2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
September 27 - October 1, 2021. Prague, Czech Republic

2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) | 978-1-6654-1714-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/IROS51168.2021.9636759

XAI-N: Sensor-based Robot Navigation using Expert Policies and Decision Trees

Aaron M. Roth
Department of Computer Science
University of Maryland
College Park, Maryland, USA
amroth@umd.edu

Jing Liang
Department of Computer Science
University of Maryland
College Park, Maryland, USA

Dinesh Manocha
Department of Computer Science
University of Maryland
College Park, Maryland, USA

***Abstract*—We present a novel sensor-based learning navigation algorithm to compute a collision-free trajectory for a robot in dense and dynamic environments with moving obstacles or targets. Our approach uses deep reinforcement learning-based expert policy that is trained using a sim2real paradigm. In order to increase the reliability and handle the failure cases of the expert policy, we combine with a policy extraction technique to transform the resulting policy into a decision tree format. We use properties of decision trees to analyze and modify the policy and improve performance of navigation algorithm including smoothness, frequency of oscillation, frequency of immobilization, and obstruction of target. Overall, we are able to modify the policy to design an improved learning algorithm without retraining. We highlight the benefits of our approach in simulated environments and navigating a Clearpath Jackal robot among moving pedestrians. (Videos at this url: https://gamma.umd.edu/researchdirections/xrl/navviper)**

Learning methods are increasingly being used for robot navigation. Methods including Deep Reinforcement Learning (DRL) [1], learning from demonstration [2], imitation learning, etc. are able to integrate well with sensor data and have been used for navigation in real-world scenarios. They can work well in dense environments with multiple dynamic obstacles. However, when trying out a policy in a new environment or with a different configuration of obstacles, it can fail in simulation or in the real world, with failure modes including collisions, oscillatory behaviors/non-smooth paths, or agent-induced immobilization ("freezing"), among others [3]–[9].

The policies that result from a learning method like DRL are typically opaque as to their inner workings, and cannot easily be directly analyzed or modified without revising the method and repeating the time-consuming training step. Furthermore, it is hard to predict when errors would occur without running the policy.This makes it difficult to have any confidence or reliability in future performance, especially if the operating environment differs from the training data.

There is considerable interest in developing explainability and interpretability in deep learning and reinforcement learning methods [10], [11]. The ultimate goal is to develop methods and AI techniques such that the results of the solution can be understood by humans. This is in contrast with the most widely used machine learning methods that tend to act like a black box and even the designers cannot explain why the underlying method arrived at a specific decision. Our main goal is to design explainable algorithms for robot navigation, where we can offer some insights about their performance in different scenarios. In this context, we address the problem of modifying the policy to improve the performance of learning-based navigation methods.

This work was supported in part by ARO Grants W911NF1910069, W911NF2110026 and U.S. Army Grant No. W911NF2120076

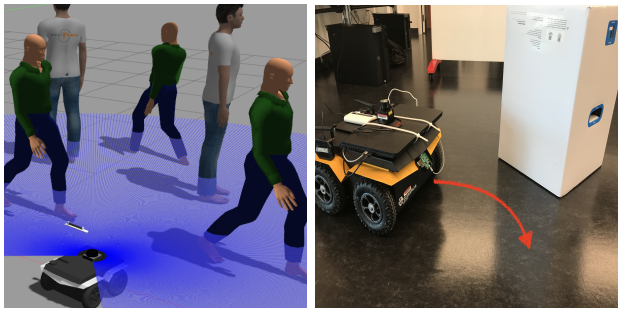

Fig. 1. Robot Navigation using XAI-N: The left image corresponds to training scenarios used for training a DRL policy; right image is testing of the tree policy demonstrated on a Clearpath Jackal robot. XAI-N generates an interpretable tree policy that allows us to identify and fix failure modes of the DRL policy without retraining. This results in improved navigation behavior, including fewer oscillations or freezing problems in dynamic environments.

Main Results: We introduce a novel scheme, XAI-N, which integrates the concepts of expert policies, policy extraction, and decision trees and utilizes them to make policy modifications that improve navigation in dynamic environments. Our approach first learns a navigation policy using DRL. We then transform this neural net policy into a decision tree (DT) policy using an imitation learning policy extraction method. A decision tree is a flowchart like tree structure (a binary tree in our case) that classifies (or maps) a space of numerical features into subsections (leaves) corresponding to classes. We use a tree where the features correspond to sensor inputs and other aspects of a state space and the leaf classifications correspond to discrete action choices, allowing us to use a tree as policy for robot navigation. DTs are inherently interpretable, as every output of a decision tree is tractable [12]. Once transformed into a DT, we show the navigation policy's structure can be

analyzed, interpreted, and modified to design an improved navigation algorithm.

We take advantage of the tree structure to detect and address suboptimalities in the policy and improve the navigation across several metrics: i) smoothness/oscillation frequency, ii) freezing frequency, iii) total path length, iv) blocking/obstruction occurrence, and v) reward per timestep (a scalar measure combining multiple of the preceding metrics). In this manner, we get the best of both worlds–the ability of the DRL to learn complex tasks and handle sensor data, and the comprehensible malleability of the decision tree. The novel contributions of our paper include

1. XAI-N, robot navigation learning method that combines traditional DRL learning with rule-based domain-specific algorithms.
2. Take advantage of the extracted tree structure to improve overall navigation scheme:
   a Detect situations that could cause "freezing" and modify policy to preclude such failure cases
   b Observe when oscillation occurs and modify policy to smooth the path, decreasing oscillation.
   c Prevent robot obstructing a human it is following

We highlight the benefits of our approach in many simulated scenarios and on a Clearpath Jackal robot navigating among obstacles and pedestrians.

