to the agent's goals.

Examining the ratio of undesirable behavior, $\frac{UNDESIRABLE}{DESIRABLE+UNDESIRABLE}$, as a function of the reward modifier (right-hand side of Fig. 7), we again observe a direct correlation between lower reward modifier values and the scarcity of undesirable behavior. We were able to decrease the undesirable behavior ratio by around 89.4%, which is highly significant, and which implies that the undesirable behavior barely occurs anymore.

Overall, we conclude that our framework was able to achieve its objectives for the Traffic Control case study: the undesirable behavior was significantly reduced, without degrading performance — and even improving it.

Snake. Finally, we applied our method to the Snake case study, using 5 different reward modifier values — $\{1, 0.75, 0.5, 0.25, 0.1\}$. We trained 4 different models for each reward modifier value, and then performed 100 games of Snake for each of these models. For training, we used approximately

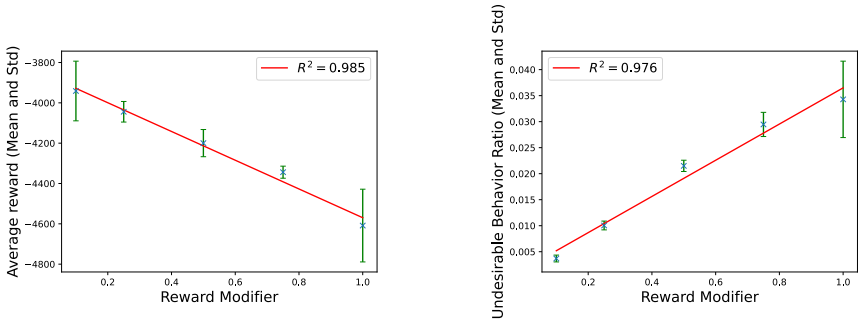


Fig. 7. Traffic Control: on the left-hand side, the average reward value per reward modifier value. On the right-hand side, the average undesirable behavior ratio per reward modifier value.

16,000 state-action pairs. These state-action pairs were recorded while training regular models. The overhead in training was negligible ($\tilde{1}$%).

The left-hand side of Fig. 8 demonstrates that in this case, applying our approach actually decreased the average reward by approximately 1.5%. This implies that if we penalize the agent too harshly (via a very small reward modifier value), performance is actually slightly hurt. This outlines the trade-off between performance and undesirable behavior reduction.

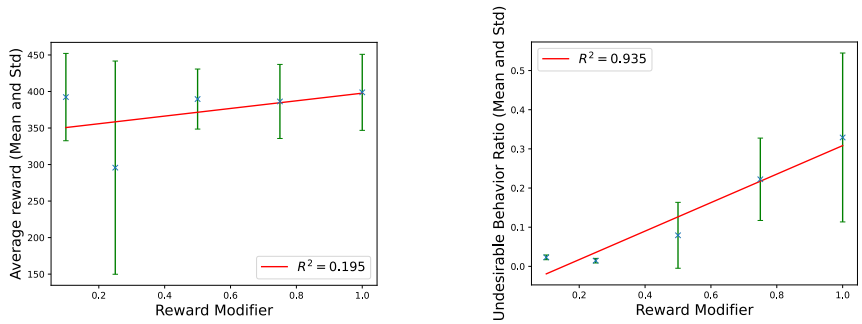


Fig. 8. Snake: on the left-hand side, the average reward value per reward modifier value. On the right-hand side, the average undesirable behavior ratio per reward modifier value.

Examining the ratio of undesirable behavior as a function of the reward modifier used (right-hand side of Fig. 8), we once again observe a direct correlation between lower reward modifier values and the scarcity of undesirable behavior. We were able to decrease the undesirable behavior ratio by around 93%, which is highly significant, and implies that undesirable behavior barely occurs anymore.

Overall, we conclude that our framework was able to achieve its main objective in the Snake Control case study.

Experiment 1: Discussion. First and foremost, we observe that the our method caused the undesirable behavior ratio to significantly decrease in all three case studies — by 60%, 89% and 93%. This implies that our approach indeed achieves its main objective, which is to reduce undesirable behavior.