I. Related Work

A. Learning for Navigation

The last decade has seen the rise of learning-based robot navigation algorithms [13]–[16], which can directly handle the real-world representations captured using commodity visual sensors. This enhances the ability of a robot to adapt and reach the goal even in new, unknown environments. Some of the widely used methods are based on Deep Learning (DL) or Reinforcement Learning (RL) [14], [17], [18]. Xie et al. [19] trained a network to convert RGB images to depth images and then used deep double-Q network(D3QN) algorithm to navigate the robot avoiding collisions [20]. In order to perform dynamic obstacles avoidance, Everett et al. [14] proposed a strategy, GPU/CPU Asynchronous Advantage Actor-Critic for collision avoidance with Deep RL(GA3C-CADRL), using LSTM. Lötjens et al. [21] developed an uncertainty-aware navigation method to avoid pedestrians. A common limitation of all these learning methods is that the black-box properties of neural networks make it hard to modify them, except by attempting to retrain or develop an improved learning method.

B. Policy Extraction and Imitation Learning

Initiation Learning [22], [23] involves learning a policy via copying an existing "expert" policy or deriving a policy that best fits observed procedure (learning from demonstration [2]). Policy Extraction (also called Policy Distillation) is the process of taking an existing trained policy and transforming it into a different format. This could be transforming a neural network into a smaller neural network [24] or turning a neural network into some other format such as a tree. [25] VIPER [26] is an algorithm which learns an "expert" policy using a neural net (such as PPO [27]) and then uses imitation learning to fit a decision tree to replicate the expert policy. VIPER has been used to generate decision tree policies for proof-of-concept problems such as CartPole [28], Atari Pong [26] and other simulations such as CARLA [29]. Our approach also uses VIPER.

C. XAI and Analyzing or Utilizing Decision Trees

Motivated by the desire to understand the sometimes opaque and inscrutabble nature of many advanced deep learning methods, Explainble AI (XAI) is a growing area of exploration [10], [11]. One class of XAI methods is that of globally intrinsic [30] explanation methods, such as decision trees. There is prior work on using a directly interpretable structure such as a tree or graph [31]. Previous authors have used decision trees in conjunction with RL. A deep neural network can be distilled into a soft decision tree [32], or learned via RL using Policy Tree [33]. However, neither of these methods are interpretable. Some methods such as the Pyeatt Method [34] and Conservative Q-Improvement [35] use an RL method to learn a decision tree in an additive manner. Decision trees, while hard to learn, are attractive as policies because they yield benefits in terms of interpretation and verifiability. DT are also well-suited for safety-critical applications because there are a range of standard techniques (such as Z3 [36]) that can be used to perform verification analysis on them [37].

There is work on modifying decision trees to better fit a dataset, such a simplification [38] or pruning [39]. This could result in loss of accuracy, and it is more about changing structure without impeding performance than it is about improving performance. There is also work on adapting a DT from one task to another. [40] Excluding a paper on classification [41], and retraining a tree after modifying a dataset, we found no prior work on tree modification for the purpose of addressing a specific domain goal as we do.

II. Problem Setup and Overview

A. Problem Setup

We model our navigation task as a Partially Observable Markov Decision Process (POMDP), which is represented by a tuple (Z, S, A, P, R, γ). Z is the real state space, S is the observed state space, A is the action space, R is rewards, and P is the state transition dynamics: $S \times A \to S$, and $\gamma \in (0, 1]$ is a discount factor. Our goal is to generate an optimal policy π which maximizes the discounted reward function:

$$\eta(\pi) = \mathbb{E}_\pi \left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t) \right]. \quad (1)$$

The goal of the task is to make a robot learn to efficiently go to its goal position and at the same time avoid collisions with obstacles. Obstacles can be static or dynamic. The robot performs local navigation using only what is observed by the sensors and knowledge of the most recently taken action. At each step, a robot knows the goal position relative to itself (due

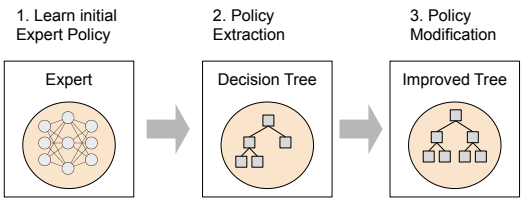

Fig. 2. Our XAI-N Robot Navigation Algorithm with three stages: In stage 1, we train an expert policy π^* with a learning method such as DRL. In stage 2, we perform policy extraction to decision tree policy $\pi^\dagger$. In stage 3, we apply modifications (such as the oscillation fix and freezing fix) to correct errors and improve the policy with regards to navigation metrics. This results in an improved interpretable navigation policies as compared to DRL without retraining.

to sensors). The episode ends when the goal state is reached (to within a tolerance) or a collision occurs. We develop an XAI method to address this type of robot navigation problem.

B. XAI Robot Navigation Algorithm Overview

A diagram of our XAI-N process is shown in Figure 2. In the first stage, a robot navigation policy is learned as an "expert policy". This process is discussed briefly in Section III-A, and the resulting policy is referred to as the "expert policy." Next, some appropriately chosen Policy Extraction [24], [25] or Imitation Learning [22] process is used to transform the expert policy into a decision tree format policy. A DT policy $\pi^\dagger$ uses a DT to perform the mapping of state s to action a. This is described in detail in Section III-B and illustrated in Figure 3. (Learning a DT directly on a complex environment is often too time-consuming or difficult to be feasible. XAI-N enables utilizing an optimal initial learning method whilst taking advantage of tree structure after imitation.) Finally, the third stage is the modification stage where the policy is augmented. These augmentations, discussed in Section III-C, can potentially improve the policy performance on several different

$s \in S$	a state s is an array representing the state of the world and the robot in it
$a \in A$	an action a is a single discrete action in the set of possible actions A
a_F	the "stop" action
a_L	a "rotate left" action
a_R	a "rotate right" action
a_D	a "forward" action
$\mathbf{C}_F$	a set of column indices of polar columns in front of robot
$\mathbf{A}_D$	the set of actions containing a component of forward movement
P	state transition dynamics
r, R	reward (for a single (s, a) pair or in general, respectively)
γ	future discount factor
E	An environment (real or simulated). Receives an action a and provides the perceived state of the world s, reward r, and boolean indication of whether the goal has been reached ("done").
$\pi : (s \to a)$	a policy, mapping a state s to action a
π^*	an expert policy (neural net in our work, but can be anything)
$\pi^\dagger$	a decision tree policy
$F^\dagger(\pi)$	policy extraction conversion function, outputs $\pi^\dagger$
m_A	how much movement to allow in a "static" s during freezing detection
$\mathcal{F}_O$	takes a history of (s, a) pairs and outputs boolean indicating whether oscillation has occurred or not
$\mathcal{N}$	a set of nodes with errors detected
$\mathcal{O}_C(i)$, where $n_i \in \mathcal{N}$	set of states in state subspace of node i where oscillation occurs
$\mathcal{O}_X(i)$ where $n_i \in \mathcal{N}$	set of states in state subspace of node i where oscillation does not occur

TABLE I
SYMBOLS AND NOTATION USED IN THE PAPER

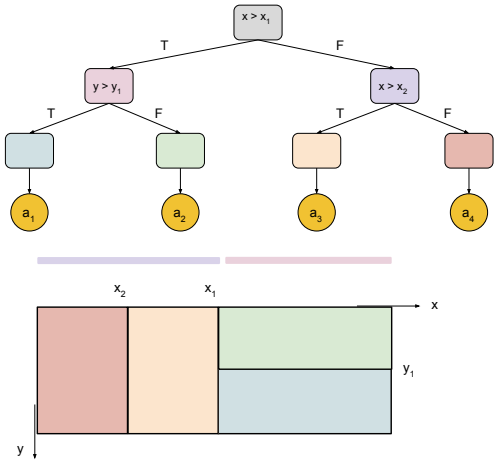

Fig. 3. This figure demonstrates how a decision tree can be used as a policy for robot navigation. The rounded rectangles are branch and leaf nodes corresponding to abstract states. Each abstract state is a subset of the robot's state space, and any given state s falls inside the bounds of *exactly* one leaf node abstract state (as well as the abstract states of all parent nodes to that leaf node). The root node has abstract space equivalent to the entire state space. Yellow circles are actions (a_i) (classes). In this example, a two-dimensional state-space is shown below the tree as a rectangle. The space is divided up by the tree. The colors demonstrate how the tree subdivides the state space into abstract states. A single s_j would correspond to a single point within the bounds of the rectangle. Whichever leaf node / smallest divisible rectangle this state falls into will guide which action class the tree policy outputs for that state. This example uses arbitrarily chosen points of x_1, x_2, y_1 on two dimensions x and y. Features could include meaning such as distance from or direction to a goal location, or the presence of obstacles, and the robot would take different actions in each case. Our actual environmental setup involves a policy with 213 state dimensions and 6 action classes (or 10 action classes after the oscillation-fixing procedure in Section III-C2).

navigation metrics, in some cases beyond that achieved by the expert policy.

III. Approach: XAI-N Learning Method

A. Initial Learning Methods

The first stage can use any method of policy generation. For example, it could be created via any robot motion planning algorithm (eg. Sampling Based or Optimization-based algorithm) [42] or using reinforcement learning [43]. The important aspect of the first stage is that it can encompass any existing method that results in a robot policy. We define an **expert policy** $\pi^* : (s \to a)$ as a function or object that maps from a state $s_i \in \mathcal{S}$ to an action $a_i \in \mathcal{A}, \forall s_i \in \mathcal{S}$ where $\mathcal{S}$ is the set of all possible states and $\mathcal{A}$ is a set of possible actions. Thus at every timestep t, the robot can observe state s_t, query π^* to determine action a_t to take, take that action, receive a new observation s_{t+1}, and repeat.

To generate our expert policy, we use Deep Reinforcement Learning, specifically Proximal Policy Optimization [27] in a Curriculum Learning (CL) pipeline. [44] We follow the procedure used in [45].

B. Extraction to Decision Tree

In the second stage, the expert policy is transformed from its current format into a decision tree format, called the

extracted policy $\pi^\dagger$. We chose a decision tree as opposed to a regression algorithm because we want this stage to be more interpretable and modifiable. The conversion process $F^\dagger$ converts $\pi^\dagger = F^\dagger(\pi^*)$. Like π^*, $\pi^\dagger$ maps from states to actions, but whereas the expert policy can have any internal structure (neural net, ensemble method, mixture of trees, planning algorithm, arbitrary code, etc) so long as it performs the $s \rightarrow a$ mapping, we constrain the decision tree to use a particular format, shown in Figure 3. The features of the decision tree correspond element-by-element to the features of the *state space*, which is the term to describe S, or the space of all possible states. A single state s can be represented by an array of numbers. If the robot's raw observation is in a different format, such as a camera image, this input can be flattened or preprocessed into such an array. S can be described by two arrays each equal in length to an s-array, and describing upper and lower bounds on the total state space. A *state subspace* (or an abstract state) is a subset of the state space, and can be similarly described by upper and lower bounds that demarcate a smaller space inside S. The elements of s represent features, and these features are the features of the tree. Each branching node of the tree thus splits on one feature of the state subspace, splitting it into two further subspaces, as shown. Each leaf node's class label corresponds to an action (this can be a discrete action or an action probability distribution).