Next, we observe that the average rewards obtained by the modified agents increased in two out of three cases, by 3% and 16%; and that in the third case, the average reward decreased by a slight

1.5%. This variability in reward values highlights the fact that the original reward function may or may not penalize the behavior marked as undesirable (which is perhaps not surprising, given that crafting reward functions is considered very difficult [Li 2017]). In the first two cases, by penalizing undesirable behavior during training, our method caused the trained agent to converge to a policy that achieved a higher average reward — and we conclude that in these cases, avoiding the specified behaviors at least partly coincides with the reward function. In the third case, we conclude that the reward function actually encourages some of the undesirable behavior, and so by forcing the agent to avoid it, we caused it to achieve a lower average reward value. We regard all three cases as instances where the method worked precisely as expected; although the third case should perhaps prompt the engineers to reevaluate the reward function originally selected. This effort could be supported by explainability afforded by our trained decision trees, as studied in Experiment 2.

Our method incurred a training overhead in all three case studies, ranging from fairly negligible (1% and 11%) to medium (38%). This is due to the need to evaluate the decision tree on the state-action pair in each step during training. The variability between case studies appears to be caused by different efficiencies or inefficiencies among the underlying implementations (each case study was implemented in a different environment).

Finally, we conclude that our approach was indeed successful in significantly reducing undesirable behavior in all cases, sometimes even improving the overall average reward along the way, but at the cost of incurring slight to medium training overhead.

5.2 Experiment 2: Explainability

Another advantage of our approach is that it can assist in *explaining* the undesirable behavior that it attempts to minimize. This is performed by presenting the decision tree, which is already produced as part of our approach, for manual inspection by the engineers. In this experiment, we set out to answer RQ 2, and determine whether our approach is indeed useful in explaining undesirable behavior. As we later see, the answer is affirmative. We point out that there is a certain trade-off between our two RQs: a deeper decision tree will likely result in higher model accuracy, but will be more difficult for humans to manually parse; and vice versa. Below, we analyze this trade-off using our case studies.

Aurora. In Aurora, the behavior we labeled as undesirable is when the network conditions are generally good (low latency, low packet loss), but the agent chooses to decrease the sending rate. We trained a tree over state-action pairs of desirable and undesirable behavior, which reached over 99.5% accuracy with a depth of 9. When inspecting this tree, we observed 2 significant paths:

- $ACT > 0 \rightarrow$ **Desirable**. This is a significant path, because it implies that the selected action pays a key role in classification — if the action is greater than 0, the behavior is always desirable. This is consistent with our targeting of cases where the agent would reduce the sending rate, despite network conditions being good.
- $((Value_9 < 1.195) \wedge (Value_8 < 1.195) \wedge (Value_6 < 1.195) \wedge (Value_2 < 1.195) \wedge (AVG_{1,2,3} < 1.2)) \leftrightarrow$ **Undesirable**. In this path, if any of the listed clauses are False, then the behavior is desirable; and if all of them are True, then the behavior is undesirable. This implies that if the values in places 9, 8, 6 or 2 (representing past sending rates) are smaller than 1.195, and if the average of indices 1,2,3 is smaller than 1.2, then the behavior is undesirable. We deduce from here that the tree learned that values 1.2 and 1.195 are significant thresholds, which separate good network conditions (where the sending rate should not be decreased) from bad ones. This is again consistent with our original labeling of traces.

Combining the information these two paths, we obtain a general and rigorous specification of the undesirable behavior — if the agent chooses to decrease the sending rate, and also the specified

values are below the 1.195 threshold, then the behavior is undesirable. These values could then be inspected and fine-tuned by a domain expert.

Traffic Control. In Traffic Control, we labeled state-action pairs as undesirable when the agent failed to take the obvious action needed to resolve traffic jams. Our tree achieved over 98% balanced accuracy, with precision of 91% and recall of 98% at the depth of 10. Because a tree of depth 10 is difficult to parse manually, we began by focusing on its top 3 layers, where the most dominant features reside. In our case, these were the *Average* and *Action* features. Two interesting paths that we identified are:

- E2W has cars, Action is EW, N2S has fewer than 2 segments filled with cars near the intersection → **Desirable**.
- E2W has no cars, N2S has at least one segment filled with cars, Action is not NS → **Undesirable**.