There are a number of ways to perform policy extraction (or policy distillation). We use the VIPER family of methods because they result in a single tree policy and are applicable regardless of the internal structure of the expert policy [26]. In VIPER and its extensions, the expert policy π^* is executed in the environment E. Each timestep, a state s_t is observed and an action a_t is chosen by π^*. We associate (s_t, a_t) together as a "state-action pair." The environment E after receiving a_t provides updated state s_{t+1}, and the cycle repeats until the goal is reached. This is called an "episode," and multiple episodes are run, producing trajectories, or sequences of state-action pairs $\{(s_i, a_i), (s_{i+1}, a_{i+1}), ...\}$. These trajectories can be combined into a dataset of state-action pairs. Multiple datasets of state-action pairs are sampled from the total pairs generated. These datasets are used in a supervised learning manner to learn a decision tree policy $\hat{\pi}_i$ using the CART method [46], with the state forming the features and the actions forming the labels. The resulting DT is a binary tree, where branch nodes test a condition regarding the feature space (which is the state space), and leaf nodes represent discrete action classes. A diagram of this is shown in Figure 3. Whichever policy performs the best (as determined by which policy achieves the maximum average reward on a series of trials) is regarded as the best decision tree policy $\pi^\dagger$. Reward is a property of environment E and is constructed as a scalar that serves as a combined measure of the degree to which a robot is achieving certain navigation metrics. We use an E where reward increases for reaching the goal or following a target, and doing so smoothly and quickly (described more in Section IV-A), such that $\pi^\dagger$ most closely achieves the levels achieved by π^* on these navigation metrics.

C. Modification Methods

In the third stage, we introduce modifications to improve the robot's ability to reach the goal without colliding with obstacles or freezing, to increase overall trajectory smoothness by reducing oscillation, and to avoid obstructing a human. Modifications targeting other navigation metrics could also be developed using a similar approach to what we have developed here. A neural net format policy would not be able to be modified in the manner described in the following sections, hence the appeal of the DT.

1) Fix Freezing: One of the standard issues with navigation learning methods is "freezing." The robot chooses to remain immobile in the face of certain obstacles. Naturally, freezing helps prevent crashing, but the robot is also no longer moving towards the goal. In particular, when the given obstacles are static, it is a failure mode from which it cannot escape.

We present a method to identify nodes in the tree that could be contributing to the freezing issue, and then modify those nodes to mitigate the danger of such an error occurring. The procedure for identifying nodes is shown in Algorithm 1

Algorithm 1: Detect Freezing

```
Detect Freezing Nodes( π†, a_F, m_A ):
N ← ∅;
for node n ∈ π† do
    if n is a leaf node then
        m_C ← the number of cells in the occupancy
          grid in which movement occurs;
        if m_C < m_A and n[action] = a_F then
            Add n to N;
        end
    end
end
Return N
```

where $\pi^\dagger$ is the tree policy, a_F represents an action or grouping of actions corresponding to the "stop" action, and m_A is a tunable integer parameter indicating "in how many cells in the occupancy grid should movement be allowed while still declaring the obstacles stationary." (See section IV-A to explain the occupancy grid.) The algorithm checks each leaf node of the tree. If the obstacles detected are stationary within some tolerance indicated by m_A and if the node's action is the Stop action, then the node is added to the list of problematic potential-freezing nodes. The m_A parameter is included because in some situations we may not want to be completely strict about everything being perfectly still. Setting $m_A = 0$ requires perfect stillness to consider a node a freezing possibility and setting m_A to the maximum means the algorithm will return all nodes with the stop action a_F regardless of obstacle position and movement. If a node's subspace dimensions encompass both moving and non-moving situations, the condition will be true for the purposes of this

algorithm in the case that the bounds of those dimensions are unchanged for all timesteps (since even though movement could occur sometimes, the case where an obstacle is still is also included in this subspace). The algorithm intended to alleviate this issue is found in Algorithm 2, where a_R and a_L are actions corresponding to pure right and left rotation (no linear velocity) respectively. This safely allows the robot to find an observed state where it can extract itself from stasis.

Algorithm 2: Alleviate Freezing

Modify Freezing Nodes($\pi^\dagger, \mathcal{N}, a_R, a_L$):
for *node* $n \in \mathcal{N}$ **do**
  if *majority of obstacles are on the right* **then**
    n[action] $\leftarrow a_L$;
  else
    n[action] $\leftarrow a_R$;
  end
end
Return the updated $\pi^\dagger$

2) Fix Oscillation: Another observed issue with some of the expert policies was oscillation. When seeking to circumvent certain obstacles, the robot would alternate between turning too far away from and towards the obstacle, resulting in aesthetically displeasing and inefficient behavior. We developed a fix that involves running the policy in simulation and observing it to identify parts of the tree policy that contribute to the oscillation, and modifying the tree by adding nodes or modifying existing nodes to involve new actions with lower linear and angular velocities. Detecting problematic nodes is done using Algorithm 3, where E is an environment, $\mathcal{F}_O$ is a function that takes in a history of state-action pairs and outputs a boolean indicating whether oscillation has occurred or not, L is the length of that history, and n_e is the total number of episodes to observe. The modification procedure to correct this error is shown in Algorithm 4. Nodes with subspaces that correspond to instances of oscillation are split, with the child leaf node corresponding to that subspace assigned a lower magnitude velocity action, and the sibling leaf node assigned the action of the original node.

In Algorithm 4, each $\mathcal{O}_C(i) \in \mathcal{O}_C$ is a set of states in state subspace of node i where oscillation occurs, and each $\mathcal{O}_X(i) \in \mathcal{O}_X$ is a set of states in state subspace of node i where oscillation does not occur, and z is a boolean.