These tree paths highlight a connection between the first segments of a lane being populated by cars, and the undesirability of not assigning it a green light; and indeed, similar connections appear also in other, deeper parts of the tree. For example, when inspecting deeper layers of the tree (up to depth 5), we observed an additional interesting path; see the diagram in Fig. 9. The splits along this path examine various lanes in sequence, each time checking whether a lane is populated with cars, at least in its first segments. If this is the case, but the selected action does not assign that lane a green light, this is generally classified as *undesirable* behavior; and otherwise, the behavior is *desirable*. This already provides a clear, even if partial, formulation of the property at hand.

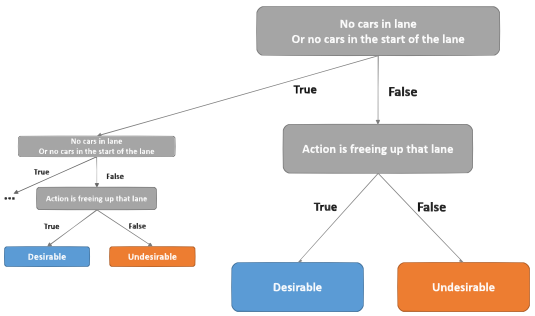


Fig. 9. Traffic decision tree: diagram of a specific path.

We also carried out an additional experiment, in order to assess the effect of different grammars on the interpretability of the resulting tree. Specifically, we took the grammar used for the Snake case study, and used it to train a tree for Traffic Control. This was achieved by adjusting the functions used as grammar rules in "Snake" to receive the input vectors specified in "Traffic Control". The new features were the action selected by the agent, along with comparisons between corresponding segments across pairs of lanes. At depth 10, the resulting tree reached precision of 87% and recall of 97.7%, with balanced accuracy of 97.8%, which is slightly worse than the original tree, and was generally more difficult to interpret. For example, if we inspect the top 3 layers of the tree, we can indeed infer that the first two segments in a lane are crucial for the outcome of the classification, as the following path demonstrates:

- Either N2S or E2W, but not both, have cars in their second segment, ACT is NS, and E2W has cars in its second segment → **Undesirable**.

But the conclusion is not as concrete as it was when inspecting the earlier tree. Ultimately, when inspecting deeper trees, we can come to similar (but less general) conclusions as with the first tree, but this process was more difficult to carry out and not as intuitive — indicating the importance of picking a grammar that is appropriate for the system at hand.

Snake. In Snake, we labeled state-action pairs as undesirable when the agent made the snake turn away from an apple. We now wish to see whether we can better characterize this undesirable behavior, using the tree. For this, we inspect a sub-tree with depth 3, which reaches 94.8% balanced accuracy, with precision of 76% and recall of over 98% (the tree used for the re-training is of depth 8 and reaches 99.92% balanced accuray, with precision and recall of over 98%).

Analyzing the paths of this tree, we observe that the dominant features are "IsEqual" features that compare DIR with ACT or with APL. We can deduce from this that DIR is the most important category in determining undesirable behavior. Two of the relevant tree paths are:

- DIR and ACT are equal in the direction "LEFT", DIR and ACT are equal in the direction "RIGHT", DIR and ACT are equal in the direction "UP" → **Desirable**. Because we know ACT and DIR are binary vectors of size 4, and that each vector has exactly one "1" entry and the rest are "0", this path implies that if DIR == ACT, then the behavior is desirable. This is indeed a key characteristic of the undesirable behavior — i.e., that DIR has to be different from ACT for the behavior to be undesirable.
- DIR and ACT are not equal in the direction "LEFT", DIR and APL are equal in the direction "LEFT" → mostly **Undesirable**. This path indicates that if DIR and ACT are not equal, but DIR and APL are equal, this is mostly undesirable.