3) Fix Blocking/Obstruction: In the warehouse environment, where the robot locates and follows a human, we found that the robot sometimes would place itself in the human's path, blocking the human. This is inefficient, and would be annoying or dangerous in real life. Find the algorithm used to detect potential nodes contributing to this situation in Algorithm 5, where $\pi^\dagger$ is the tree policy, a_F represents an action or grouping of actions corresponding to the "stop" action, m_A is a tunable parameter and $\mathbf{A}_D$ is the set of actions that imply no blocking is occurring (ie all the movement actions with a forward component. The algorithm intended

Algorithm 3: Detect Oscillation

Detect Oscillation Nodes($\pi^\dagger, E, \mathcal{F}_O, n_e, L$):
$\mathbb{H}^L \leftarrow$ initialize an empty queue;
$\mathcal{N} \leftarrow \varnothing$;
$\mathcal{O}_C(i) \leftarrow \varnothing \quad \forall \quad i \in \{$ ids of nodes in $\pi^\dagger\}$;
$\mathcal{O}_X(i) \leftarrow \varnothing \quad \forall \quad i \in \{$ ids of nodes in $\pi^\dagger\}$;
for n_e *episodes* **do**
  Reset environment E;
  while *E does not indicate episode done* **do**
    $s \leftarrow$ get current state from E;
    $a \leftarrow \pi^\dagger(s)$;
    Append (s, a) to $\mathbb{H}^L$, removing the oldest if the length of the queue is $> L$;
    if $\mathcal{F}_O(\mathbb{H}^L)$ **then**
      $\{n_i, n_{i+1}, ..., n_{i+L}\} \leftarrow$ leaf nodes in $\pi^\dagger$ corresponding to each $s \in \mathbb{H}^L$;
      Add $\{n_i, n_{i+1}, ..., n_{i+L}\}$ to $\mathcal{N}$;
      Add all $s \in \mathbb{H}^L$ to $\mathcal{O}_C(d_i), \mathcal{O}_C(d_{i+1}), ..., \mathcal{O}_C(d_{i+L})$, where d_d is the corresponding id of each node in $\{n_i, n_{i+1}, ..., n_{i+L}\}$;
    else
      $n_i \leftarrow$ leaf node in $\pi^\dagger$ corresponding to s;
      Add s to $\mathcal{O}_X(d)$, where d is the id of n_i;
    end
    Execute action a in environment E;
  end
end
Return $\mathcal{N}, \mathcal{O}_C(i), \mathcal{O}_X(i)$;

Algorithm 4: Alleviate Oscillation

Alleviate Oscillation Nodes($\pi^\dagger, \mathcal{N}, \mathcal{O}_C, \mathcal{O}_X, z$):
// Note that all $n \in \mathcal{N}$ are in $\pi^\dagger$
for $n_i \in \mathcal{N}$ **do**
  if $\mathcal{O}_X(i) = \varnothing$ *or* z **then**
    // All states visited on this node are oscillation
    n_i[action] $\leftarrow$ action with linear and angular velocity of reduced magnitude;
  else
    $X \leftarrow$ new leaf node with action n_i[action];
    $C \leftarrow$ new leaf node with action with linear and angular velocity of reduced magnitude compared to n_i[action];
    n_i is turned into a branch node with X and C as children, splitting on the best split that best separates the states in $\mathcal{O}_C(i)$ to node C and states in $\mathcal{O}_S(i)$ to node X;
  end
end
Return updated $\pi^\dagger$;

Algorithm 5: Detect Blocking

Detect Blocking Nodes($\pi^\dagger, \mathbf{A}_D, m_A$):
$\mathcal{N} \leftarrow \varnothing$;
for *node* $n \in \pi^\dagger$ **do**
  if *n is a leaf node* **then**
    $m_C \leftarrow$ the number of cells in the occupancy grid in which movement occurs;
    if $m_C < m_A$ *and* $n[action] \notin \mathbf{A}_D$ **then**
      Add n to $\mathcal{N}$;
    end
  end
end
Return $\mathcal{N}$

to alleviate this issue is found in Algorithm 6, where a_R and

Algorithm 6: Alleviate Blocking

Modify Blocking Nodes($\pi^\dagger, \mathcal{N}, a_L, a_R, a_D, \mathbf{C}_F, n_r$):
for *node* $n \in \mathcal{N}$ **do**
  clear_in_front $\leftarrow$ a boolean true if the columns in $\mathbf{C}_F$ are clear for a distance of n_r rows, starting from the nearest row;
  if *clear_in_front* **then**
    n[action] $\leftarrow a_D$;
  else
    if *majority of obstacles are on the right* **then**
      n[action] $\leftarrow a_L$;
    else
      n[action] $\leftarrow a_R$;
    end
  end
end
Return the updated $\pi^\dagger$

a_L are the right and left rotation actions, a_D is the "forward" action, $\mathbf{C}_F$ are indices of the columns of the polar grid directly in front of the robot (encompassing a traversable expanse, such that the robot could proceed forward into that region without collision), and n_r is how far ahead to look in number-of-rows when checking whether those columns are occupied.

In this manner, the robot's policy is changed so that in situations where it might be stuck, it seeks open space and moves there, presumably away from a near obstacle which may or may not be a human.

IV. Evaluation

A. Environments

We demonstrate our improvements on two environments.

1) Mobile Robot Navigation: The robot starts in a random location and must navigate around obstacles to a random goal location. Obstacles can be static or dynamic. We desire that the policy should perform well in terms of avoiding the pedestrians and obstacles. We created a simulation of this environment and also test in a real-world setup, in both cases with a Clearpath Jackal. We formulate the environment as an AI Gym [47], a common RL interface for environments, and release it as open-source code for others to use as well [48].