By looking at this fairly shallow tree, we can make deductions that assist us in formalizing the undesirable behavior. By inspecting deeper trees, we can observe that the classifier is indeed attempting to figure out the indices in which the DIR, APL, and ACT are equal, in order to make its classification. We observe that OBS has little effect here, but that it does appear in the deeper levels of the tree — presumably because the cases where an obstacle is nearby are few and far between.

Next, if we compare this tree to a tree produced using the "Traffic" grammar, we observe that although both grammars result in trees with comparable precision and recall (above 98% and 99%, respectively, for both trees), the "Traffic" grammar requires a tree of depth 13 to achieve this, compared to depth 6 for the "Snake" grammar — which hampers explainability. Also, manually inspecting the tree produced using the "Traffic" grammar proved much more difficult, because the features were less relevant — for example, the first split is determined by whether the Average of the "UP" and "RIGHT" entries in "APL" equals 0, which is a convoluted way of asking whether the direction of APL is either "UP" or "RIGHT". A deeper inspection revealed that this second tree ended up making similar decisions to the first, but this took a great deal of effort. These results again highlight the importance of picking an appropriate grammar.

5.3 Comparison to the State of the Art

Because our approach can be integrated with any black-box reward reshaping method, the most relevant comparison is between our approach and other *Safe-RL methods*, i.e., methods aimed at reducing undesirable behavior in RL-based models. Here, we ran into the following difficulties: (i) one of the main advantages of our approach is that we do not assume an a-priori, rigorous characterization of the undesirable behavior we want to reduce, whereas state-of-the-art techniques require such a characterization; and (ii) the goal of existing Safe-RL techniques is to completely eliminate undesirable behavior, whereas our approach allows fine-tuning the trade-off between eliminating such behavior and maintaining high performance. These two issues prevented us from conducting a meaningful comparison to many of the existing techniques.

For these reasons, the most relevant existing technique to which we compared our proposed approach is VIPER, proposed by Bastani et al. [Bastani et al. 2018]. There, the authors extract a decision tree policy from a pre-trained DNN. This tree, which is assumed to mimic the behavior of the DNN fairly well, is then used for verification purposes. (We stress again that VIPER does not apply reward reshaping, and that it does not solve the exact same problem as our approach.)

To compare the approaches, we focused on the Traffic Control use-case, and performed a comparison between three models: (i) the original Traffic Control model; (ii) the decision tree produced by VIPER; and (iii) the model produced by our approach, with the reward modifier arbitrarily set to 0.1, referred to as UBR (for undesirable-behavior-reduced). For each of these models, we calculated the undesirable behavior ratio and the final reward over 100 runs. Figure 10 depicts a density plot and a histogram plot comparing the results.

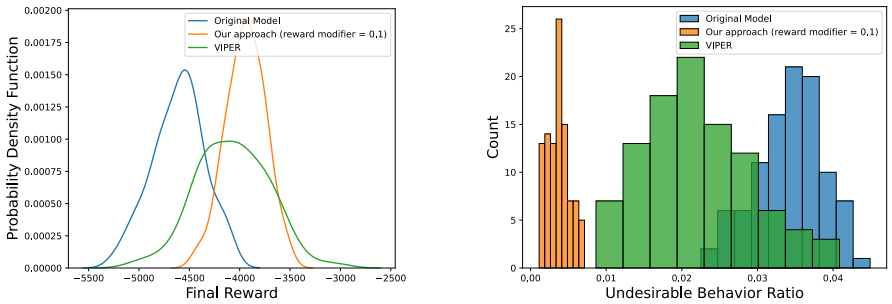


Fig. 10. Traffic Control: Comparing our approach and VIPER.

As Fig. 10 shows, in terms of the reward values achieved the original model is outperformed by both our model and the VIPER model; whereas VIPER performs comparably, and even slightly better, than our approach. In terms of undesirable behavior, VIPER significantly outperforms the original model, and our model significantly outperforms VIPER. These results are to be expected, as our approach reduces the frequency of undesirable behavior, but achieves this by altering the training process, possibly reducing the achieved reward.

Apart from these criteria, we also set out to evaluate the explainability afforded by our approach, compared to VIPER. The VIPER tree had 3139 nodes, whereas the UBR tree had only 511. Further, the VIPER tree is trained on the state space (here, 80 binary values), as opposed to our tree that is trained using a user-defined grammar, and consequently its tree nodes were not as uninformative. Inspecting the different paths of the VIPER tree for depth 3, we observed the following:

- W2E[0] = 0 ⟶ E2W[0] = 0 ⟶ E2TL[0] = 0 ⟶ EWL
- W2E[0] = 0 ⟶ E2W[0] = 0 ⟶ E2TL[0] = 1 ⟶ NSL
- W2E[0] = 0 ⟶ E2W[0] = 1 ⟶ E2TL[0] = 0 ⟶ NS
- W2E[0] = 0 ⟶ E2W[0] = 1 ⟶ E2TL[0] = 1 ⟶ NSL
- W2E[0] = 1 ⟶ E2W[0] = 0 ⟶ S2TL[0] = 0 ⟶ EW
- W2E[0] = 1 ⟶ E2W[0] = 0 ⟶ S2TL[0] = 1 ⟶ NSL
- W2E[0] = 1 ⟶ E2W[0] = 1 ⟶ S2N[1] = 0 ⟶ EW
- W2E[0] = 1 ⟶ E2W[0] = 1 ⟶ S2N[1] = 1 ⟶ NS

Clearly, these paths indicate that the first indices of each lane are the most dominant, and that the selected action roughly corresponds to the lane in which there are cars. However, by just inspecting these paths we cannot fully explain the selected actions — especially when considering that the state-space is rich, and that selecting an action based strictly on which lanes have cars is suboptimal.

Delving deeper into the tree affords a better understanding, but this required significantly more work than with the UBR tree.

Next, we repeated the experiment with the Snake case-study, comparing the original model (using an average of 4 models), our undesirable-behavior-reduced (UBR) model (trained with reward modifier 0.1, using an average of 4 models), and the VIPER model. We calculated the undesirable behavior ratio and the final reward for 100 runs for all models. The results appear in Fig. 11.

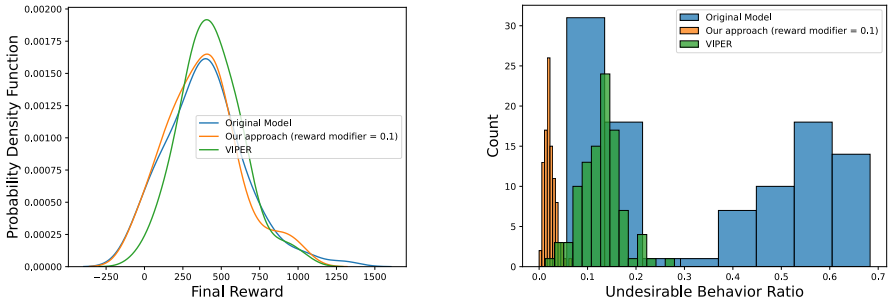


Fig. 11. Snake: Comparing our approach and VIPER.

Fig. 11 shows that all 3 models perform similarly, with VIPER very slightly outperforming the original model, and our UBR model slightly outperforming both. Also, VIPER has a slightly lower undesirability ratio than the original model, and the UBR model has the lowest undesirability ratio.

Comparing the explainability of the VIPER and UBR trees, we again observe that UBR affords better explainability. Specifically, the UBR tree has fewer nodes — 211, compared to 241 in the VIPER tree. Inspecting the paths of the VIPER tree up to depth 3, we get:

- DIRECTION is not RIGHT → DIRECTION is not LEFT → APPLE is not DOWN → LEFT
- DIRECTION is not RIGHT → DIRECTION is not LEFT → APPLE is DOWN → LEFT
- DIRECTION is not RIGHT → DIRECTION is LEFT → APPLE is not DOWN → DOWN
- DIRECTION is not RIGHT → DIRECTION is LEFT → APPLE is DOWN → LEFT
- DIRECTION is RIGHT → APPLE is not RIGHT → APPLE is not UP → LEFT
- DIRECTION is RIGHT → APPLE is not RIGHT → APPLE is UP → UP
- DIRECTION is RIGHT → APPLE is RIGHT → OBSTACLE is not RIGHT → RIGHT
- DIRECTION is RIGHT → APPLE is RIGHT → OBSTACLE is RIGHT → LEFT