XAI-N Stage	Policy Type	Avg Reward per timestep	% crash	% freeze	Oscill-ation %	Avg Osc. length	Path Length(m)
1	Expert (PPO)	0.226	0%	0%	100.%	8.07	9.94
2	M-VIPER	-0.276	67%	0%	95%	1.73	8.18
3	M-VIPER + Oscillation Fix (XAI-N)	**0.241**	4%	0%	**6%**	**1.33**	8.29

TABLE II

Policy at different stages of our XAI-N algorithm. At the intermediate stage 2 (Figure 2), average reward per timestep decreases due to crashing, this is resolved by stage 3, where crashing decreases, average reward per timestep is higher than preceding stages including stage 1 expert policy, and the 100% oscillation from stage 1 is reduced to 6%. Overall, we design an improved learning-based navigation algorithm.

2) Game Character Locomotion and Animation: In this environment, the agent is a character which spawns in a complex multi-room environment with obstacles and other characters with which to interact. There are three stages in this game: i) learning to exit the room, ii) learning to exit the room and finding another certain autonomous character, iii) following this other character as they move.

3) Sensors and State Space: The sensor setup in both cases involves lidar and a pozyx system (an ultra-wideband based localization system) [49]. The state space contains information about the goal location (relative to the robot) in polar coordinates, the previous robot action, and the physical surroundings of the robot as sensed by the lidar. The lidar we use scans 512 ranges from $-\frac{2}{3}\pi$ to $\frac{2}{3}\pi$ radians (with 0 radians corresponding to straight ahead). We transform this into a radial occupancy grid. In our implementation of this benchmark we use a grid with 10 evenly spaced columns, and rows start 10 cm from the center of the robot, with distances of the 7 rows as (listed in order from nearest to farthest from the robot) 0.2 m, 0.2 m, 0.2 m, 0.3 m, 1 m, 1 m, 1 m. The state space contains the occupancy grid information from the current time step and previous two timesteps. There are thus 210 features describing obstacle position and movement, 2 features indicating relative goal position, and 1 feature indicating the previous action chosen by the agent (for a total of 213 features). The action space is a discrete action space: 1) Forward and Left, 2) Rotate Left, 3) Straight Forward, 4) No movement, 5) Forward and Right, 6) Rotate Right. (We also implemented an expanded action space that contains four additional actions that correspond to actions 1, 2, 5, and 6 but with smaller magnitude velocities.)

We design the reward function with three major parts, as follows:

$$r_i^t = (r_g)_i^t + (r_c)_i^t + (r_o)_i^t. \tag{2}$$

where $(r_g)_i^t$ rewards movement towards and reaching the goal or person, $(r_c)_i^t$ penalizes collisions with or proximity to obstacles, $(r_o)_i^t$ penalizes oscillations and rewards smoothness.

We used Gazebo 9.0 simulator with ROS Melodic on Ubuntu 18.04 to create multiple scenarios with different types and layouts of the obstacles.

B. Results

Find our detailed results in Table II, and you can also view the accompanying video for a live demonstration. CrowdEnv scenario 10 was used to test and produce the data. Scenario 10 was **not** a configuration of obstacles that any of the policies saw during their training. "Path Length" is an average of total path lengths, counting only those runs where the robot successfully reached the goal. The expert policy π^* (labeled "PPO" in reference to the DRL training method used) demonstrates an average of over 8 meters of oscillation per run, and oscillated during every run. The policy after the conversion to decision tree $\pi^\dagger$ is noted as Modified VIPER (M-VIPER). Generally the fixed decision tree inherits the optimality regarding path length of training based algorithm and also improves the performance of navigation regarding the specific issues of decreasing crash rate, freeze rate and oscillations where they occur.

Reduced Oscillations: To demonstrate the oscillation fix, we chose one of the extracted $\pi^\dagger$ that still had a significant amount of oscillation after extraction. Labeled as "M-VIPER", we see it has an oscillation 95% of the time, and an oscillation length of 1.73. Firstly, something interesting to note is that the extraction process itself reduced the length of oscillation significantly. At this intermediate stage, the average reward per timestep decreases since it crashes more than expert, due to imperfect imitation. After performing our oscillation fix (with $n_e = 20, L = 5, z =$true) , we obtain the policy shown in the "M-VIPER + Oscillation Fix" (i.e. after Stage 3 of our XAI-N approach), The oscillation fix procedure identified 11 nodes in the DT that might be contributing to oscillation, and applied the fix to them. The crashing issue is resolved. In the policy, oscillation occurs in only 6% of runs, and has a oscillation length reduced to an average cumulative of 1.33 m in those rare instances where it does occur. Find an illustration in Figure 4. This is a significant reduction in oscillation beyond that achieved by the standalone DRL method, despite the fact that the reward function for the DRL method included a parameter to reward smoothness (decreasing oscillation).

Eliminate Freezing: We applied the freezing fix to a different M-VIPER policy as shown in Table III. This policy would freeze 28% of the time. The freezing fix identified 30 nodes that may have contributed to the error, and modified them accordingly (out of a total of 621 nodes, 311 of which are leaf nodes). After applying the freezing fix, freezing was eliminated using our XAI-N approach.

XAI-N Stage	Policy Type	% freezing
2	M-VIPER	28%
3	M- VIPER + Freezing Fix	0%

TABLE III

OUR METHOD DETECTED FREEZING AFTER IMITATION LEARNING IN A STAGE 2 M-VIPER. WE RAN THE FREEZING FIX TO CORRECT THIS ISSUE. OUR RESULTING LEARNING-ALGORITHM EXHIBITS NO FREEZING ISSUES.