We observe that the most dominant features are DIRECTION and APPLE, similarly to the UBR tree. However, other than the paths "DIRECTION is RIGHT → APPLE is RIGHT → OBSTACLE is not RIGHT → RIGHT"", and "DIRECTION is RIGHT → APPLE is not RIGHT → APPLE is UP → UP", in which the action is to move towards the apple, it is not intuitively clear how decisions are made. It also seems that the "default" choice is to turn LEFT, even though this can be suboptimal. Additional behaviors of the tree can be inferred via a deeper inspection, but this is not intuitive.

We conclude that in both case-studies, we were able to train a VIPER model that adequately mimics the original model — but also that in both cases, the UBR approach achieved higher rewards than VIPER, and afforded superior explainability. This is perhaps not surprising, as VIPER is geared towards verification, and not towards interpretability.

6 RELATED WORK

Reducing undesirable behavior in deep reinforcement learning models has been studied extensively [Gu et al. 2023]. Tessler et al. [Tessler et al. 2018] propose an approach that uses an actor-critic

method to penalize the agent's reward, in order to change the policy being learned into one that satisfies various constraints. Zhang and Guo [Zhang and Guo 2022] suggest a risk-preventative training method, which utilizes a classifier that predicts the risk of traces becoming unsafe, and penalizes the reward accordingly — in order to prevent risky behavior. Dalal et al. [Dalal et al. 2018] propose to add a safety layer that corrects selected actions into the closest action that does not lead to a safety violation, and in that way prevent the agent from ever reaching unsafe states during training and after deployment. Our approach is similar to the aforementioned ones, in that it reduces undesirable behaviors prevalent in the model; but it is less restrictive. For example, we do not attempt to eliminate the undesirable behavior entirely, instead allowing the user to decide on the trade-off between reducing it and reducing performance (as we saw, e.g., in Case Study 4.3). Also, existing approaches often assume that the undesirable behavior is well specified, as a hard safety constraint. In our approach, we circumvent this requirement, and only assume that a human engineer flags undesirable behavior, without necessarily understanding the underlying causes.

Thomas et al. [Thomas et al. 2019] tackle a similar goal but from a different angle, and design a machine-learning framework that uses a "Seldonian optimization" approach in order to prevent undesirable behavior specified by the user. Unlike our framework, this approach does not afford improved explainability, but it does demonstrate the usefulness of a user-provided flagging of undesirable behavior — which is encouraging evidence of the potential of this line of research.

Bastani et al. [Bastani et al. 2018] train a decision tree based on the DRL's labeling, and try to replicate the DRL policy with the decision tree. This allows much easier verification, as a tree is much simpler to verify than a DNN. We explored the similarities and differences to our approach in the previous section.

7 CONCLUSION AND FUTURE WORK

The increasing pervasiveness of DRL poses new challenges when it comes to safety and explainability. We presented here an approach aimed at rendering DRL models safer, by identifying undesired behaviors in their traces and then using this information, via reward reshaping, to improve their training. In addition to increased safety, our techniques also serve to increase the explainability of these systems. Our evaluation on three diverse and significant case studies indicates the great potential of this line of work. The main novelty of our approach is that, in contrast to Safe-RL methods, it does not assume a rigorous, a-priori characterization of undesirable behavior. Instead, we learn the undesirable behavior from user input, and then retrain the model in order to reduce its undesirable behavior. The main contribution of this work is thus the proposed framework, which is straightforward to implement on top of existing DRL frameworks, and offers a simple way for reducing undesirable behavior. Although applying our techniques requires input from a human-in-the-loop, the cognitive load that this incurs is fairly small.

Moving forward, we plan to generalize our method to entire traces of undesirable/desirable behavior, as opposed to individual state-action pairs. This would allow our learned decision trees to affect the reward function in more subtle ways, and hopefully result in safer models. Another angle we plan to pursue is to investigate the effect of allowing negative reward modifiers as part of our approach. Using negative reward modifiers would punish undesirable behavior more harshly, but may slow down the training process because of convergence issues. Finally, we intend to optimize the performance of our proof-of-concept implementation, in order to reduce the computational overhead incurred by our approach.

DATA AVAILABILITY

Our code and benchmarks are available online [Carmel and Katz 2024].

REFERENCES

D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané. 2016. Concrete Problems in AI Safety. Technical Report. https://arxiv.org/abs/1606.06565.

K. Arulkumaran, M. Deisenroth, M. Brundage, and A. Bharath. 2017. Deep Reinforcement Learning: A Brief Survey. *IEEE Signal Processing Magazine* 34, 6 (2017), 26–38.

O. Bastani, Y. Pu, and A. Solar-Lezama. 2018. Verifiable Reinforcement Learning via Policy Extraction. In *Proc. 32nd Conf. on Neural Information Processing Systems (NeurIPS)*.

S. Bhatt. 2019. Reinforcement Learning 101. https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292

M. Cai, M. Mann, Z. Serlin, K. Leahy, and C.-I. Vasile. 2023. Learning Minimally-Violating Continuous Control for Infeasible Linear Temporal Logic Specifications. Technical Report. https://arxiv.org/abs/2210.01162.

O. M. Carmel and G. Katz. 2024. On Reducing Undesirable Behavior in Deep-Reinforcement-Learning-Based Software: Code and Benchmarks. https://github.com/ophircarmel/Reducing-Undesirable-Behaviour

E. Clarke, T. Henzinger, H. Veith, and R. Bloem. 2018. *Handbook of Model Checking*. Springer.

G. Dalal, K. Dvijotham, M. Vecerik, T. Hester, C. Paduraru, and Y. Tassa. 2018. Safe Exploration in Continuous Action Spaces. Technical Report. https://arxiv.org/abs/1801.08757.

T. Eliyahu, Y. Kazak, G. Katz, and M. Schapira. 2021. Verifying Learning-Augmented Systems. In *Proc. Conf. of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*. 305–318.

G. Gu, L. Yang, Y. Du, G. Chen, F. Walter, J. Wang, Y. Yang, and A. Knoll. 2023. A Review of Safe Reinforcement Learning: Methods, Theory and Applications. Technical Report. https://arxiv.org/abs/2205.10330.

H. Harder. 2022. Snake Played by a Deep Reinforcement Learning Agent. https://towardsdatascience.com/snake-played-by-a-deep-reinforcement-learning-agent-53f2c4331d36

Y. Hu, W. Wang, H. Jia, Y. Wang, Y. Chen, J. Hao, F. Wu, and C. Fan. 2020. Learning to Utilize Shaping Rewards: A New Approach of Reward Shaping. Technical Report. https://arxiv.org/abs/2011.02669.

N. Jay, N. Rotman, B. Godfrey, M. Schapira, and A. Tamar. 2019. A Deep Reinforcement Learning Perspective on Internet Congestion Control. In *Proc. Int. Conf. on Machine Learning (ICML)*. 3050–3059.

B. Johnson, Y. Brun, and A. Meliou. 2020. Causal Testing: Understanding Defects' Root Causes. In *Proc. 42nd Int. Conf. on Software Engineering (ICSE)*. 87–99.

D. Karger, S. Oh, and D. Shah. 2013. Efficient Crowdsourcing for Multi-Class Labeling. In *Proc. ACM Int. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 81–92.

G. Katz, C. Barrett, D. Dill, K. Julian, and M. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *Proc. 29th Int. Conf. on Computer Aided Verification (CAV)*. 97–117.

Y. Kazak, C. Barrett, G. Katz, and M. Schapira. 2019. Verifying Deep-RL-Driven Systems. In *Proc. 1st ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*. 83–89.

C. Kingsford and S. Salzberg. 2008. What are Decision Trees? *Nature Biotechnology* 26, 9 (2008), 1011–1013.

G. Lample and D. Chaplot. 2017. Playing FPS Games with Deep Reinforcement Learning. In *Proc. 31st AAAI Conf. on Artificial Intelligence (AAAI)*.