XAI-N Stage	Policy Type	Avg % blocking
2	M-VIPER	0.7
3	M- VIPER + Blocking Fix	0.0

TABLE IV

OUR METHOD DETECTED POTENTIAL BLOCKING BEHAVIOR AFTER IMITATION LEARNING IN THE WAREHOUSE ENVIRONMENT. WE RAN THE BLOCKING FIX TO CORRECT THIS ISSUE AND OUR LEARNING ALGORITHM EXHIBITS NO BLOCKING BEHAVIOR.

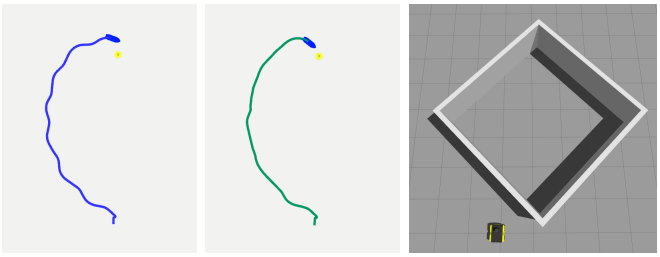

Fig. 4. At right, an obstacle the robot must circumnavigate. At left, the robot's path before the oscillation fix is applied (i.e, DRL expert policy). At center, the robot's path after oscillation fix is applied using policy extraction and DTs.

Eliminate Blocking: We applied the blocking fix to a warehouse policy that exhibited blocking, as shown in Table IV. The blocking fix identified 381 nodes to potentially modify out of 1559 nodes total. 151 nodes were adjusted to move the robot forward, and the other 230 were given rotation actions to orient the robot in a manner where it could more safely move out of the human's way.

The blocking fix is an example of a case of trade-offs. Although blocking was eliminated, it decreased the efficiency of the path (increasing average trajectory length from 9.45 to 17.3). In the pure navigation environment, this would not be desirable. The warehouse environment, however, simulates a human-robot interaction scenario, and in this situation one can imagine a preference for safety and comfortable robot interaction, in comparison to "most-efficent" paths that might nip at a humans' heels or obstruct the human's path. This kind of domain-specific customization, based around a similar sensor scheme for a robot and similar learning procedures, demonstrates the usefulness of our paradigm.

V. CONCLUSION AND FUTURE WORK

We provide XAI-N, an improved learning algorithm for sensor-based robot navigation. Starting with training an expert policy (e.g trained by DRL), we extract a decision tree policy, the interpretable properties of which we utilize to modify the tree. This allows for improving smoothness of path, mitigating the chance of obstructing a human, and reducing the problem of freezing. We are able to modify the policy to address these imperfections without retraining, combining the learning power of deep learning with the control of domain-specific algorithms. We demonstrated fixes across two environments, a robot navigation among pedestrians and obstacles, and a warehouse game with an agent following a person.

One limitation is that the maximum speed of the dynamic obstacles should not be more than the maximum speed of the robot itself. Future work could address this, could include modification techniques for tackling additional navigation issues beyond freezing, oscillation, and blocking, or could combine XAI-N with other motion planning methods [50].

REFERENCES

[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[2] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and autonomous systems*, vol. 57, no. 5, pp. 469–483, 2009.

[3] T. Kahan, Y. Bukchin, R. Menassa, and I. Ben-Gal, "Backup strategy for robots' failures in an automotive assembly system," *International Journal of Production Economics*, vol. 120, no. 2, pp. 315–326, 2009.

[4] D. J. Brooks, "A human-centric approach to autonomous robot failures," Ph.D. dissertation, University of Massachusetts Lowell, 2017.

[5] M. Jain, P. Kumar, R. Kota, and S. N. Patel, "Evaluating and informing the design of chatbots," in *Proceedings of the 2018 Designing Interactive Systems Conference*, 2018, pp. 895–906.

[6] M. K. Lee, S. Kiesler, J. Forlizzi, S. Srinivasa, and P. Rybski, "Gracefully mitigating breakdowns in robotic services," in *2010 5th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. IEEE, 2010, pp. 203–210.

[7] C. G. Morales, E. J. Carter, X. Z. Tan, and A. Steinfeld, "Interaction needs and opportunities for failing robots," in *Proceedings of the 2019 on Designing Interactive Systems Conference*, 2019, pp. 659–670.

[8] D. Kontogiorgos, S. van Waveren, O. Wallberg, A. Pereira, I. Leite, and J. Gustafson, "Embodiment effects in interactions with failing robots," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–14.

[9] M. S. Ramanagopal, C. Anderson, R. Vasudevan, and M. Johnson-Roberson, "Failing to learn: autonomously identifying perception failures for self-driving cars," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3860–3867, 2018.

[10] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," *arXiv preprint arXiv:1702.08608*, 2017.

[11] F. Sado, C. K. Loo, M. Kerzel, and S. Wermter, "Explainable goal-driven agents and robots–a comprehensive review and new framework," *arXiv preprint arXiv:2004.09705*, 2020.

[12] L. O. Hall, N. Chawla, and K. W. Bowyer, "Decision tree learning on very large data sets," in *SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218)*, vol. 3. IEEE, 1998, pp. 2579–2584.

[13] L. Tai, J. Zhang, M. Liu, and W. Burgard, "Socially compliant navigation through raw depth inputs with generative adversarial imitation learning," in *ICRA*, May 2018, pp. 1111–1117.

[14] M. Everett, Y. F. Chen, and J. P. How, "Motion planning among dynamic, decision-making agents with deep reinforcement learning," in *IROS*. IEEE, 2018, pp. 3052–3059.