Y. LeCun, Y. Bengio, and G. Hinton. 2015. Deep Learning. *Nature* 521, 7553 (2015), 436–444.

M. Leszak, D. Perry, and D. Stoll. 2000. A Case Study in Root Cause Defect Analysis. In *Proc. 22nd Int. Conf. on Software Engineering (ICSE)*. 428–437.

Y. Li. 2017. Deep Reinforcement Learning: An Overview. (2017). Technical Report. https://arxiv.org/abs/1701.07274.

Q. Liu and Y. Wu. 2012. *Supervised Learning*. Springer.

E. Marchesini and A. Farinelli. 2020. Discrete Deep Reinforcement Learning for Mapless Navigation. In *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*. 10688–10694.

V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, and et al. 2015. Human-Level Control through Deep Reinforcement Learning. , 529–533 pages.

S. Mousavi, M. Schukat, and E. Howley. 2018. Deep Reinforcement Learning: An Overview. In *Proc. SAI Intelligent Systems Conf. (IntelliSys)*. 426–440.

A. Ng, D. Harada, and S. Russell. 1999. Policy Invariance under Reward Transformations: Theory and Application to Reward Shaping. In *Proc. Int. Conf. on Machine Learning (ICML)*. 278–287.

P. Palanisamy. 2020. Multi-Agent Connected Autonomous Driving using Deep Reinforcement Learning. In *Proc. Int. Joint Conf. on Neural Networks (IJCNN)*. 1–7.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research (JMLR)* 12 (2011), 2825–2830.

S. Safavian and D. Landgrebe. 1991. A Survey of Decision Tree Classifier Methodology. *IEEE Transactions on Systems, Man, and Cybernetics* 21, 3 (1991), 660–674.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. 2017. Proximal Policy Optimization Algorithms. Technical Report. http://arxiv.org/abs/1707.06347.

V. Sheng and J. Zhang. 2019. Machine Learning with Crowdsourcing: A Brief Summary of the Past Research and Future Directions. In *Proc. 33rd AAAI Conf. on Artificial Intelligence (AAAI)*.

D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, and S. Dieleman. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (2016), 484–489.

R. Sutton and A. Barto. 1998. *Introduction to Reinforcement Learning*. MIT press Cambridge.

R. Sutton and A. Barto. 2018. *Reinforcement Learning: An Introduction*. MIT press.

C. Tessler, D. Mankowitz, and S. Mannor. 2018. Reward Constrained Policy Optimization. Technical Report. https://arxiv.org/abs/1805.11074.

P. Thomas, B. Castro da Silva, A. Barto, S. Giguere, Y. Brun, and E. Brunskill. 2019. Preventing Undesirable Behavior of Intelligent Machines. *Science* 366, 6468 (2019), 999–1004.

H. Van Hasselt, A. Guez, and D. Silver. 2016. Deep Reinforcement Learning with Double Q-Learning. In *Proc. 30th AAAI Conf. on Artificial Intelligence (AAAI)*.

A. Vidali. 2019. Deep Q-Learning Agent for Traffic Signal Control. https://github.com/AndreaVidali/Deep-QLearning-Agent-for-Traffic-Signal-Control

H.-N. Wang, N. Liu, Y.-Y. Zhang, D.-W. Feng, F. Huang, D.-S. Li, and Y.-M. Zhang. 2020. Deep Reinforcement Learning: A Survey journal=Frontiers of Information Technology & Electronic Engineering. , 1726–1744 pages.

E. Wiewiora. 2010. *Reward Shaping*. Springer, 863–865.

E. Wiewiora, G. Cottrell, and C. Elkan. 2003. Principled Methods for Advising Reinforcement Learning Agents. In *Proc. 20th Int. Conf. on Machine Learning (ICML)*. 792–799.

D. Ye, G. Chen, W. Zhang, S. Chen, B. Yuan, B. Liu, J. Chen, and et al. 2020. Towards Playing Full MOBA Games with Deep Reinforcement Learning. Technical Report. https://arxiv.org/abs/2011.12692.

H. Zhang and Y. Guo. 2022. Safe Reinforcement Learning with Contrastive Risk Prediction. Technical Report. https://arxiv.org/abs/2209.09648.

Received 2023-09-28; accepted 2024-04-16