[15] P. Long, T. Fan, X. Liao, W. Liu, H. Zhang, and J. Pan, "Towards Optimally Decentralized Multi-Robot Collision Avoidance via Deep Reinforcement Learning," *arXiv e-prints*, p. arXiv:1709.10082, Sep 2017.

[16] A. J. Sathyamoorthy, J. Liang, U. Patel, T. Guan, R. Chandra, and D. Manocha, "Densecavoid: Real-time navigation in dense crowds using anticipatory behaviors," *arXiv preprint arXiv:2002.03038*, 2020.

[17] P. Mirowski, R. Pascanu, F. Viola, H. Soyer, A. J. Ballard, A. Banino, M. Denil, R. Goroshin, L. Sifre, K. Kavukcuoglu *et al.*, "Learning to navigate in complex environments," *arXiv preprint arXiv:1611.03673*, 2016.

[18] J. Zhang, J. T. Springenberg, J. Boedecker, and W. Burgard, "Deep reinforcement learning with successor features for navigation across similar environments," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 2371–2378.

[19] L. Xie, S. Wang, A. Markham, and N. Trigoni, "Towards monocular vision based obstacle avoidance through deep reinforcement learning," *arXiv preprint arXiv:1706.09829*, 2017.

[20] N. K. Govindaraju, M. C. Lin, and D. Manocha, "Quick-cullide: Fast inter-and intra-object collision culling using graphics hardware," in *IEEE Proceedings. VR 2005. Virtual Reality, 2005.* IEEE, 2005, pp. 59–66.

[21] B. Lötjens, M. Everett, and J. P. How, "Safe reinforcement learning with model uncertainty estimates," in *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 2019, pp. 8662–8668.

[22] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation learning: A survey of learning methods," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–35, 2017.

[23] J. Ho and S. Ermon, "Generative adversarial imitation learning," in *Advances in neural information processing systems*, 2016, pp. 4565–4573.

[24] A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell, "Policy distillation," *arXiv preprint arXiv:1511.06295*, 2015.

[25] A. Jhunjhunwala, "Policy extraction via online q-value distillation," Master's thesis, University of Waterloo, 2019.

[26] O. Bastani, Y. Pu, and A. Solar-Lezama, "Verifiable reinforcement learning via policy extraction," in *Advances in Neural Information Processing Systems 31*. Curran Associates, Inc., 2018, pp. 2494–2504. [Online]. Available: http://papers.nips.cc/paper/7516-verifiable-reinforcement-learning-via-policy-extraction.pdf

[27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv e-prints*, p. arXiv:1707.06347, Jul 2017.

[28] S. Bhupatiraju, K. K. Agrawal, and R. Singh, "Towards mixed optimization for reinforcement learning with program synthesis," *arXiv preprint arXiv:1807.00403*, 2018.

[29] D. Chen, B. Zhou, V. Koltun, and P. Krähenbühl, "Learning by cheating," in *Conference on Robot Learning*. PMLR, 2020, pp. 66–75.

[30] A. Alharin, T.-N. Doan, and M. Sartipi, "Reinforcement learning interpretation methods: A survey," *IEEE Access*, 2020.

[31] A. M. Roth, "Structured representations for behaviors of autonomous robots," Master's thesis, Carnegie Mellon University, Pittsburgh, PA, July 2019.

[32] N. Frosst and G. Hinton, "Distilling a neural network into a soft decision tree," *arXiv preprint arXiv:1711.09784*, 2017.

[33] U. Das Gupta, "Adaptive representation for policy gradient," ., 2015.

[34] L. D. Pyeatt, "Reinforcement learning with decision trees." in *Applied Informatics*, 2003, pp. 26–31.

[35] A. M. Roth, N. Topin, P. Jamshidi, and M. Veloso, "Conservative q-improvement: Reinforcement learning for an interpretable decision-tree policy," *arXiv preprint arXiv:1907.01180*, 2019.

[36] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[37] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A static analyzer for large safety-critical software," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003, pp. 196–207.

[38] L. A. Breslow and D. W. Aha, "Simplifying decision trees: A survey," *Knowledge engineering review*, vol. 12, no. 1, pp. 1–40, 1997.

[39] J. Eggermont, J. N. Kok, and W. A. Kosters, "Detecting and pruning introns for faster decision tree evolution," in *International Conference on Parallel Problem Solving from Nature*. Springer, 2004, pp. 1071–1080.

[40] J. won Lee and C. Giraud-Carrier, "Transfer learning in decision trees," in *2007 International joint conference on neural networks*. IEEE, 2007, pp. 726–731.

[41] M. J. Aitkenhead, "A co-evolving decision tree classification method," *Expert Systems with Applications*, vol. 34, no. 1, pp. 18–25, 2008.

[42] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.

[43] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[44] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning*. ACM, 2009, pp. 41–48.

[45] T. Fan, X. Cheng, J. Pan, D. Manocha, and R. Yang, "Crowdmove: Autonomous mapless navigation in crowded scenarios," *arXiv preprint arXiv:1807.07870*, 2018.

[46] R. J. Lewis, "An introduction to classification and regression tree (cart) analysis," in *Annual meeting of the society for academic emergency medicine in San Francisco, California*, vol. 14, 2000.

[47] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[48] A. M. Roth and J. Liang, "Jackal crowd env," https://github.com/AMR-/JackalCrowdEnv, 2021.

[49] K.-M. Mimoune, I. Ahriz, and J. Guillory, "Evaluation and improvement of localization algorithms based on uwb pozyx system," in *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE, 2019, pp. 1–5.

[50] D. Manocha, *Algebraic and numeric techniques in modeling and robotics*. University of California at Berkeley, 1992.